Dynamic Programming in the .NET Framework

.NET Framework (current version)

This section of the documentation provides information about dynamic programming in the .NET Framework.

In This Section

Reflection in the .NET Framework
    Describes how to use reflection to work with objects at run time.

Emitting Dynamic Methods and Assemblies
    Describes how to create methods and assemblies at run time by using Reflection.Emit.

Dynamic Language Runtime Overview
    Describes the features of the dynamic language runtime.

Dynamic Source Code Generation and Compilation
    Describes how to generate and compile dynamic source code.

Related Sections

.NET Framework Development Guide

Advanced Reading for the .NET Framework

Reflection in the .NET Framework

.NET Framework (current version)

The classes in the System.Reflection namespace, together with System.Type, enable you to obtain information about loaded assemblies and the types defined within them, such as classes, interfaces, and value types. You can also use reflection to create type instances at run time, and to invoke and access them. For topics about specific aspects of reflection, see Related Topics at the end of this overview.

The common language runtime loader manages application domains, which constitute defined boundaries around objects that have the same application scope. This management includes loading each assembly into the appropriate application domain and controlling the memory layout of the type hierarchy within each assembly.

Assemblies contain modules, modules contain types, and types contain members. Reflection provides objects that encapsulate assemblies, modules, and types. You can use reflection to dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object. You can then invoke the type's methods or access its fields and properties. Typical uses of reflection include the following:

- Use Assembly to define and load assemblies, load modules that are listed in the assembly manifest, and locate a type from this assembly and create an instance of it.
- Use Module to discover information such as the assembly that contains the module and the classes in the module. You can also get all global methods or other specific, nonglobal methods defined on the module.
- Use ConstructorInfo to discover information such as the name, parameters, access modifiers (such as **public** or **private**), and implementation details (such as **abstract** or **virtual**) of a constructor. Use the GetConstructors or GetConstructor method of a Type to invoke a specific constructor.
- Use MethodInfo to discover information such as the name, return type, parameters, access modifiers (such as **public** or **private**), and implementation details (such as **abstract** or **virtual**) of a method. Use the GetMethods or GetMethod method of a Type to invoke a specific method.
- Use FieldInfo to discover information such as the name, access modifiers (such as **public** or **private**) and implementation details (such as **static**) of a field, and to get or set field values.
- Use EventInfo to discover information such as the name, event-handler data type, custom attributes, declaring type, and reflected type of an event, and to add or remove event handlers.
- Use PropertyInfo to discover information such as the name, data type, declaring type, reflected type, and read-only or writable status of a property, and to get or set property values.
- Use ParameterInfo to discover information such as a parameter's name, data type, whether a parameter is an input or output parameter, and the position of the parameter in a method signature.
- Use CustomAttributeData to discover information about custom attributes when you are working in the reflection-only context of an application domain. CustomAttributeData allows you to examine attributes without creating instances of them.

The classes of the System.Reflection.Emit namespace provide a specialized form of reflection that enables you to build types at run time.

Reflection can also be used to create applications called type browsers, which enable users to select types and then view the information about those types.

There are other uses for reflection. Compilers for languages such as JScript use reflection to construct symbol tables. The classes in the System.Runtime.Serialization namespace use reflection to access data and to determine which fields to persist. The classes in the System.Runtime.Remoting namespace use reflection indirectly through serialization.

Runtime Types in Reflection

Reflection provides classes, such as Type and MethodInfo, to represent types, members, parameters, and other code entities. However, when you use reflection you don't work directly with these classes, most of which are abstract (**MustInherit** in Visual Basic). Instead, you work with types provided by the common language runtime (CLR).

For example, when you use the C# **typeof** operator (**GetType** in Visual Basic) to obtain a Type object, the object is really a **RuntimeType**. **RuntimeType** derives from Type, and provides implementations of all the abstract methods.

These runtime classes are **internal** (**Friend** in Visual Basic). They are not documented separately from their base classes, because their behavior is described by the base class documentation.

Related Topics

Title	**Description**
Viewing Type Information	Describes the Type class and provides code examples that illustrate how to use Type with several reflection classes to obtain information about constructors, methods, fields, properties, and events.
Reflection and Generic Types	Explains how reflection handles the type parameters and type arguments of generic types and generic methods.
Security Considerations for Reflection	Describes the rules that determine to what degree reflection can be used to discover type information and access types.
Dynamically Loading and Using Types	Describes the reflection custom-binding interface that supports late binding.
How to: Load Assemblies into the Reflection-Only Context	Describes the reflection-only load context. Shows how to load an assembly, how to test the context, and how to examine attributes applied to an assembly in the reflection-only context.
Accessing Custom Attributes	Demonstrates using reflection to query attribute existence and values.
Specifying Fully Qualified Type Names	Describes the format of fully qualified type names in terms of the Backus-Naur form (BNF), and the syntax required for specifying special characters, assembly names, pointers, references, and arrays.

How to: Hook Up a Delegate Using Reflection	Explains how to create a delegate for a method and hook the delegate up to an event. Explains how to create an event-handling method at run time using DynamicMethod.
Emitting Dynamic Methods and Assemblies	Explains how to generate dynamic assemblies and dynamic methods.

Reference

System.Type

System.Reflection

System.Reflection.Emit

Back to Top

Viewing Type Information

.NET Framework (current version)

The System.Type class is central to reflection. The common language runtime creates the **Type** for a loaded type when reflection requests it. You can use a **Type** object's methods, fields, properties, and nested classes to find out everything about that type.

Use Assembly.GetType or Assembly.GetTypes to obtain **Type** objects from assemblies that have not been loaded, passing in the name of the type or types you want. Use Type.GetType to get the **Type** objects from an assembly that is already loaded. Use Module.GetType and Module.GetTypes to obtain module **Type** objects.

Note

If you want to examine and manipulate generic types and methods, please see the additional information provided in Reflection and Generic Types and How to: Examine and Instantiate Generic Types with Reflection.

The following example shows the syntax necessary to get the Assembly object and module for an assembly.

VB

```
' Gets the mscorlib assembly in which the object is defined.
Dim a As Assembly = GetType(Object).Module.Assembly
```

The following example demonstrates getting **Type** objects from a loaded assembly.

VB

```
' Loads an assembly using its file name.
Dim a As Assembly = Assembly.LoadFrom("MyExe.exe")
' Gets the type names from the assembly.
Dim types2() As Type = a.GetTypes()
For Each t As Type In types2
    Console.WriteLine(t.FullName)
Next t
```

Once you obtain a **Type**, there are many ways you can discover information about the members of that type. For example, you can find out about all the type's members by calling the Type.GetMembers method, which obtains an array of MemberInfo objects describing each of the members of the current type.

You can also use methods on the **Type** class to retrieve information about one or more constructors, methods, events, fields, or properties that you specify by name. For example, Type.GetConstructor encapsulates a specific constructor of the current class.

If you have a **Type**, you can use the Type.Module property to obtain an object that encapsulates the module containing that type. Use the Module.Assembly property to locate an object that encapsulates the assembly containing the module. You can obtain the assembly that encapsulates the type directly by using the Type.Assembly property.

System.Type and ConstructorInfo

The following example shows how to list the constructors for a class, in this case, the String class.

VB

```
' This program lists all the public constructors
' of the System.String class.

Imports System
Imports System.Reflection

Class ListMembers
    Public Shared Sub Main()
        Dim t As Type = GetType(String)
        Console.WriteLine("Listing all the public constructors of the {0} type", t)
        ' Constructors.
        Dim ci As ConstructorInfo() = t.GetConstructors((BindingFlags.Public Or
BindingFlags.Instance))
        Console.WriteLine("//Constructors")
        PrintMembers(ci)
    End Sub
    Public Shared Sub PrintMembers(ms() As MemberInfo)
        Dim m As MemberInfo
        For Each m In ms
            Console.WriteLine("{0}{1}", "     ", m)
        Next m
        Console.WriteLine()
    End Sub
End Class
```

MemberInfo, MethodInfo, FieldInfo, and PropertyInfo

Obtain information about the type's methods, properties, events, and fields using MemberInfo, MethodInfo, FieldInfo, or PropertyInfo objects.

The following example uses **MemberInfo** to list the number of members in the **System.IO.File** class and uses the System.Type.IsPublic property to determine the visibility of the class.

VB

```
Imports System
Imports System.IO
Imports System.Reflection

Class Mymemberinfo
    Public Shared Sub Main()
```

```
        Console.WriteLine ("\nReflection.MemberInfo")
        ' Gets the Type and MemberInfo.
        Dim MyType As Type = Type.GetType("System.IO.File")
        Dim Mymemberinfoarray() As MemberInfo = MyType.GetMembers()
        ' Gets and displays the DeclaringType method.
        Console.WriteLine("\nThere are {0} members in {1}.",
            Mymemberinfoarray.Length, MyType.FullName)
        Console.WriteLine("{0}.", MyType.FullName)
        If MyType.IsPublic
            Console.WriteLine("{0} is public.", MyType.FullName)
        End If
    End Sub
End Class
```

The following example investigates the type of the specified member. It performs reflection on a member of the **MemberInfo** class, and lists its type.

