Getting Started with the DLR as a Library Author

Alex Turner and Bill Chiles

1 Introduction 2
2 ExpandoObject 2
3 DynamicObject 4
3.1 DynamicBag: Implementing Our Own ExpandoObject 5
3.2 NamedBag: Optimizing DynamicObject with Statically Defined Members 6
4 IDynamicMetaObjectProvider and DynamicMetaObject 8
4.1 FastNBag: Faster Bags for N Slots 8
4.1.1 BindSetMember Method 9
4.1.2 BindGetMember Method 10
4.1.3 GetDynamicMemberNames Method 13
4.2 Further Reading 14
5 Appendix 14
5.1 DynamicObject Virtual Methods 14
5.2 FastNBag Full Source 15

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 in this space easily too.

A key aspect of the DLR is a common dynamic object interoperability protocol. Language implementers targeting the DLR participate in the protocol and use the DLR's dynamic dispatch caching mechanism to make their dynamic objects fast and portable across languages. Library authors can participate in this protocol at a very high-level and still enjoy much of the efficiency that language implementers get. Library authors can also go further and engage at the same level languages do, but they usually do not need to do that.

For example, if you have a library for crawling through XML or working with JSON objects, you'd really like to enable your objects to appear as dynamic objects to C# 4.0, VB 10.0 and dynamic languages. This lets consumers write syntactically simpler and more natural code for accessing members, drilling into, and operating on your objects. They can write code like XMLElement.Customer.Name instead of XElement.Element("Customer").Element("Name"). Although the calls to Element in the second form are "statically typed", the tokens meaningful to you, Customer and Name, are passed as strings anyway, so the code has been dynamic in practice all along.

The DLR provides two high-level helper objects so that library authors do not have to work at the level of language implementers to support dynamic operations. These types are ExpandoObject and DynamicObject, which are explained below with samples.

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, 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 example, you could use instances of the ExpandoObject class to hold onto Name and Age values for the effect of dynamic or structural typing:

```
static void Main(string[] args) {
    dynamic student, teacher;

    student = new ExpandoObject();
    student.Name = "Billy";
    student.Age = 12;
```

```
    teacher = new ExpandoObject();
    teacher.Name = "Ms. Anderson";
    teacher.Age = "thirty";

    WritePerson(student);
    WritePerson(teacher);
}

private static void WritePerson(dynamic person) {
    Console.WriteLine("{0} is {1} years old.",
                      person.Name, person.Age);
}
```

This produces the following output:

```
Billy is 12 years old.
Ms. Anderson is thirty years old.
```

You can also store lambda expressions as delegate values to define first-class function members, which you can later invoke:

```
static void Main(string[] args) {
    dynamic employee = new ExpandoObject();

    employee.Name = "Mr. Smith";
    employee.Age = 42;
    employee.CelebrateBirthday
        = (Action<dynamic>)(person => { person.Age++; });

    WritePerson(employee);
    // Note: ExpandoObject doesn't support InvokeMember with an
    // implicit self parameter.  You fetch a member and supply
    // all arguments explicitly.
    employee.CelebrateBirthday(employee);
}

private static void WritePerson(dynamic person) {
    Console.WriteLine("{0} is {1} years old.",
                      person.Name, person.Age);
}
```

This produces the following output:

```
Mr. Smith is 42 years old.
Mr. Smith is 43 years old.
```

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 respects the case-sensitivity of the language binder interacting with it, binding to an existing member if there is exactly one case-insensitive match and throwing an AmbiguousMatchException if there are multiple.

To allow easy enumeration of its values, ExpandoObject implements IDictionary<String, Object>. Casting an ExpandoObject to this interface will allow you to enumerate its Keys and Values

collections as you could with a standard Dictionary<String, Object> object. This can be useful when your key value is specified in a string variable and thus you cannot specify the member name at compile-time.

ExpandoObject also implements INotifyPropertyChanging, raising a PropertyChanging event whenever a member is modified. This allows ExpandoObject to work well with WPF data-binding and other environments that need to know when the contents of the ExpandoObject change.

ExpandoObject is a useful library class when you need a reliable, plain-vanilla dynamic object, as an end-user or even as a library author, but what if you want more power to define your own types with their own dynamic dispatch semantics? This is where DynamicObject comes in.

3 DynamicObject

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.