VB

```
' This code displays information about the GetValue method of FieldInfo.
Imports System
Imports System.Reflection
Class MyMethodInfo
    Public Shared Sub Main()
        Console.WriteLine("Reflection.MethodInfo")
        ' Gets and displays the Type.
        Dim MyType As Type = Type.GetType("System.Reflection.FieldInfo")
        ' Specifies the member for which you want type information.
        Dim Mymethodinfo As MethodInfo = MyType.GetMethod("GetValue")
        Console.WriteLine((MyType.FullName & "." & Mymethodinfo.Name))
        ' Gets and displays the MemberType property.
        Dim Mymembertypes As MemberTypes = Mymethodinfo.MemberType
        If MemberTypes.Constructor = Mymembertypes Then
            Console.WriteLine("MemberType is of type All")
        ElseIf MemberTypes.Custom = Mymembertypes Then
            Console.WriteLine("MemberType is of type Custom")
        ElseIf MemberTypes.Event = Mymembertypes Then
            Console.WriteLine("MemberType is of type Event")
        ElseIf MemberTypes.Field = Mymembertypes Then
            Console.WriteLine("MemberType is of type Field")
        ElseIf MemberTypes.Method = Mymembertypes Then
            Console.WriteLine("MemberType is of type Method")
        ElseIf MemberTypes.Property = Mymembertypes Then
            Console.WriteLine("MemberType is of type Property")
        ElseIf MemberTypes.TypeInfo = Mymembertypes Then
            Console.WriteLine("MemberType is of type TypeInfo")
        End If
        Return
    End Sub
End Class
```

The following example uses all the Reflection ***Info** classes along with BindingFlags to list all the members (constructors, fields, properties, events, and methods) of the specified class, dividing the members into static and instance categories.

VB

```
' This program lists all the members of the
' System.IO.BufferedStream class.
Imports System
Imports System.IO
Imports System.Reflection

Class ListMembers
    Public Shared Sub Main()
        ' Specifies the class.
        Dim t As Type = GetType(System.IO.BufferedStream)
        Console.WriteLine("Listing all the members (public and non public) of the {0}
type", t)
        ' Lists static fields first.
        Dim fi As FieldInfo() = t.GetFields((BindingFlags.Static Or
BindingFlags.NonPublic Or BindingFlags.Public))
        Console.WriteLine("// Static Fields")
        PrintMembers(fi)
        ' Static properties.
        Dim pi As PropertyInfo() = t.GetProperties((BindingFlags.Static Or
BindingFlags.NonPublic Or BindingFlags.Public))
        Console.WriteLine("// Static Properties")
        PrintMembers(pi)
        ' Static events.
        Dim ei As EventInfo() = t.GetEvents((BindingFlags.Static Or
BindingFlags.NonPublic Or BindingFlags.Public))
        Console.WriteLine("// Static Events")
        PrintMembers(ei)
        ' Static methods.
        Dim mi As MethodInfo() = t.GetMethods((BindingFlags.Static Or
BindingFlags.NonPublic Or BindingFlags.Public))
        Console.WriteLine("// Static Methods")
        PrintMembers(mi)
        ' Constructors.
        Dim ci As ConstructorInfo() = t.GetConstructors((BindingFlags.Instance Or
BindingFlags.NonPublic Or BindingFlags.Public))
        Console.WriteLine("// Constructors")
        PrintMembers(ci)
        ' Instance fields.
        fi = t.GetFields((BindingFlags.Instance Or BindingFlags.NonPublic Or
BindingFlags.Public))
        Console.WriteLine("// Instance Fields")
        PrintMembers(fi)
        ' Instance properites.
        pi = t.GetProperties((BindingFlags.Instance Or BindingFlags.NonPublic Or
BindingFlags.Public))
        Console.WriteLine("// Instance Properties")
        PrintMembers(pi)
        ' Instance events.
        ei = t.GetEvents((BindingFlags.Instance Or BindingFlags.NonPublic Or
BindingFlags.Public))
```

```
            Console.WriteLine("// Instance Events")
            PrintMembers(ei)
            ' Instance methods.
            mi = t.GetMethods((BindingFlags.Instance Or BindingFlags.NonPublic Or
    BindingFlags.Public))
            Console.WriteLine("// Instance Methods")
            PrintMembers(mi)
            Console.WriteLine(ControlChars.CrLf & "Press ENTER to exit.")
            Console.Read()
        End Sub

        Public Shared Sub PrintMembers(ms() As MemberInfo)
            Dim m As MemberInfo
            For Each m In  ms
                Console.WriteLine("{0}{1}", "     ", m)
            Next m
            Console.WriteLine()
        End Sub
    End Class
```

See Also

BindingFlags
Assembly.GetType
Assembly.GetTypes
Type.GetType
Type.GetMembers
Type.GetFields
Module.GetType
Module.GetTypes
MemberInfo
ConstructorInfo
MethodInfo
FieldInfo
EventInfo
ParameterInfo
Reflection and Generic Types

Reflection and Generic Types

.NET Framework (current version)

From the point of view of reflection, the difference between a generic type and an ordinary type is that a generic type has associated with it a set of type parameters (if it is a generic type definition) or type arguments (if it is a constructed type). A generic method differs from an ordinary method in the same way.

There are two keys to understanding how reflection handles generic types and methods:

- The type parameters of generic type definitions and generic method definitions are represented by instances of the Type class.

  Note

  Many properties and methods of Type have different behavior when a Type object represents a generic type parameter. These differences are documented in the property and method topics. For example, see IsAutoClass and DeclaringType. In addition, some members are valid only when a Type object represents a generic type parameter. For example, see GetGenericTypeDefinition.

- If an instance of Type represents a generic type, then it includes an array of types that represent the type parameters (for generic type definitions) or the type arguments (for constructed types). The same is true of an instance of the MethodInfo class that represents a generic method.

Reflection provides methods of Type and MethodInfo that allow you to access the array of type parameters, and to determine whether an instance of Type represents a type parameter or an actual type.

For example code demonstrating the methods discussed here, see How to: Examine and Instantiate Generic Types with Reflection.

The following discussion assumes familiarity with the terminology of generics, such as the difference between type parameters and arguments and open or closed constructed types. For more information, see Generics in the .NET Framework.

This overview consists of the following sections:

Is This a Generic Type or Method?

When you use reflection to examine an unknown type, represented by an instance of Type, use the IsGenericType property to determine whether the unknown type is generic. It returns **true** if the type is generic. Similarly, when you examine an unknown method, represented by an instance of the MethodInfo class, use the IsGenericMethod property to determine whether the method is generic.

Is This a Generic Type or Method Definition?

Use the IsGenericTypeDefinition property to determine whether a Type object represents a generic type definition, and use the IsGenericMethodDefinition method to determine whether a MethodInfo represents a generic method definition.

Generic type and method definitions are the templates from which instantiable types are created. Generic types in the .NET Framework class library, such as Dictionary(Of TKey, TValue), are generic type definitions.

Is the Type or Method Open or Closed?

A generic type or method is closed if instantiable types have been substituted for all its type parameters, including all the type parameters of all enclosing types. You can only create an instance of a generic type if it is closed. The Type.ContainsGenericParameters property returns **true** if a type is open. For methods, the MethodInfo.ContainsGenericParameters method performs the same function.

Back to top

Generating Closed Generic Types

Once you have a generic type or method definition, use the MakeGenericType method to create a closed generic type or the MakeGenericMethod method to create a MethodInfo for a closed generic method.

Getting the Generic Type or Method Definition

If you have an open generic type or method that is not a generic type or method definition, you cannot create instances of it and you cannot supply the type parameters that are missing. You must have a generic type or method definition. Use the GetGenericTypeDefinition method to obtain the generic type definition or the GetGenericMethodDefinition method to obtain the generic method definition.

For example, if you have a Type object representing `Dictionary<int, string>` (`Dictionary(Of Integer, String)` in Visual Basic) and you want to create the type `Dictionary<string, MyClass>`, you can use the GetGenericTypeDefinition method to get a Type representing `Dictionary<TKey, TValue>` and then use the MakeGenericType method to produce a Type representing `Dictionary<int, MyClass>`.

For an example of an open generic type that is not a generic type, see "Type Parameter or Type Argument" later in this topic.

Back to top

Examining Type Arguments and Type Parameters

Use the Type.GetGenericArguments method to obtain an array of Type objects that represent the type parameters or type arguments of a generic type, and use the MethodInfo.GetGenericArguments method to do the same for a generic method.

Once you know that a Type object represents a type parameter, there are many additional questions reflection can answer. You can determine the type parameter's source, its position, and its constraints.

Type Parameter or Type Argument

To determine whether a particular element of the array is a type parameter or a type argument, use the IsGenericParameter property. The IsGenericParameter property is **true** if the element is a type parameter.

A generic type can be open without being a generic type definition, in which case it has a mixture of type arguments and type parameters. For example, in the following code, class `D` derives from a type created by substituting the first type parameter of `D` for the second type parameter of `B`.

VB

```vb
Class B(Of T, U)
End Class
Class D(Of V, W)
    Inherits B(Of Integer, V)
End Class
```

If you obtain a Type object representing `D<V, W>` and use the BaseType property to obtain its base type, the resulting `type B<int, V>` is open, but it is not a generic type definition.

Source of a Generic Parameter

A generic type parameter might come from the type you are examining, from an enclosing type, or from a generic method. You can determine the source of the generic type parameter as follows:

- First, use the DeclaringMethod property to determine whether the type parameter comes from a generic method. If the property value is not a null reference (**Nothing** in Visual Basic), then the source is a generic method.
- If the source is not a generic method, use the DeclaringType property to determine the generic type the generic type parameter belongs to.

If the type parameter belongs to a generic method, the DeclaringType property returns the type that declared the generic method, which is irrelevant.

Position of a Generic Parameter

In rare situations, it is necessary to determine the position of a type parameter in the type parameter list of its declaring class. For example, suppose you have a Type object representing the `B<int, V>` type from the preceding example. The GetGenericArguments method gives you a list of type arguments, and when you examine `V` you can use the DeclaringMethod and DeclaringType properties to discover where it comes from. You can then use the GenericParameterPosition property to determine its position in the type parameter list where it was defined. In this example, `V` is at position 0 (zero) in the type parameter list where it was defined.

Base Type and Interface Constraints

Use the GetGenericParameterConstraints method to obtain the base type constraint and interface constraints of a type parameter. The order of the elements of the array is not significant. An element represents an interface constraint if it is an interface type.

Generic Parameter Attributes

The GenericParameterAttributes property gets a GenericParameterAttributes value that indicates the variance (covariance or contravariance) and the special constraints of a type parameter.

Covariance and Contravariance

To determine whether a type parameter is covariant or contravariant, apply the GenericParameterAttributes.VarianceMask mask to the GenericParameterAttributes value that is returned by the GenericParameterAttributes property. If the result is GenericParameterAttributes.None, the type parameter is invariant. See Covariance and Contravariance in Generics.

Special Constraints

To determine the special constraints of a type parameter, apply the GenericParameterAttributes.SpecialConstraintMask mask to the GenericParameterAttributes value that is returned by the GenericParameterAttributes property. If the result is GenericParameterAttributes.None, there are no special constraints. A type parameter can be constrained to be a reference type, to be a non-nullable value type, and to have a default constructor.

Back to top

Invariants

For a table of the invariant conditions for common terms in reflection for generic types, see Type.IsGenericType. For additional terms relating to generic methods, see MethodInfo.IsGenericMethod.

Back to top

Related Topics

Title	Description
How to: Examine and Instantiate Generic Types with Reflection	Shows how to use the properties and methods of Type and MethodInfo to examine generic types.
Generics in the .NET Framework	Describes the generics feature and how it is supported in the .NET Framework.
How to: Define a Generic Type with Reflection Emit	Shows how to use reflection emit to generate generic types in dynamic assemblies.
Viewing Type Information	Describes the Type class and provides code examples that illustrate how to use Type with various reflection classes to obtain information about constructors, methods, fields, properties, and events.

How to: Examine and Instantiate Generic Types with Reflection

.NET Framework (current version)

Information about generic types is obtained in the same way as information about other types: by examining a Type object that represents the generic type. The principle difference is that a generic type has a list of Type objects representing its generic type parameters. The first procedure in this section examines generic types.

You can create a Type object that represents a constructed type by binding type arguments to the type parameters of a generic type definition. The second procedure demonstrates this.

To examine a generic type and its type parameters

1. Get an instance of Type that represents the generic type. In the following code, the type is obtained using the C# **typeof** operator (**GetType** in Visual Basic, **typeid** in Visual C++). See the Type class topic for other ways to get a Type object. Note that in the rest of this procedure, the type is contained in a method parameter named `t`.

VB

```
Dim d1 As Type = GetType(Dictionary(Of ,))
```

2. Use the IsGenericType property to determine whether the type is generic, and use the IsGenericTypeDefinition property to determine whether the type is a generic type definition.

VB

```
Console.WriteLine("   Is this a generic type? " _
    & t.IsGenericType)
Console.WriteLine("   Is this a generic type definition? " _
    & t.IsGenericTypeDefinition)
```

3. Get an array that contains the generic type arguments, using the GetGenericArguments method.

VB

```
Dim typeParameters() As Type = t.GetGenericArguments()
```

4. For each type argument, determine whether it is a type parameter (for example, in a generic type definition) or a type that has been specified for a type parameter (for example, in a constructed type), using the IsGenericParameter property.

VB

```
Console.WriteLine("   List {0} type arguments:", _
        typeParameters.Length)
    For Each tParam As Type In typeParameters
        If tParam.IsGenericParameter Then
            DisplayGenericParameter(tParam)
        Else
            Console.WriteLine("      Type argument: {0}", _
                tParam)
        End If
    Next
```

5. In the type system, a generic type parameter is represented by an instance of Type, just as ordinary types are. The following code displays the name and parameter position of a Type object that represents a generic type parameter. The parameter position is trivial information here; it is of more interest when you are examining a type parameter that has been used as a type argument of another generic type.

VB

```
Private Shared Sub DisplayGenericParameter(ByVal tp As Type)
    Console.WriteLine("      Type parameter: {0} position {1}", _
        tp.Name, tp.GenericParameterPosition)
```

6. Determine the base type constraint and the interface constraints of a generic type parameter by using the GetGenericParameterConstraints method to obtain all the constraints in a single array. Constraints are not guaranteed to be in any particular order.

VB

```
Dim classConstraint As Type = Nothing

For Each iConstraint As Type In tp.GetGenericParameterConstraints()
    If iConstraint.IsInterface Then
        Console.WriteLine("         Interface constraint: {0}", _
            iConstraint)
    End If
Next

If classConstraint IsNot Nothing Then
    Console.WriteLine("         Base type constraint: {0}", _
        tp.BaseType)
Else
    Console.WriteLine("         Base type constraint: None")
End If
```

7. Use the GenericParameterAttributes property to discover the special constraints on a type parameter, such as requiring that it be a reference type. The property also includes values that represent variance, which you can mask off as shown in the following code.

VB

```
Dim sConstraints As GenericParameterAttributes = _
    tp.GenericParameterAttributes And _
    GenericParameterAttributes.SpecialConstraintMask
```

8. The special constraint attributes are flags, and the same flag (GenericParameterAttributes.None) that represents no special constraints also represents no covariance or contravariance. Thus, to test for either of these conditions you must use the appropriate mask. In this case, use GenericParameterAttributes.SpecialConstraintMask to isolate the special constraint flags.

VB

```
If sConstraints = GenericParameterAttributes.None Then
    Console.WriteLine("        No special constraints.")
Else
    If GenericParameterAttributes.None <> (sConstraints And _
        GenericParameterAttributes.DefaultConstructorConstraint) Then
        Console.WriteLine("        Must have a parameterless constructor.")
    End If
    If GenericParameterAttributes.None <> (sConstraints And _
        GenericParameterAttributes.ReferenceTypeConstraint) Then
        Console.WriteLine("        Must be a reference type.")
    End If
    If GenericParameterAttributes.None <> (sConstraints And _
        GenericParameterAttributes.NotNullableValueTypeConstraint) Then
        Console.WriteLine("        Must be a non-nullable value type.")
    End If
End If
```

Constructing an Instance of a Generic Type

A generic type is like a template. You cannot create instances of it unless you specify real types for its generic type parameters. To do this at run time, using reflection, requires the MakeGenericType method.

To construct an instance of a generic type

1. Get a Type object that represents the generic type. The following code gets the generic type Dictionary(Of TKey, TValue) in two different ways: by using the Type.GetType(String) method overload with a string describing the type, and by calling the GetGenericTypeDefinition method on the constructed type `Dictionary<String, Example>` (`Dictionary(Of String, Example)` in Visual Basic). The MakeGenericType method requires a generic type definition.

VB

```
' Use the GetType operator to create the generic type
' definition directly. To specify the generic type definition,
' omit the type arguments but retain the comma that separates
' them.
Dim d1 As Type = GetType(Dictionary(Of ,))

' You can also obtain the generic type definition from a
' constructed class. In this case, the constructed class
' is a dictionary of Example objects, with String keys.
Dim d2 As New Dictionary(Of String, Example)
' Get a Type object that represents the constructed type,
' and from that get the generic type definition. The
' variables d1 and d4 contain the same type.
Dim d3 As Type = d2.GetType()
Dim d4 As Type = d3.GetGenericTypeDefinition()
```

2. Construct an array of type arguments to substitute for the type parameters. The array must contain the correct number of Type objects, in the same order as they appear in the type parameter list. In this case, the key (first type parameter) is of type String, and the values in the dictionary are instances of a class named `Example`.

VB

```
Dim typeArgs() As Type = _
    { GetType(String), GetType(Example) }
```

3. Call the MakeGenericType method to bind the type arguments to the type parameters and construct the type.

VB

```
Dim constructed As Type = _
    d1.MakeGenericType(typeArgs)
```

4. Use the CreateInstance(Type) method overload to create an object of the constructed type. The following code stores two instances of the `Example` class in the resulting `Dictionary<String, Example>` object.

VB

```
Dim o As Object = Activator.CreateInstance(constructed)
```

Example

The following code example defines a `DisplayGenericType` method to examine the generic type definitions and constructed types used in the code and display their information. The `DisplayGenericType` method shows how to use the IsGenericType, IsGenericParameter, and GenericParameterPosition properties and the GetGenericArguments method.

The example also defines a `DisplayGenericParameter` method to examine a generic type parameter and display its constraints.

The code example defines a set of test types, including a generic type that illustrates type parameter constraints, and shows how to display information about these types.

The example constructs a type from the Dictionary(Of TKey, TValue) class by creating an array of type arguments and calling the MakeGenericType method. The program compares the Type object constructed using MakeGenericType with a Type object obtained using **typeof** (**GetType** in Visual Basic), demonstrating that they are the same. Similarly, the program uses the GetGenericTypeDefinition method to obtain the generic type definition of the constructed type, and compares it to the Type object representing the Dictionary(Of TKey, TValue) class.