DynamicObject provides a set of 12 virtual methods, each representing a possible dynamic operation 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)
```

We refer to each place in your code where a dynamic operation occurs as a dynamic call site. Each dynamic call site contains a cache of how to perform a given operation at that location in the code, given the particular types of objects that have been seen previously at that site. Each site has attached to it a runtime binder from the language in which the code is written. The binder encodes the static information about the operation at this call site. The binder then uses this static information and the arguments at runtime to figure out how to perform the specified operation for the given argument types. By convention, however, language binders defer binding to dynamic objects first, before doing the binding on their own. This follows the DLR principle that "the object is king", and allows objects from dynamic languages to bind under the semantics of the language that defined them.

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.

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.

3.1 DynamicBag: Implementing Our Own ExpandoObject

As a simple example of using DynamicObject, let's implement DynamicBag, our own version of ExpandoObject. DynamicBag derives from DynamicObject and overrides the TryGetMember and TrySetMember methods:

```
public class DynamicBag : DynamicObject {
    Dictionary<string, object> items
        = new Dictionary<string, object>();

    public override bool TryGetMember(
        GetMemberBinder binder, out object result) {
        return items.TryGetValue(binder.Name, out result);
    }

    public override bool TrySetMember(
```

```
            SetMemberBinder binder, object value) {
            items[binder.Name] = value;
            return true;
        }
    }
```

We first set up a Dictionary that maps string values to object values, and then implement TryGetMember and TrySetMember to manipulate entries in this Dictionary. The first parameter of each override is the DLR call site binder provided by the dynamic call site that began this operation. In this case, the binder has stashed at compile time the name of the member being retrieved or being set.

Further parameters serve two purposes. Some are the runtime arguments to the operation, such as the value parameter on TrySetMember. Some are the out parameters through which the operation's value is returned, such as the result parameter on TryGetMember. In the case of TryGetMember, we call the TryGetValue method on the items Dictionary and return its false return value if the item can't be found. By returning false instead of just throwing an exception, we're telling the DLR to fall back to the specific error semantics of the calling language (which might be returning a sentinel value like JS's $Undefined).

Because the DynamicObject base class implements IDynamicMetaObjectProvider, this class is now ready to act as an Expando object in C#, Visual Basic, Python, Ruby or any other language that can dynamically consume DLR objects:

```
static void Main(string[] args) {
    dynamic student, teacher;

    student = new DynamicBag();
    student.Name = "Billy";
    student.Age = 12;

    teacher = new DynamicBag();
    teacher.Name = "Ms. Anderson";
    teacher.Age = "thirty";

    WritePerson(student);
    WritePerson(teacher);
}

private static void WritePerson(dynamic person) {
    Console.WriteLine("{0} is {1} years old.",
                      person.Name, person.Age);
}
```

3.2 NamedBag: Optimizing DynamicObject with Statically Defined Members

We can do some simple optimizations to tune our object. Let's say we want to allow dynamic members, but we know the object will always have a Name member. We can simply include these statically defined members on our object. The DLR calls on the language to look for these names first before calling TryGetMember:

```
public class NamedBag : DynamicObject {
    Dictionary<string, object> items
        = new Dictionary<string, object>();
```

```
        public string Name;

        public string LowercaseName {
            get { return Name.ToLower(); }
        }

        public override bool TryGetMember(GetMemberBinder binder,
                                          out object result) {
            // Don't need to test for "Name" or "LowercaseName".
            // Will never get called with those names.
            result = items[binder.Name];
            return true;
        }