VB

```
Imports System
Imports System.Reflection
Imports System.Collections.Generic
Imports System.Security.Permissions

' Define an example interface.
Public Interface ITestArgument
End Interface

' Define an example base class.
Public Class TestBase
End Class

' Define a generic class with one parameter. The parameter
' has three constraints: It must inherit TestBase, it must
' implement ITestArgument, and it must have a parameterless
' constructor.
Public Class Test(Of T As {TestBase, ITestArgument, New})
End Class

' Define a class that meets the constraints on the type
' parameter of class Test.
Public Class TestArgument
    Inherits TestBase
    Implements ITestArgument
    Public Sub New()
    End Sub
End Class

Public Class Example
    ' The following method displays information about a generic
    ' type.
    Private Shared Sub DisplayGenericType(ByVal t As Type)
        Console.WriteLine(vbCrLf & t.ToString())
        Console.WriteLine("   Is this a generic type? " _
            & t.IsGenericType)
        Console.WriteLine("   Is this a generic type definition? " _
            & t.IsGenericTypeDefinition)

        ' Get the generic type parameters or type arguments.
        Dim typeParameters() As Type = t.GetGenericArguments()

        Console.WriteLine("   List {0} type arguments:", _
            typeParameters.Length)
        For Each tParam As Type In typeParameters
            If tParam.IsGenericParameter Then
                DisplayGenericParameter(tParam)
            Else
                Console.WriteLine("      Type argument: {0}", _
                    tParam)
            End If
        Next
    End Sub

    ' The following method displays information about a generic
    ' type parameter. Generic type parameters are represented by
    ' instances of System.Type, just like ordinary types.
    Private Shared Sub DisplayGenericParameter(ByVal tp As Type)
        Console.WriteLine("      Type parameter: {0} position {1}", _
            tp.Name, tp.GenericParameterPosition)

        Dim classConstraint As Type = Nothing

        For Each iConstraint As Type In tp.GetGenericParameterConstraints()
            If iConstraint.IsInterface Then
                Console.WriteLine("         Interface constraint: {0}", _
                    iConstraint)
            End If
        Next

        If classConstraint IsNot Nothing Then
            Console.WriteLine("         Base type constraint: {0}", _
                tp.BaseType)
        Else
            Console.WriteLine("         Base type constraint: None")
        End If

        Dim sConstraints As GenericParameterAttributes = _
            tp.GenericParameterAttributes And _
            GenericParameterAttributes.SpecialConstraintMask
        If sConstraints = GenericParameterAttributes.None Then
            Console.WriteLine("         No special constraints.")
        Else
            If GenericParameterAttributes.None <> (sConstraints And _
                GenericParameterAttributes.DefaultConstructorConstraint) Then
                Console.WriteLine("         Must have a parameterless constructor.")
            End If
            If GenericParameterAttributes.None <> (sConstraints And _
                GenericParameterAttributes.ReferenceTypeConstraint) Then
                Console.WriteLine("         Must be a reference type.")
            End If
            If GenericParameterAttributes.None <> (sConstraints And _
                GenericParameterAttributes.NotNullableValueTypeConstraint) Then
                Console.WriteLine("         Must be a non-nullable value type.")
            End If
        End If
    End Sub

    <PermissionSetAttribute(SecurityAction.Demand, Name:="FullTrust")> _
    Public Shared Sub Main()
        ' Two ways to get a Type object that represents the generic
        ' type definition of the Dictionary class.
        '
        ' Use the GetType operator to create the generic type
        ' definition directly. To specify the generic type definition,
        ' omit the type arguments but retain the comma that separates
        ' them.
        Dim d1 As Type = GetType(Dictionary(Of ,))

        ' You can also obtain the generic type definition from a
        ' constructed class. In this case, the constructed class
        ' is a dictionary of Example objects, with String keys.
        Dim d2 As New Dictionary(Of String, Example)
        ' Get a Type object that represents the constructed type,
        ' and from that get the generic type definition. The
        ' variables d1 and d4 contain the same type.
        Dim d3 As Type = d2.GetType()
        Dim d4 As Type = d3.GetGenericTypeDefinition()

        ' Display information for the generic type definition, and
        ' for the constructed type Dictionary(Of String, Example).
        DisplayGenericType(d1)
        DisplayGenericType(d2.GetType())

        ' Construct an array of type arguments to substitute for
        ' the type parameters of the generic Dictionary class.
        ' The array must contain the correct number of types, in
        ' the same order that they appear in the type parameter
        ' list of Dictionary. The key (first type parameter)
        ' is of type string, and the type to be contained in the
        ' dictionary is Example.
        Dim typeArgs() As Type = _
            { GetType(String), GetType(Example) }

        ' Construct the type Dictionary(Of String, Example).
        Dim constructed As Type = _
            d1.MakeGenericType(typeArgs)

        DisplayGenericType(constructed)

        Dim o As Object = Activator.CreateInstance(constructed)

        Console.WriteLine(vbCrLf & _
            "Compare types obtained by different methods:")
        Console.WriteLine("   Are the constructed types equal? " _
            & (d2.GetType() Is constructed))
        Console.WriteLine("   Are the generic definitions equal? " _
            & (d1 Is constructed.GetGenericTypeDefinition()))

        ' Demonstrate the DisplayGenericType and
        ' DisplayGenericParameter methods with the Test class
        ' defined above. This shows base, interface, and special
        ' constraints.
        DisplayGenericType(GetType(Test(Of )))
    End Sub
End Class
```

Compiling the Code

- The code contains the C# **using** statements (**Imports** in Visual Basic) necessary for compilation.
- No additional assembly references are required.
- Compile the code at the command line using csc.exe, vbc.exe, or cl.exe. To compile the code in Visual Studio, place it in a console application project template.

See Also

Type
MethodInfo
Reflection and Generic Types
Viewing Type Information
Generics in the .NET Framework

Security Considerations for Reflection

.NET Framework (current version)

Reflection provides the ability to obtain information about types and members, and to access members (that is, to call methods and constructors, to get and set property values, to add and remove event handlers, and so on). The use of reflection to obtain information about types and members is not restricted. All code can use reflection to perform the following tasks:

- Enumerate types and members, and examine their metadata.
- Enumerate and examine assemblies and modules.

Using reflection to access members, by contrast, is subject to restrictions. Beginning with the .NET Framework 4, only trusted code can use reflection to access security-critical members. Furthermore, only trusted code can use reflection to access nonpublic members that would not be directly accessible to compiled code. Finally, code that uses reflection to access a safe-critical member must have whatever permissions the safe-critical member demands, just as with compiled code.

Subject to necessary permissions, code can use reflection to perform the following kinds of access:

- Access public members that are not security-critical.
- Access nonpublic members that would be accessible to compiled code, if those members are not security-critical. Examples of such nonpublic members include:
  - Protected members of the calling code's base classes. (In reflection, this is referred to as family-level access.)
  - **internal** members (**Friend** members in Visual Basic) in the calling code's assembly. (In reflection, this is referred to as assembly-level access.)
  - Private members of other instances of the class that contains the calling code.

For example, code that is run in a sandboxed application domain is limited to the access described in this list, unless the application domain grants additional permissions.

Starting with the .NET Framework 2.0 Service Pack 1, attempting to access members that are normally inaccessible generates a demand for the grant set of the target object plus ReflectionPermission with the ReflectionPermissionFlag.MemberAccess flag. Code that is running with full trust (for example, code in an application that is launched from the command line) can always satisfy these permissions. (This is subject to limitations in accessing security-critical members, as described later in this article.)

Optionally, a sandboxed application domain can grant ReflectionPermission with the ReflectionPermissionFlag.MemberAccess flag, as described in the section Accessing Members That Are Normally Inaccessible, later in this article.

Accessing Security-Critical Members

A member is security-critical if it has the SecurityCriticalAttribute, if it belongs to a type that has the SecurityCriticalAttribute, or if it is in a security-critical assembly. Beginning with the .NET Framework 4, the rules for accessing security-critical members are as follows:

- Transparent code cannot use reflection to access security-critical members, even if the code is fully trusted. A MethodAccessException, FieldAccessException, or TypeAccessException is thrown.
- Code that is running with partial trust is treated as transparent.

These rules are the same whether a security-critical member is accessed directly by compiled code, or accessed by using reflection.

Application code that is run from the command line runs with full trust. As long as it is not marked as transparent, it can use reflection to access security-critical members. When the same code is run with partial trust (for example, in a sandboxed application domain) the assembly's trust level determines whether it can access security-critical code: If the assembly has a strong name and is installed in the global assembly cache, it is a trusted assembly and can call security-critical members. If it is not trusted, it becomes transparent even though it was not marked as transparent, and it cannot access security-critical members.

For more information about the security model in the .NET Framework 4, see Security Changes in the .NET Framework.

Reflection and Transparency

Beginning with the .NET Framework 4, the common language runtime determines the transparency level of a type or member from several factors, including the trust level of the assembly and the trust level of the application domain. Reflection provides the IsSecurityCritical, IsSecuritySafeCritical, and IsSecurityTransparent properties to enable you to discover the transparency level of a type. The following table shows the valid combinations of these properties.

Security level	**IsSecurityCritical**	**IsSecuritySafeCritical**	**IsSecurityTransparent**
Critical	**true**	**false**	**false**
Safe-critical	**true**	**true**	**false**
Transparent	**false**	**false**	**true**

Using these properties is much simpler than examining the security annotations of an assembly and its types, checking the current trust level, and attempting to duplicate the runtime's rules. For example, the same type can be security-critical when it is run from the command line, or security-transparent when it is run in a sandboxed application domain.

There are similar properties on the MethodBase, FieldInfo, TypeBuilder, MethodBuilder, and DynamicMethod classes. (For other reflection and reflection emit abstractions, security attributes are applied to the associated methods; for example, in the case of properties they are applied to the property accessors.)

Accessing Members That Are Normally Inaccessible

To use reflection to invoke members that are inaccessible according to the accessibility rules of the common language runtime, your code must be granted one of two permissions:

- To allow code to invoke any nonpublic member: Your code must be granted ReflectionPermission with the ReflectionPermissionFlag.MemberAccess flag.

  Note

  By default, security policy denies this permission to code that originates from the Internet. This permission should never be granted to code that originates from the Internet.

- To allow code to invoke any nonpublic member, as long as the grant set of the assembly that contains the invoked member is the same as, or a subset of, the grant set of the assembly that contains the invoking code: Your code must be granted ReflectionPermission with the ReflectionPermissionFlag.RestrictedMemberAccess flag.

For example, suppose you grant an application domain Internet permissions plus ReflectionPermission with the ReflectionPermissionFlag.RestrictedMemberAccess flag, and then run an Internet application with two assemblies, A and B.

- Assembly A can use reflection to access private members of assembly B, because the grant set of assembly B does not include any permissions that A has not been granted.
- Assembly A cannot use reflection to access private members of .NET Framework assemblies such as mscorlib.dll, because mscorlib.dll is fully trusted and therefore has permissions that have not been granted to assembly A. A MemberAccessException is thrown when code access security walks the stack at run time.

Serialization

For serialization, SecurityPermission with the SecurityPermissionAttribute.SerializationFormatter flag provides the ability to get and set members of serializable types, regardless of accessibility. This permission enables code to discover and change the private state of an instance. (In addition to being granted the appropriate permissions, the type must be marked as serializable in metadata.)