        public override bool TrySetMember(SetMemberBinder binder,
                                          object value) {
            //if (binder.Name == "LowercaseName") return false;
            items[binder.Name] = value;
            return true;
        }
    }
```

This NamedBag now stores the value of its Name member in a real string field, making access to this field faster. We've also added a property LowercaseName which gives us the name converted to lowercase. The DynamicObject's DynamicMetaObject first calls on the binder to get a language-specific expression tree that captures any access to statically defined members. The meta object then causes that expression to be wrapped so that if the static binding fails, the expression then calls TryGetMember or TrySetMember. Finally if that fails too, the meta object's expression includes a language-specific error (or sentinel value like JS's $Undefined).

In TrySetMember you can decide whether to protect against dynamically setting the read-only property. A language may not roll over to the try to set the name dynamically when it sees the read-only property. However, by convention the language may still allow TrySetMember to execute. If it set the name to a value in the dictionary, there's no harm done, but the programmer will never be able to retrieve that value. The statically defined LowercaseName property will always dominate the binding for reading the value. Testing for the name and returning false would cause the language to throw an exception about setting a read-only property.

You've seen how you can easily start handling dynamic operations by deriving from DynamicObject and then implementing just the operations you care about. Most often, this should be flexible enough and fast enough for your needs as a library author, although it's still not as optimized as it could be. For example, lookups that we don't fast-path to fields still require a hashtable lookup, which is not nearly as efficient. If we know there are a few member names we'll be accessing most often, but don't know their names at compile time, we may wish to optimize our bag for that case.

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, and 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.

To take advantage of the DLR caching system and get the flexibility to tweak our performance, we'll look at how to get closer to the metal of the DLR and directly implement IDynamicMetaObjectProvider (and DynamicMetaObject) ourselves.

4 IDynamicMetaObjectProvider and DynamicMetaObject

IDynamicMetaObjectProvider and **DynamicMetaObject** are the core of the DLR's interoperability protocol. This is the level at which languages plug in for maximum power. For example, IronPython's PythonCallableObject or IronRuby's RubyMutableString implementation objects implement IDynamicMetaObjectProvider and produce their own DynamicMetaObjects for these runtime objects. Their DynamicMetaObjects have full control over how they participate in the interoperability protocol and the DLR's fast dynamic dispatch caching.

During binding of dynamic operations, the DLR checks if a target object implements IDynamicMetaObjectProvider, which has a single method GetMetaObject. If the target object does implement IDynamicMetaObjectProvider, the DLR uses its DynamicMetaObject instead of making a default reflection-based DynamicMetaObject. Binders call on DynamicMetaObjects by convention to give them first crack at binding operations, and an IDynamicMetaObjectProvider's custom DynamicMetaObject can offer its own binding in the form of an expression tree that represents the semantics desired for that requested operation. The DLR combines the expression tree with others and compiles them into a highly-optimized cache using the DLR's fast dynamic dispatch mechanism.

In the case of DynamicObject, this detail is abstracted away. You simply override methods for the dynamic operations in which your dynamic object should participate. The DLR automatically creates a DynamicMetaObject for your DynamicObject and returns expression trees to the dynamic caching that simply call your GetMember method. You do get some of the benefit of caching in that the call to your GetMember function is extremely fast. However you get far more efficiency if you implement IDynamicMetaObjectProvider yourself, where the cache can store an Expression Tree that performs the behavior for that specific name.

4.1 FastNBag: Faster Bags for N Slots

Let's say that we know that the first set of values added to our bag class will be accessed most often, with later additions accessed less frequently. In this case, we can create a custom bag class, FastNBag, which optimizes for that situation by storing the first N members in a fast array. The members added after the first N, go in a dictionary. Our bag can then generate optimized rules for the DLR call site caches by returning expression trees that directly access these array elements for the first N fast members.

After initializing the fast array and the hashtable, the bulk of the logic is then our DynamicMetaObject, MetaFastNBag. Its BindGetMember and BindSetMember methods effectively return expressions that manage adding and fetching the members appropriately.

Any access to the first N members is essentially a type check, field access, and index operation at a known index.

You can find the full source for FastNBag in the appendix.

4.1.1 BindSetMember Method

BindSetMember is the simpler of the two to understand. We're less concerned with optimizing member assignments than member fetches, so the expression trees remain simpler. They just call our helper method, SetValue:

```
public override DynamicMetaObject BindSetMember(
    SetMemberBinder binder, DynamicMetaObject value)
{
    var self = this.Expression;

    var keyExpr = Expression.Constant(binder.Name);
    var valueExpr = Expression.Convert(
                        value.Expression,
                        typeof(object)
                    );

    var target =
        Expression.Call(
            Expression.Convert(self, typeof(FastNBag)),
            typeof(FastNBag).GetMethod("SetValue"),
            keyExpr,
            valueExpr
        );

    var restrictions = BindingRestrictions
                           .GetTypeRestriction(self, typeof(FastNBag));