Parameters of Type MethodInfo

Avoid writing public members that take MethodInfo parameters, especially for trusted code. Such members might be more vulnerable to malicious code. For example, consider a public member in highly trusted code that takes a MethodInfo parameter. Assume that the public member indirectly calls the Invoke method on the supplied parameter. If the public member does not perform the necessary permission checks, the call to the Invoke method will always succeed, because the security system determines that the caller is highly trusted. Even if malicious code does not have the permission to directly invoke the method, it can still do so indirectly by calling the public member.

Version Information

- Beginning with the .NET Framework 4, transparent code cannot use reflection to access security-critical members.
- The ReflectionPermissionFlag.RestrictedMemberAccess flag is introduced in the .NET Framework 2.0 Service Pack 1. Earlier versions of the .NET Framework require the ReflectionPermissionFlag.MemberAccess flag for code that uses reflection to access nonpublic members. This is a permission that should never be granted to partially trusted code.
- Beginning with the .NET Framework 2.0, using reflection to obtain information about nonpublic types and members does not require any permissions. In earlier versions, ReflectionPermission with the ReflectionPermissionFlag.TypeInformation flag is required.

See Also

ReflectionPermissionFlag
ReflectionPermission
SecurityPermission
Security Changes in the .NET Framework
Code Access Security
Security Issues in Reflection Emit
Viewing Type Information
Applying Attributes
Accessing Custom Attributes

Dynamically Loading and Using Types

.NET Framework (current version)

Reflection provides infrastructure used by language compilers such as Microsoft Visual Basic 2005 and JScript to implement implicit late binding. Binding is the process of locating the declaration (that is, the implementation) that corresponds to a uniquely specified type. When this process occurs at run time rather than at compile time, it is called late binding. Visual Basic 2005 allows you to use implicit late binding in your code; the Visual Basic compiler calls a helper method that uses reflection to obtain the object type. The arguments passed to the helper method cause the appropriate method to be invoked at run time. These arguments are the instance (an object) on which to invoke the method, the name of the invoked method (a string), and the arguments passed to the invoked method (an array of objects).

In the following example, the Visual Basic compiler uses reflection implicitly to call a method on an object whose type is not known at compile time. A **HelloWorld** class has a **PrintHello** method that prints out "Hello World" concatenated with some text that is passed to the **PrintHello** method. The **PrintHello** method called in this example is actually a Type.InvokeMember; the Visual Basic code allows the **PrintHello** method to be invoked as if the type of the object (helloObj) were known at compile time (early binding) rather than at run time (late binding).

```
Imports System
Module Hello
    Sub Main()
        ' Sets up the variable.
        Dim helloObj As Object
        ' Creates the object.
        helloObj = new HelloWorld()
        ' Invokes the print method as if it was early bound
        ' even though it is really late bound.
        helloObj.PrintHello("Visual Basic Late Bound")
    End Sub
End Module
```

Custom Binding

In addition to being used implicitly by compilers for late binding, reflection can be used explicitly in code to accomplish late binding.

The common language runtime supports multiple programming languages, and the binding rules of these languages differ. In the early-bound case, code generators can completely control this binding. However, in late binding through reflection, binding must be controlled by customized binding. The Binder class provides custom control of member selection and invocation.

Using custom binding, you can load an assembly at run time, obtain information about types in that assembly, specify the type that you want, and then invoke methods or access fields or properties on that type. This technique is useful if you do not know an object's type at compile time, such as when the object type is dependent on user input.

The following example demonstrates a simple custom binder that provides no argument type conversion. Code for

Simple_Type.dll precedes the main example. Be sure to build Simple_Type.dll and then include a reference to it in the project at build time.

VB

```
' Code for building SimpleType.dll.
Imports System
Imports System.Reflection
Imports System.Globalization
Imports Simple_Type

Namespace Simple_Type
    Public Class MySimpleClass
        Public Sub MyMethod(str As String, i As Integer)
            Console.WriteLine("MyMethod parameters: {0}, {1}", str, i)
        End Sub

        Public Sub MyMethod(str As String, i As Integer, j As Integer)
            Console.WriteLine("MyMethod parameters: {0}, {1}, {2}",
                str, i, j)
        End Sub
    End Class
End Namespace

Namespace Custom_Binder
    Class MyMainClass
        Shared Sub Main()
            ' Get the type of MySimpleClass.
            Dim myType As Type = GetType(MySimpleClass)

            ' Get an instance of MySimpleClass.
            Dim myInstance As New MySimpleClass()
            Dim myCustomBinder As New MyCustomBinder()

            ' Get the method information for the particular overload
            ' being sought.
            Dim myMethod As MethodInfo = myType.GetMethod("MyMethod",
                BindingFlags.Public Or BindingFlags.Instance,
                myCustomBinder, New Type() {GetType(String),
                GetType(Integer)}, Nothing)
            Console.WriteLine(myMethod.ToString())

            ' Invoke the overload.
            myType.InvokeMember("MyMethod", BindingFlags.InvokeMethod,
                myCustomBinder, myInstance,
                New Object() {"Testing...", CInt(32)})
        End Sub
    End Class

    ' ****************************************************
    '  A simple custom binder that provides no
    '  argument type conversion.
    ' ****************************************************
    Class MyCustomBinder
        Inherits Binder

        Public Overrides Function BindToMethod(bindingAttr As BindingFlags,
            match() As MethodBase, ByRef args As Object(),
            modIfiers() As ParameterModIfier, culture As CultureInfo,
            names() As String, ByRef state As Object) As MethodBase

            If match is Nothing Then
                Throw New ArgumentNullException("match")
            End If
            ' Arguments are not being reordered.
            state = Nothing
            ' Find a parameter match and return the first method with
            ' parameters that match the request.
            For Each mb As MethodBase in match
                Dim parameters() As ParameterInfo = mb.GetParameters()

                If ParametersMatch(parameters, args) Then
                    Return mb
                End If
            Next mb
            Return Nothing
        End Function

        Public Overrides Function BindToField(bindingAttr As BindingFlags,
            match() As FieldInfo, value As Object, culture As CultureInfo) As FieldInfo
            If match Is Nothing
                Throw New ArgumentNullException("match")
            End If
            For Each fi As FieldInfo in match
                If fi.GetType() = value.GetType() Then
                    Return fi
                End If
            Next fi
            Return Nothing
        End Function

        Public Overrides Function SelectMethod(bindingAttr As BindingFlags,
            match() As MethodBase, types() As Type,
            modifiers() As ParameterModifier) As MethodBase

            If match Is Nothing Then
                Throw New ArgumentNullException("match")
            End If

            ' Find a parameter match and return the first method with
            ' parameters that match the request.
            For Each mb As MethodBase In match
                Dim parameters() As ParameterInfo = mb.GetParameters()
                If ParametersMatch(parameters, types) Then
                    Return mb
                End If
            Next mb
```

```
            Return Nothing
        End Function

        Public Overrides Function SelectProperty(
            bindingAttr As BindingFlags, match() As PropertyInfo,
            returnType As Type, indexes() As Type,
            modIfiers() As ParameterModIfier) As PropertyInfo

            If match Is Nothing Then
                Throw New ArgumentNullException("match")
            End If
            For Each pi As PropertyInfo In match
                If pi.GetType() = returnType And
                    ParametersMatch(pi.GetIndexParameters(), indexes) Then
                    Return pi
                End If
            Next pi
            Return Nothing
        End Function

        Public Overrides Function ChangeType(
            value As Object,
            myChangeType As Type,
            culture As CultureInfo) As Object

            Try
                Dim newType As Object
                newType = Convert.ChangeType(value, myChangeType)
                Return newType
            ' Throw an InvalidCastException If the conversion cannot
            ' be done by the Convert.ChangeType method.
            Catch
                Return Nothing
            End Try
        End Function

        Public Overrides Sub ReorderArgumentArray(ByRef args() As Object, state As
Object)
            ' No operation is needed here because BindToMethod does not
            ' reorder the args array. The most common implementation
            ' of this method is shown below.

            ' ((BinderState)state).args.CopyTo(args, 0)
        End Sub

        ' Returns true only If the type of each object in a matches
        ' the type of each corresponding object in b.
        Private Overloads Function ParametersMatch(a() As ParameterInfo, b() As Object)
As Boolean
            If a.Length <> b.Length Then
                Return false
            End If
            For i As Integer = 0 To a.Length - 1
                If a(i).ParameterType <> b(i).GetType() Then
```

```
                    Return false
                End If
            Next i
            Return true
        End Function

        ' Returns true only If the type of each object in a matches
        ' the type of each corresponding enTry in b.
        Private Overloads Function ParametersMatch(a() As ParameterInfo,
            b() As Type) As Boolean

            If a.Length <> b.Length Then
                Return false
            End If
            For i As Integer = 0 To a.Length - 1
                If a(i).ParameterType <> b(i)
                    Return false
                End If
            Next
            Return true
        End Function
    End Class
End Namespace
```

InvokeMember and CreateInstance

Use Type.InvokeMember to invoke a member of a type. The **CreateInstance** methods of various classes, such as System.Activator and System.Reflection.Assembly, are specialized forms of **InvokeMember** that create new instances of the specified type. The **Binder** class is used for overload resolution and argument coercion in these methods.

The following example shows the three possible combinations of argument coercion (type conversion) and member selection. In Case 1, no argument coercion or member selection is needed. In Case 2, only member selection is needed. In Case 3, only argument coercion is needed.