    return new DynamicMetaObject(target, restrictions);
}
```

The goal of the method is to produce a DynamicMetaObject that represents the result of our binding. When building a DynamicMetaObject, the most important components are the target and the restrictions:

- A DynamicMetaObject's **target** is the expression tree that represents the desired semantics we'd like for a given operation. In the case of BindSetMember, the tree we generate just calls our general SetValue method in all cases, passing it the binder's member name as the key and the value object as the value. Our SetValue method decides whether to assign the new entry to the fast array or to the hashtable, and increments a version counter to indicate that our object has changed. This target expression will ultimately be stored in the call site's cache. When this member is assigned again, the site hits the cache, avoiding another call to BindSetMember and directly calling SetValue. It is often easier to create methods like SetValue that implement the functionality you want and call that from the expression tree, rather than implement all of the logic in the expression tree itself.
- A DynamicMetaObject's **restrictions** let us constrain when the target expression we've specified will be applicable again in the future. A given SetMemberBinder encapsulates

a specific member name, so we need to decide which other restrictions will be necessary. In this case, by calling BindingRestrictions.GetTypeRestriction, we're creating a **type restriction**, constraining this target to apply in all future cases where the bag object has the exact type of FastNBag. This way, if we were to define a subclass of FastNBag with a different implementation, the two types would have distinct cache entries. However, the same cache rule works for multiple instances of one of the types.

4.1.2 BindGetMember Method

Our more involved caching logic is present inside BindGetMember, where we want accesses to the first N values to go directly to the array, storing the array index inside the target expression as a constant:

```
public override DynamicMetaObject BindGetMember
        (GetMemberBinder binder)
{
    var self = this.Expression;
    var bag = (FastNBag)base.Value;

    int index = bag.GetFastIndex(binder.Name);

    Expression target;

    // If match found in fast array:
    if (index != -1)
    {
        // Fetch result from fast array.
        target =
            Expression.Call(
                Expression.Convert(self, typeof(FastNBag)),
                typeof(FastNBag).GetMethod("GetFastValue"),
                Expression.Constant(index)
            );
    }
    // Else, if no match found in fast array, but fast array is full:
    else if (bag.fastTable.Count == bag.fastCount)
    {
        // Fetch result from dictionary.
        var keyExpr = Expression.Constant(binder.Name);
        var valueExpr = Expression.Variable(typeof(object));

        var dictCheckExpr =
            Expression.Call(
                Expression.Convert(self, typeof(FastNBag)),
                typeof(FastNBag).GetMethod("TryGetValue"),
                keyExpr,
                valueExpr
            );
        var dictFailExpr =
            Expression.Block(
                binder.FallbackGetMember(this).Expression,
                Expression.Default(typeof(object))
            );

        target =
```

```
            Expression.Block(
                new [] { valueExpr },
                Expression.Condition(
                    dictCheckExpr,
                    valueExpr,
                    dictFailExpr
                )
            );
    }
    // Else, no match found in fast array, fast array is not yet full:
    else
    {
        // Fail binding, but only until fast array is updated.
        var versionCheckExpr =
            Expression.Call(
                Expression.Convert(self, typeof(FastNBag)),
                typeof(FastNBag).GetMethod("CheckVersion"),
                Expression.Constant(bag.Version)
            );
        var versionMatchExpr =
            binder.FallbackGetMember(this).Expression;
        var updateExpr =
            binder.GetUpdateExpression(versionMatchExpr.Type);

        target =
            Expression.Condition(
                versionCheckExpr,
                versionMatchExpr,
                updateExpr
            );
    }

    var restrictions = BindingRestrictions
                           .GetInstanceRestriction(self, bag);