VB

```
Public Class CustomBinderDriver
    Public Shared Sub Main()
        Dim t As Type = GetType(CustomBinderDriver)
        Dim binder As New CustomBinder()
        Dim flags As BindingFlags = BindingFlags.InvokeMethod Or
BindingFlags.Instance Or
            BindingFlags.Public Or BindingFlags.Static
        Dim args() As Object

        ' Case 1. Neither argument coercion nor member selection is needed.
        args = New object() {}
        t.InvokeMember ("PrintBob", flags, binder, Nothing, args)

        ' Case 2. Only member selection is needed.
        args = New object() {42}
        t.InvokeMember ("PrintValue", flags, binder, Nothing, args)
```

```
            ' Case 3. Only argument coercion is needed.
            args = New object() {"5.5"}
            t.InvokeMember("PrintNumber", flags, binder, Nothing, args)
        End Sub

        Public Shared Sub PrintBob()
            Console.WriteLine ("PrintBob")
        End Sub

        Public Shared Sub PrintValue(value As Long)
            Console.WriteLine("PrintValue ({0})", value)
        End Sub

        Public Shared Sub PrintValue(value As String)
            Console.WriteLine("PrintValue ""{0}"")", value)
        End Sub

        Public Shared Sub PrintNumber(value As Double)
            Console.WriteLine("PrintNumber ({0})", value)
        End Sub
    End Class
```

Overload resolution is needed when more than one member with the same name is available. The Binder.BindToMethod and Binder.BindToField methods are used to resolve binding to a single member. **Binder.BindToMethod** also provides property resolution through the **get** and **set** property accessors.

BindToMethod returns the MethodBase to invoke, or a null reference (**Nothing** in Visual Basic) if no such invocation is possible. The **MethodBase** return value need not be one of those contained in the *match* parameter, although that is the usual case.

When ByRef arguments are present, the caller might want to get them back. Therefore, **Binder** allows a client to map the array of arguments back to its original form if **BindToMethod** has manipulated the argument array. In order to do this, the caller must be guaranteed that the order of the arguments is unchanged. When arguments are passed by name, **Binder** reorders the argument array, and that is what the caller sees. For more information, see Binder.ReorderArgumentArray.

The set of available members are those members defined in the type or any base type. If BindingFlags.NonPublic is specified, members of any accessibility will be returned in the set. If **BindingFlags.NonPublic** is not specified, the binder must enforce accessibility rules. When specifying the **Public** or **NonPublic** binding flag, you must also specify the **Instance** or **Static** binding flag, or no members will be returned.

If there is only one member of the given name, no callback is necessary, and binding is done on that method. Case 1 of the code example illustrates this point: Only one **PrintBob** method is available, and therefore no callback is needed.

If there is more than one member in the available set, all these methods are passed to **BindToMethod**, which selects the appropriate method and returns it. In Case 2 of the code example, there are two methods named **PrintValue**. The appropriate method is selected by the call to **BindToMethod**.

ChangeType performs argument coercion (type conversion), which converts the actual arguments to the type of the formal arguments of the selected method. **ChangeType** is called for every argument even if the types match exactly.

In Case 3 of the code example, an actual argument of type **String** with a value of "5.5" is passed to a method with a formal argument of type **Double**. For the invocation to succeed, the string value "5.5" must be converted to a double value. **ChangeType** performs this conversion.

ChangeType performs only lossless or widening coercions, as shown in the following table.

Source type	**Target type**
Any type	Its base type
Any type	Interface it implements
Char	UInt16, UInt32, Int32, UInt64, Int64, Single, Double
Byte	Char, UInt16, Int16, UInt32, Int32, UInt64, Int64, Single, Double
SByte	Int16, Int32, Int64, Single, Double
UInt16	UInt32, Int32, UInt64, Int64, Single, Double
Int16	Int32, Int64, Single, Double
UInt32	UInt64, Int64, Single, Double
Int32	Int64, Single, Double
UInt64	Single, Double
Int64	Single, Double
Single	Double
Nonreference type	Reference type

The Type class has **Get** methods that use parameters of type **Binder** to resolve references to a particular member. Type.GetConstructor, Type.GetMethod, and Type.GetProperty search for a particular member of the current type by providing signature information for that member. Binder.SelectMethod and Binder.SelectProperty are called back on to select the given signature information of the appropriate methods.

See Also

Type.InvokeMember
Assembly.Load
Viewing Type Information
Type Conversion in the .NET Framework

How to: Load Assemblies into the Reflection-Only Context

.NET Framework (current version)

The reflection-only load context allows you to examine assemblies compiled for other platforms or for other versions of the .NET Framework. Code loaded into this context can only be examined; it cannot be executed. This means that objects cannot be created, because constructors cannot be executed. Because the code cannot be executed, dependencies are not automatically loaded. If you need to examine them, you must load them yourself.

To load an assembly into the reflection-only load context

1. Use the ReflectionOnlyLoad(String) method overload to load the assembly given its display name, or the ReflectionOnlyLoadFrom method to load the assembly given its path. If the assembly is a binary image, use the ReflectionOnlyLoad(Byte()) method overload.

   Note

   You cannot use the reflection-only context to load a version of mscorlib.dll from a version of the .NET Framework other than the version in the execution context.

2. If the assembly has dependencies, the ReflectionOnlyLoad method does not load them. If you need to examine them, you must load them yourself,.

3. Determine whether an assembly is loaded into the reflection-only context by using the assembly's ReflectionOnly property.

4. If attributes have been applied to the assembly or to types in the assembly, examine those attributes by using the CustomAttributeData class to ensure that no attempt is made to execute code in the reflection-only context. Use the appropriate overload of the CustomAttributeData.GetCustomAttributes method to obtain CustomAttributeData objects representing the attributes applied to an assembly, member, module, or parameter.

   Note

   Attributes applied to the assembly or to its contents might be defined in the assembly, or they might be defined in another assembly loaded into the reflection-only context. There is no way to tell in advance where the attributes are defined.

Example

The following code example shows how to examine the attributes applied to an assembly loaded into the reflection-only context.

The code example defines a custom attribute with two constructors and one property. The attribute is applied to the assembly, to a type declared in the assembly, to a method of the type, and to a parameter of the method. When executed, the assembly loads itself into the reflection-only context and displays information about the custom attributes that were applied to it and to the types and members it contains.

Note

To simplify the code example, the assembly loads and examines itself. Normally, you would not expect to find the same assembly loaded into both the execution context and the reflection-only context.

VB

```vb
Imports System
Imports System.Reflection
Imports System.Collections.Generic
Imports System.Collections.ObjectModel

' The example attribute is applied to the assembly.
<Assembly:Example(ExampleKind.ThirdKind, Note:="This is a note on the assembly.")>

' An enumeration used by the ExampleAttribute class.
Public Enum ExampleKind
    FirstKind
    SecondKind
    ThirdKind
    FourthKind
End Enum

' An example attribute. The attribute can be applied to all
' targets, from assemblies to parameters.
'
<AttributeUsage(AttributeTargets.All)> _
Public Class ExampleAttribute
    Inherits Attribute

    ' Data for properties.
    Private kindValue As ExampleKind
    Private noteValue As String
    Private arrayStrings() As String
    Private arrayNumbers() As Integer

    ' Constructors. The parameterless constructor (.ctor) calls
    ' the constructor that specifies ExampleKind and an array of
    ' strings, and supplies the default values.
    '
    Public Sub New(ByVal initKind As ExampleKind, ByVal initStrings() As String)
        kindValue = initKind
        arrayStrings = initStrings
    End Sub
    Public Sub New(ByVal initKind As ExampleKind)
        Me.New(initKind, Nothing)
    End Sub
    Public Sub New()
        Me.New(ExampleKind.FirstKind, Nothing)
    End Sub

    ' Properties. The Note and Numbers properties must be read/write, so they
    ' can be used as named parameters.
    '
    Public ReadOnly Property Kind As ExampleKind
        Get
            Return kindValue
        End Get
    End Property
    Public ReadOnly Property Strings As String()
        Get
            Return arrayStrings
        End Get
    End Property
    Public Property Note As String
        Get
            Return noteValue
        End Get
        Set
            noteValue = value
        End Set
    End Property
    Public Property Numbers As Integer()
        Get
            Return arrayNumbers
        End Get
        Set
            arrayNumbers = value
        End Set
    End Property
End Class

' The example attribute is applied to the test class.
'
<Example(ExampleKind.SecondKind, _
         New String() { "String array argument, line 1", _
                        "String array argument, line 2", _
                        "String array argument, line 3" }, _
         Note := "This is a note on the class.", _
         Numbers := New Integer() { 53, 57, 59 })> _
Public Class Test
    ' The example attribute is applied to a method, using the
    ' parameterless constructor and supplying a named argument.
    ' The attribute is also applied to the method parameter.
    '
    <Example(Note:="This is a note on a method.")> _
    Public Sub TestMethod(<Example()> ByVal arg As Object)
    End Sub
    ' Sub Main gets objects representing the assembly, the test
    ' type, the test method, and the method parameter. Custom
    ' attribute data is displayed for each of these.
    '
    Public Shared Sub Main()
        Dim asm As [Assembly] = Assembly.ReflectionOnlyLoad("source")
        Dim t As Type = asm.GetType("Test")
        Dim m As MethodInfo = t.GetMethod("TestMethod")
        Dim p() As ParameterInfo = m.GetParameters()

        Console.WriteLine(vbCrLf & "Attributes for assembly: '{0}'", asm)
        ShowAttributeData(CustomAttributeData.GetCustomAttributes(asm))
        Console.WriteLine(vbCrLf & "Attributes for type: '{0}'", t)
        ShowAttributeData(CustomAttributeData.GetCustomAttributes(t))
        Console.WriteLine(vbCrLf & "Attributes for member: '{0}'", m)
        ShowAttributeData(CustomAttributeData.GetCustomAttributes(m))
        Console.WriteLine(vbCrLf & "Attributes for parameter: '{0}'", p)
        ShowAttributeData(CustomAttributeData.GetCustomAttributes(p(0)))
    End Sub

    Private Shared Sub ShowAttributeData( _
        ByVal attributes As IList(Of CustomAttributeData))

        For Each cad As CustomAttributeData _
            In CType(attributes, IEnumerable(Of CustomAttributeData))

            Console.WriteLine("   {0}", cad)
            Console.WriteLine("      Constructor: '{0}'", cad.Constructor)

            Console.WriteLine("      Constructor arguments:")
            For Each cata As CustomAttributeTypedArgument _
                In CType(cad.ConstructorArguments, IEnumerable(Of
CustomAttributeTypedArgument))

                ShowValueOrArray(cata)
            Next

            Console.WriteLine("      Named arguments:")
            For Each cana As CustomAttributeNamedArgument _
                In CType(cad.NamedArguments, IEnumerable(Of CustomAttributeNamedArgument))

                Console.WriteLine("         MemberInfo: '{0}'", _
                    cana.MemberInfo)
                ShowValueOrArray(cana.TypedValue)
            Next
        Next
    End Sub

    Private Shared Sub ShowValueOrArray(ByVal cata As CustomAttributeTypedArgument)
        If cata.Value.GetType() Is GetType(ReadOnlyCollection(Of
CustomAttributeTypedArgument)) Then
            Console.WriteLine("         Array of '{0}':", cata.ArgumentType)
            For Each cataElement As CustomAttributeTypedArgument In cata.Value
                Console.WriteLine("             Type: '{0}'  Value: '{1}'", _
                    cataElement.ArgumentType, cataElement.Value)
            Next
        Else
            Console.WriteLine("         Type: '{0}'  Value: '{1}'", _
                cata.ArgumentType, cata.Value)
        End If
    End Sub
End Class

' This code example produces output similar to the following:
'
'Attributes for assembly: 'source, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null'
'   [System.Runtime.CompilerServices.CompilationRelaxationsAttribute((Int32)8)]
'      Constructor: 'Void .ctor(Int32)'
'      Constructor arguments:
'         Type: 'System.Int32'  Value: '8'
'      Named arguments:
'   [System.Runtime.CompilerServices.RuntimeCompatibilityAttribute(WrapNonExceptionThrows
= True)]
'      Constructor: 'Void .ctor()'
'      Constructor arguments:
'      Named arguments:
'         MemberInfo: 'Boolean WrapNonExceptionThrows'
'         Type: 'System.Boolean'  Value: 'True'
'   [ExampleAttribute((ExampleKind)2, Note = "This is a note on the assembly.")]
'      Constructor: 'Void .ctor(ExampleKind)'
'      Constructor arguments:
'         Type: 'ExampleKind'  Value: '2'
'      Named arguments:
'         MemberInfo: 'System.String Note'
'         Type: 'System.String'  Value: 'This is a note on the assembly.'
'
'Attributes for type: 'Test'
'   [ExampleAttribute((ExampleKind)1, new String[3] { "String array argument, line 1",
"String array argument, line 2", "String array argument, line 3" }, Note = "This is a
note on the class.", Numbers = new Int32[3] { 53, 57, 59 })]
'      Constructor: 'Void .ctor(ExampleKind, System.String[])'
'      Constructor arguments:
'         Type: 'ExampleKind'  Value: '1'
'         Array of 'System.String[]':
'             Type: 'System.String'  Value: 'String array argument, line 1'
'             Type: 'System.String'  Value: 'String array argument, line 2'
'             Type: 'System.String'  Value: 'String array argument, line 3'
'      Named arguments:
'         MemberInfo: 'System.String Note'
'         Type: 'System.String'  Value: 'This is a note on the class.'
'         MemberInfo: 'Int32[] Numbers'
'         Array of 'System.Int32[]':
'             Type: 'System.Int32'  Value: '53'
'             Type: 'System.Int32'  Value: '57'
'             Type: 'System.Int32'  Value: '59'
'
```

```
'Attributes for member: 'Void TestMethod(System.Object)'
'   [ExampleAttribute(Note = "This is a note on a method.")]
'      Constructor: 'Void .ctor()'
'      Constructor arguments:
'      Named arguments:
'         MemberInfo: 'System.String Note'
'         Type: 'System.String'  Value: 'This is a note on a method.'
'
'Attributes for parameter: 'System.Object arg'
'   [ExampleAttribute()]
'      Constructor: 'Void .ctor()'
'      Constructor arguments:
'      Named arguments:
```

See Also

ReflectionOnlyLoad
ReflectionOnly
CustomAttributeData

Accessing Custom Attributes

.NET Framework (current version)

After attributes have been associated with program elements, reflection can be used to query their existence and values. In the .NET Framework version 1.0 and 1.1, custom attributes are examined in the execution context. The .NET Framework version 2.0 provides a new load context, the reflection-only context, which can be used to examine code that cannot be loaded for execution.

The Reflection-Only Context

Code loaded into the reflection-only context cannot be executed. This means that instances of custom attributes cannot be created, because that would require executing their constructors. To load and examine custom attributes in the reflection-only context, use the CustomAttributeData class. You can obtain instances of this class by using the appropriate overload of the static CustomAttributeData.GetCustomAttributes method. See How to: Load Assemblies into the Reflection-Only Context.

The Execution Context

The main reflection methods to query attributes in the execution context are MemberInfo.GetCustomAttributes and Attribute.GetCustomAttributes.

The accessibility of a custom attribute is checked with respect to the assembly in which it is attached. This is equivalent to checking whether a method on a type in the assembly in which the custom attribute is attached can call the constructor of the custom attribute.

Methods such as Assembly.GetCustomAttributes(Boolean) check the visibility and accessibility of the type argument. Only code in the assembly that contains the user-defined type can retrieve a custom attribute of that type using **GetCustomAttributes**.

The following C# example is a typical custom attribute design pattern. It illustrates the runtime custom attribute reflection model.

```
System.DLL
public class DescriptionAttribute : Attribute
{
}

System.Web.DLL
internal class MyDescriptionAttribute : DescriptionAttribute
{
}

public class LocalizationExtenderProvider
{
    [MyDescriptionAttribute(...)]
    public CultureInfo GetLanguage(...)
    {
    }
}
```

If the runtime is attempting to retrieve the custom attributes for the public custom attribute type DescriptionAttribute attached to the **GetLanguage** method, it performs the following actions:

1. The runtime checks that the type argument **DescriptionAttribute** to **Type.GetCustomAttributes**(Type *type*) is public, and therefore is visible and accessible.
2. The runtime checks that the user-defined type **MyDescriptionAttribute** that is derived from **DescriptionAttribute** is visible and accessible within the **System.Web.DLL** assembly, where it is attached to the method **GetLanguage**().
3. The runtime checks that the constructor of **MyDescriptionAttribute** is visible and accessible within the **System.Web.DLL** assembly.
4. The runtime calls the constructor of **MyDescriptionAttribute** with the custom attribute parameters and returns the new object to the caller.

The custom attribute reflection model could leak instances of user-defined types outside the assembly in which the type is defined. This is no different from the members in the runtime system library that return instances of user-defined types, such as Type.GetMethods() returning an array of **RuntimeMethodInfo** objects. To prevent a client from discovering information about a user-defined custom attribute type, define the type's members to be nonpublic.

The following example demonstrates the basic way of using reflection to get access to custom attributes.

VB

```
Imports System

Public Class ExampleAttribute
    Inherits Attribute

    Private stringVal As String

    Public Sub New()
        stringVal = "This is the default string."
    End Sub

    Public Property StringValue() As String
        Get
            Return stringVal
        End Get
        Set(Value As String)
            stringVal = Value
        End Set
    End Property
End Class
```

```
<Example(StringValue := "This is a string.")> _
Class Class1
    Public Shared Sub Main()
    Dim info As System.Reflection.MemberInfo = GetType(Class1)
        For Each attrib As Object In info.GetCustomAttributes(true)
            Console.WriteLine(attrib)
        Next attrib
    End Sub
End Class
```

See Also

MemberInfo.GetCustomAttributes
Attribute.GetCustomAttributes
Viewing Type Information
Security Considerations for Reflection

Specifying Fully Qualified Type Names

.NET Framework (current version)

You must specify type names to have valid input to various reflection operations. A fully qualified type name consists of an assembly name specification, a namespace specification, and a type name. Type name specifications are used by methods such as Type.GetType, Module.GetType, ModuleBuilder.GetType, and Assembly.GetType.

Backus-Naur Form Grammar for Type Names

The Backus-Naur form (BNF) defines the syntax of formal languages. The following table lists BNF lexical rules that describe how to recognize a valid input. Terminals (those elements that are not further reducible) are shown in all uppercase letters. Nonterminals (those elements that are further reducible) are shown in mixed-case or singly quoted strings, but the single quote (') is not a part of the syntax itself. The pipe character (|) denotes rules that have subrules.

BNF grammar of fully qualified type names
TypeSpec := ReferenceTypeSpec | SimpleTypeSpec
ReferenceTypeSpec := SimpleTypeSpec '&'
SimpleTypeSpec := PointerTypeSpec | ArrayTypeSpec | TypeName
PointerTypeSpec := SimpleTypeSpec '*'
ArrayTypeSpec := SimpleTypeSpec '[ReflectionDimension]' | SimpleTypeSpec '[ReflectionEmitDimension]'
ReflectionDimension := '*' | ReflectionDimension ',' ReflectionDimension | NOTOKEN
ReflectionEmitDimension := '*' | Number '..' Number

	| Number '...'
	| ReflectionDimension ',' ReflectionDimension
	| NOTOKEN
Number	:= [0-9]+
TypeName	:= NamespaceTypeName
	| NamespaceTypeName ',' AssemblyNameSpec
NamespaceTypeName	:= NestedTypeName
	| NamespaceSpec '.' NestedTypeName
NestedTypeName	:= IDENTIFIER
	| NestedTypeName '+' IDENTIFIER
NamespaceSpec	:= IDENTIFIER
	| NamespaceSpec '.' IDENTIFIER
AssemblyNameSpec	:= IDENTIFIER
	| IDENTIFIER ',' AssemblyProperties
AssemblyProperties	:= AssemblyProperty
	| AssemblyProperties ',' AssemblyProperty
AssemblyProperty	:= AssemblyPropertyName '=' AssemblyPropertyValue

Specifying Special Characters

In a type name, IDENTIFIER is any valid name determined by the rules of a language.

Use the backslash (\) as an escape character to separate the following tokens when used as part of IDENTIFIER.

Token	Meaning
\\,	Assembly separator.
\\+	Nested type separator.
\\&	Reference type.