    return new DynamicMetaObject(target, restrictions);
}
```

The BindGetMember implementation creates a target and restrictions as did BindSetMember, but creates one of three different target expression trees. The correct binding logic depends on both the member name and the state of the bag in terms of whether we've seen N names yet:

- We first look up the member name to see if it already exists as an entry in the bag's fast array. If so, the target expression is a call to the GetFastValue method, passing in the specific array index to fetch. As entries will never move around inside the array, this rule will be applicable for this bag forever. Note, we're doing the extra look up work at bind time, but from then on, the fetch is an 'if' test, field access, and array access.

- If the name is not in the fast array, and the fast array is full, this means that the name could only ever appear in the hashtable from now on. In this case, the target expression emitted is a call to the slower TryGetValue method, which checks the hashtable for the member name and returns its value through an out parameter.

  We do not check at bind time whether the member name exists in the hashtable. At runtime, TryGetValue handles missing keys by returning false. That causes execution to

flow to the the dictFailExpr branch of the Condition expression. That branch was generated by calling FallbackGetMember to fetch a binding expression from the language's binder. This Fallback call lets the language's own error semantics surface, such as a language-specific exception or even a special sentinel value like JScript's "$Undefined".

An alternative to FallbackGetMember would be to return a Throw expression tree to throw an exception when member lookup failures or to just throw an exception inside the TryGetValue method. The downside of this approach is that it does not respect the source language's error semantics in the first case. In the second case, it's important to avoid actually throwing the exceptions during binding itself as this breaks the binding machinery, which expects to always complete with with an expression. Representing exceptions and fallbacks as expression trees instead of throwing allows these failed bindings to be cached as well.

- If the key is not found in the fast array, and there are still free slots in the array, we have a trickier situation. We know the hashtable must be empty, and so we want the target to throw an exception for now, but this may not be the case forever. As more entries are added to the bag, this rule could become invalid.

To handle the situation where we've set fewer than N names, and we're trying to get a value for a name we haven't seen before, the expression tree changes its behavior based on a version test. Each time the bag sets new member, it increments an internal version number which can be used to tell if the bag has remained the same since a rule was created. In this case, we only want our throw expression to be applicable in the future if the bag is still the same version. We therefore encode a call to our CheckVersion method into the target, hard-coding the current version number. At runtime, if this rule is chosen, it will check that the version number is the same, then throw an exception since we know the member doesn't exist in that version.

Quick note, we use a distinct version counter in this example for explanation purposes, but you could just write this code using fasttable.Count as the version tick. You only really need to increment the version when you add a new member to the fast table since setting existing members doesn't change the binding logic. Setting existing members would only be interesting to version if, say, you chose to hard code the member's value itself in the rule's target (instead of an array lookup expression).

When the actual version is not the same as the version we "hard coded" into the target expression, we need this target expression to give up. It needs to defer control back to the bag's DynamicMetaObject to bind the operation again. We do this with a special expression supplied by the binder base class, the Update expression. By inserting the expression returned from DynamicMetaObjectBinder.GetUpdateExpression into a branch of our target expression, we can cause the dynamic call site to rebind and update its cache when the version number does not match. To update, the DLR continue trying cached rules that may apply. If no match is found, the DLR rebinds the CallSite's operation by calling on the binder. This lets the binder and DynamicMetaObject produce a rule based on the current state of the bag, including the bag's new version number.

Unlike BindSetMember where we always called SetValue, the rules generated by BindGetMember are specific not just to the bag's type. BindGetMember's rules use restrictions that match specific instances of the bag since the N entries in the fast array may differ between

bags. Therefore, the restriction for the applicability of BindGetMember's target expression is an **instance restriction**, generated by the helper method BindingRestrictions.GetInstanceRestriction.

Because our FastNBag implements IDynamicMetaObjectProvider, it's usable in any context where DynamicBag, NamedBag or ExpandoObject were valid before. By tuning our bags using the various mechanisms available to us, we now have bags with a range of performance characteristics. FastNBag should be faster than both DynamicBag and NamedBag as it benefits from using lower-level controls of DLR caching. FastNBag may turn out to be slower than NamedBag if the Name member is not among the first N members while, but is accessed quite often. ExpandoObject should be faster than all of these bags since it has been heavily optimized and uses an even more sophisticated scheme than our FastNBag to ensure quick, cached access for all its members in many different situations.

4.1.3 GetDynamicMemberNames Method

To aid in debugging programs that use FastNBag, we can implement the GetDynamicMemberNames method on MetaFastNBag.

By returning a sequence of member names as strings from this method, we can let consumers of our object know at runtime which member names we can currently bind. This allows an IDE populate tooltips or other IDE features with information about the members of a dynamic object. For example, Visual Studio shows a "Dynamic View" node in DataTips and in the Locals/Watch windows that expand to list out the dynamic members available on an object.

To provide this member list, we override GetDynamicMemberNames in the MetaFastNBag class:

```
public override IEnumerable<string> GetDynamicMemberNames()
{
    var bag = (FastNBag)base.Value;

    return bag.GetKeys();
}
```

Our implementation calls into a GetKeys method we define in FastNBag:

```
public IEnumerable<string> GetKeys()
{
    var fastKeys = fastTable.Keys;
    var hashKeys = hashTable.Keys;

    var keys = fastKeys.Concat(hashKeys);

    return keys;
}
```

When an IDE needs to determine the bindable members of a FastNBag instance, it can call IDynamicMetaObjectProvider.GetDynamicMetaObject for the instance and then call GetDynamicMemberNames, which will return the set of keys in both the fast array and the hashtable.

4.2 Further Reading

If you'd like to learn more about IDynamicMetaObjectProvider and DynamicMetaObject, check out the accompanying "Sites, Binders, and Dynamic Object Interop" spec, which covers the underlying mechanisms of the DLR at a deeper level.

5 Appendix

5.1 DynamicObject Virtual Methods

The 16 virtual DynamicObject methods you may override and the dynamic operations they encode are listed below:

Virtual method	Encodes dynamic operation
TryGetMember	Represents an access to an object's member that retrieves the value. *Example:* o.m
TrySetMember	Represents an access to an object's member that assigns a value. *Example:* o.m = 12
TryDeleteMember	Represents an access to an object's member that deletes the member. *Example:* delete o.m
TryGetIndex	Represents an access to an indexed element of an object or an object's member that retrieves the value. The binder may choose to create the element if it doesn't exist or throw an exception. *Example:* o[0] *or* o.m[0]
TrySetIndex	Represents an access to an indexed element of an object or an object's member that assigns a value. *Example:* o[0] = 12 *or* o.m[0] = 12
TryDeleteIndex	Represents an access to an indexed element of an object or an object's member that deletes the element. *Example:* delete o[0] *or* delete o.m[0]
TryInvoke	Represents invocation of an invocable object, such as a delegate or first-class function object, with a set of positional/named arguments. *Example:* a(3) When binding an expression like a.b(3) in a language with first-class function objects, such as Python, the intermediate object a.b will be requested first with a call to GetMember, and then the invocable object you return will be invoked using Invoke.

TryInvokeMember	Represents invocation of an invocable member on an object, such as a method, with a set of positional/named arguments. *Example:* a.b(3) InvokeMember is used instead of GetMember + Invoke in languages where invoking a member on an object is an atomic operation. For example, in C#, a.b() may refer to a method group b that represents multiple overloads and would have no intermediate object representation for GetMember implementation to return. A DynamicObject which does offer a first-class GetMember and Invoke may compose these methods to trivially implement InvokeMember.
TryCreateInstance	Represents an object instantiation with a set of positional/named constructor arguments. *Example:* new X(3, 4, 5)
TryConvert	Represents a conversion of an expression to a target type. This conversion may be marked in the ConvertBinder parameter as being an implicit compiler-inferred conversion, or an explicit conversion specified by the developer. *Example:* (TargetType)o
TryUnaryOperation **TryBinaryOperation**	Represents a miscellaneous unary or binary operation, respectively, such as unary minus, or addition. Contains an Operation string that specifies the operation to perform, such as Add, Subtract, Negate, etc.. There is a core set of operations defined that all language binders should support if they map reasonably to concepts in the language. Languages may also define their own Operation strings for features unique to their language, and may agree independently to share these strings to enable interop for these features. *Examples:* -a, a + b, a * b

5.2 FastNBag Full Source

Below is the full source code for the FastNBag implementation discussed above.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Linq;
using System.Dynamic;
using System.Linq.Expressions;
using System.ComponentModel;

namespace Bags
{
    public class FastNBag : IDynamicMetaObjectProvider, INotifyPropertyChanged
    {
```

```
private object[] fastArray;
private Dictionary<string, int> fastTable;

private Dictionary<string, object> hashTable
    = new Dictionary<string, object>();

private readonly int fastCount;

public int Version { get; set; }

public event PropertyChangedEventHandler PropertyChanged;


public FastNBag(int fastCount)
{
    this.fastCount = fastCount;
    this.fastArray = new object[fastCount];
    this.fastTable = new Dictionary<string, int>(fastCount);
}

public bool TryGetValue(string key, out object value)
{
    int index = GetFastIndex(key);
    if (index != -1)
    {
        value = GetFastValue(index);
        return true;
    }
    else if (fastTable.Count == fastCount)
    {
        return hashTable.TryGetValue(key, out value);
    }
    else
    {
        value = null;
        return false;
    }
}

public void SetValue(string key, object value)
{
    int index = GetFastIndex(key);
    if (index != -1)
        SetFastValue(index, value);
    else
        if (fastTable.Count < fastCount)
        {
            index = fastTable.Count;
            fastTable[key] = index;
            SetFastValue(index, value);
        }
        else
            hashTable[key] = value;

    Version++;

    if (PropertyChanged != null)
    {
        PropertyChanged(this, new PropertyChangedEventArgs(key));
    }
}

public object GetFastValue(int index)
{
    return fastArray[index];
}

public void SetFastValue(int index, object value)
{
    fastArray[index] = value;
}
```

```
public int GetFastIndex(string key)
{
    int index;
    if (fastTable.TryGetValue(key, out index))
        return index;
    else
        return -1;
}

public IEnumerable<string> GetKeys()
{
    var fastKeys = fastTable.Keys;
    var hashKeys = hashTable.Keys;

    var keys = fastKeys.Concat(hashKeys);

    return keys;
}

public bool CheckVersion(int ruleVersion)
{
    return (Version == ruleVersion);
}

public DynamicMetaObject GetMetaObject(Expression parameter)
{
    return new MetaFastNBag(parameter, this);
}



private class MetaFastNBag : DynamicMetaObject
{
    public MetaFastNBag(Expression expression, FastNBag value)
        : base(expression, BindingRestrictions.Empty, value) { }

    public override DynamicMetaObject BindGetMember
            (GetMemberBinder binder)
    {
        var self = this.Expression;
        var bag = (FastNBag)base.Value;

        int index = bag.GetFastIndex(binder.Name);

        Expression target;

        // If match found in fast array:
        if (index != -1)
        {
            // Fetch result from fast array.
            target =
                Expression.Call(
                    Expression.Convert(self, typeof(FastNBag)),
                    typeof(FastNBag).GetMethod("GetFastValue"),
                    Expression.Constant(index)
                );
        }
        // Else, if no match found in fast array, but fast array is full:
        else if (bag.fastTable.Count == bag.fastCount)
        {
            // Fetch result from dictionary.
            var keyExpr = Expression.Constant(binder.Name);
            var valueExpr = Expression.Variable(typeof(object));

            var dictCheckExpr =
                Expression.Call(
                    Expression.Convert(self, typeof(FastNBag)),
                    typeof(FastNBag).GetMethod("TryGetValue"),
                    keyExpr,
                    valueExpr
```

```
            );
        var dictFailExpr =
            Expression.Block(
                binder.FallbackGetMember(this).Expression,
                Expression.Default(typeof(object))
            );

        target =
            Expression.Block(
                new [] { valueExpr },
                Expression.Condition(
                    dictCheckExpr,
                    valueExpr,
                    dictFailExpr
                )
            );
    }
    // Else, no match found in fast array, fast array is not yet full:
    else
    {
        // Fail binding, but only until fast array is updated.
        var versionCheckExpr =
            Expression.Call(
                Expression.Convert(self, typeof(FastNBag)),
                typeof(FastNBag).GetMethod("CheckVersion"),
                Expression.Constant(bag.Version)
            );
        var versionMatchExpr =
            binder.FallbackGetMember(this).Expression;
        var updateExpr =
            binder.GetUpdateExpression(versionMatchExpr.Type);

        target =
            Expression.Condition(
                versionCheckExpr,
                versionMatchExpr,
                updateExpr
            );
    }

    var restrictions = BindingRestrictions
                           .GetInstanceRestriction(self, bag);

    return new DynamicMetaObject(target, restrictions);
}


public override DynamicMetaObject BindSetMember(
    SetMemberBinder binder, DynamicMetaObject value)
{
    var self = this.Expression;

    var keyExpr = Expression.Constant(binder.Name);
    var valueExpr = Expression.Convert(
                        value.Expression,
                        typeof(object)
                    );

    var target =
        Expression.Call(
            Expression.Convert(self, typeof(FastNBag)),
            typeof(FastNBag).GetMethod("SetValue"),
            keyExpr,
            valueExpr
        );

    var restrictions = BindingRestrictions
                           .GetTypeRestriction(self, typeof(FastNBag));

    return new DynamicMetaObject(target, restrictions);
}
```

```
            public override IEnumerable<string> GetDynamicMemberNames()
            {
                var bag = (FastNBag)base.Value;

                return bag.GetKeys();
            }
        }
    }
}
```