*	Pointer type.
\\[	Array dimension delimiter.
\\]	Array dimension delimiter.
\\.	Use the backslash before a period only if the period is used in an array specification. Periods in NamespaceSpec do not take the backslash.
\\\\	Backslash when needed as a string literal.

Note that in all TypeSpec components except AssemblyNameSpec, spaces are relevant. In the AssemblyNameSpec, spaces before the ',' separator are relevant, but spaces after the ',' separator are ignored.

Reflection classes, such as Type.FullName, return the mangled name so that the returned name can be used in a call to GetType, as in `MyType.GetType(myType.FullName)`.

For example, the fully qualified name for a type might be `Ozzy.OutBack.Kangaroo+Wallaby,MyAssembly`.

If the namespace were `Ozzy.Out+Back`, then the plus sign must be preceded by a backslash. Otherwise, the parser would interpret it as a nesting separator. Reflection emits this string as `Ozzy.Out\+Back.Kangaroo+Wallaby,MyAssembly`.

Specifying Assembly Names

The minimum information required in an assembly name specification is the textual name (IDENTIFIER) of the assembly. You can follow the IDENTIFIER by a comma-separated list of property/value pairs as described in the following table. IDENTIFIER naming should follow the rules for file naming. The IDENTIFIER is case-insensitive.

Property name	**Description**	**Allowable values**
Version	Assembly version number	*Major.Minor.Build.Revision*, where *Major*, *Minor*, *Build*, and *Revision* are integers between 0 and 65535 inclusive.
PublicKey	Full public key	String value of full public key in hexadecimal format. Specify a null reference (**Nothing** in Visual Basic) to explicitly indicate a private assembly.
PublicKeyToken	Public key token (8-byte hash of the full public key)	String value of public key token in hexadecimal format. Specify a null reference (**Nothing** in Visual Basic) to explicitly indicate a private assembly.
Culture	Assembly culture	Culture of the assembly in RFC-1766 format, or "neutral" for language-independent (nonsatellite) assemblies.
Custom	Custom binary large object (BLOB). This is currently used only	Custom string used by the Native Image Generator tool to notify the assembly cache that the assembly being installed is

	in assemblies generated by the Native Image Generator (Ngen).	a native image, and is therefore to be installed in the native image cache. Also called a zap string.

The following example shows an **AssemblyName** for a simply named assembly with default culture.

C#

```
com.microsoft.crypto, Culture=""
```

The following example shows a fully specified reference for a strongly named assembly with culture "en".

C#

```
com.microsoft.crypto, Culture=en, PublicKeyToken=a5d015c7d5a0b012,
    Version=1.0.0.0
```

The following examples each show a partially specified **AssemblyName**, which can be satisfied by either a strong or a simply named assembly.

C#

```
com.microsoft.crypto
com.microsoft.crypto, Culture=""
com.microsoft.crypto, Culture=en
```

The following examples each show a partially specified **AssemblyName**, which must be satisfied by a simply named assembly.

C#

```
com.microsoft.crypto, Culture="", PublicKeyToken=null
com.microsoft.crypto, Culture=en, PublicKeyToken=null
```

The following examples each show a partially specified **AssemblyName**, which must be satisfied by a strongly named assembly.

C#

```
com.microsoft.crypto, Culture="", PublicKeyToken=a5d015c7d5a0b012
com.microsoft.crypto, Culture=en, PublicKeyToken=a5d015c7d5a0b012,
    Version=1.0.0.0
```

Specifying Pointers

SimpleTypeSpec* represents an unmanaged pointer. For example, to get a pointer to type MyType, use `Type.GetType("MyType*")`. To get a pointer to a pointer to type MyType, use `Type.GetType("MyType**")`.

Specifying References

SimpleTypeSpec & represents a managed pointer or reference. For example, to get a reference to type MyType, use `Type.GetType("MyType &")`. Note that unlike pointers, references are limited to one level.

Specifying Arrays

In the BNF Grammar, ReflectionEmitDimension only applies to incomplete type definitions retrieved using ModuleBuilder.GetType. Incomplete type definitions are TypeBuilder objects constructed using Reflection.Emit but on which TypeBuilder.CreateType has not been called. ReflectionDimension can be used to retrieve any type definition that has been completed, that is, a type that has been loaded.

Arrays are accessed in reflection by specifying the rank of the array:

- `Type.GetType("MyArray[]")` gets a single-dimension array with 0 lower bound.
- `Type.GetType("MyArray[*]")` gets a single-dimension array with unknown lower bound.
- `Type.GetType("MyArray[][]")` gets a two-dimensional array's array.
- `Type.GetType("MyArray[*,*]")` and `Type.GetType("MyArray[,]")` gets a rectangular two-dimensional array with unknown lower bounds.

Note that from a runtime point of view, `MyArray[] != MyArray[*]`, but for multidimensional arrays, the two notations are equivalent. That is, `Type.GetType("MyArray [,]") == Type.GetType("MyArray[*,*]")` evaluates to **true**.

For **ModuleBuilder.GetType**, `MyArray[0..5]` indicates a single-dimension array with size 6, lower bound 0. `MyArray[4…]` indicates a single-dimension array of unknown size and lower bound 4.

See Also

AssemblyName
ModuleBuilder
TypeBuilder
Type.FullName
Type.GetType
Type.AssemblyQualifiedName
Viewing Type Information

How to: Hook Up a Delegate Using Reflection

.NET Framework (current version)

When you use reflection to load and run assemblies, you cannot use language features like the C# **+=** operator or the Visual Basic AddHandler statement to hook up events. The following procedures show how to hook up an existing method to an event by getting all the necessary types through reflection, and how to create a dynamic method using reflection emit and hook it up to an event.

Note

For another way to hook up an event-handling delegate, see the code example for the AddEventHandler method of the EventInfo class.

To hook up a delegate using reflection

1. Load an assembly that contains a type that raises events. Assemblies are usually loaded with the Assembly.Load method. To keep this example simple, a derived form in the current assembly is used, so the GetExecutingAssembly method is used to load the current assembly.

VB

```
Dim assem As Assembly = GetType(Example).Assembly
```

2. Get a Type object representing the type, and create an instance of the type. The CreateInstance(Type) method is used in the following code because the form has a default constructor. There are several other overloads of the CreateInstance method that you can use if the type you are creating does not have a default constructor. The new instance is stored as type Object to maintain the fiction that nothing is known about the assembly. (Reflection allows you to get the types in an assembly without knowing their names in advance.)

VB

```
Dim tExForm As Type = assem.GetType("ExampleForm")
Dim exFormAsObj As Object = _
    Activator.CreateInstance(tExForm)
```

3. Get an EventInfo object representing the event, and use the EventHandlerType property to get the type of delegate used to handle the event. In the following code, an EventInfo for the Click event is obtained.

VB

```
Dim evClick As EventInfo = tExForm.GetEvent("Click")
Dim tDelegate As Type = evClick.EventHandlerType
```

4. Get a MethodInfo object representing the method that handles the event. The complete program code in the Example section later in this topic contains a method that matches the signature of the EventHandler delegate, which handles the Click event, but you can also generate dynamic methods at run time. For details, see the accompanying procedure, for generating an event handler at run time by using a dynamic method.

VB

```
Dim miHandler As MethodInfo = _
    GetType(Example).GetMethod("LuckyHandler", _
        BindingFlags.NonPublic Or BindingFlags.Instance)
```

5. Create an instance of the delegate, using the CreateDelegate method. This method is static (**Shared** in Visual Basic), so the delegate type must be supplied. Using the overloads of CreateDelegate that take a MethodInfo is recommended.

VB

```
Dim d As [Delegate] = _
    [Delegate].CreateDelegate(tDelegate, Me, miHandler)
```

6. Get the **add** accessor method and invoke it to hook up the event. All events have an **add** accessor and a **remove** accessor, which are hidden by the syntax of high-level languages. For example, C# uses the **+=** operator to hook up events, and Visual Basic uses the AddHandler statement. The following code gets the **add** accessor of the Click event and invokes it late-bound, passing in the delegate instance. The arguments must be passed as an array.

VB

```
Dim miAddHandler As MethodInfo = evClick.GetAddMethod()
Dim addHandlerArgs() As Object = { d }
miAddHandler.Invoke(exFormAsObj, addHandlerArgs)
```

7. Test the event. The following code shows the form defined in the code example. Clicking the form invokes the event handler.

VB

```
Application.Run(CType(exFormAsObj, Form))
```

To generate an event handler at run time by using a dynamic method

1. Event-handler methods can be generated at run time, using lightweight dynamic methods and reflection emit. To construct an event handler, you need the return type and parameter types of the delegate. These can be obtained by examining the delegate's **Invoke** method. The following code uses the `GetDelegateReturnType` and `GetDelegateParameterTypes` methods to obtain this information. The code for these methods can be found in the Example section later in this topic.

   It is not necessary to name a DynamicMethod, so the empty string can be used. In the following code, the last argument associates the dynamic method with the current type, giving the delegate access to all the public and private members of the `Example` class.

VB

```
Dim returnType As Type = GetDelegateReturnType(tDelegate)
If returnType IsNot GetType(Void) Then
    Throw New ApplicationException("Delegate has a return type.")
End If

Dim handler As New DynamicMethod( _
    "", _
    Nothing, _
    GetDelegateParameterTypes(tDelegate), _
    GetType(Example) _
)
```

2. Generate a method body. This method loads a string, calls the overload of the MessageBox.Show method that takes a string, pops the return value off the stack (because the handler has no return type), and returns. To learn more about emitting dynamic methods, see How to: Define and Execute Dynamic Methods.

VB

```
Dim ilgen As ILGenerator = handler.GetILGenerator()

Dim showParameters As Type() = { GetType(String) }
Dim simpleShow As MethodInfo = _
    GetType(MessageBox).GetMethod("Show", showParameters)

ilgen.Emit(OpCodes.Ldstr, _
    "This event handler was constructed at run time.")
ilgen.Emit(OpCodes.Call, simpleShow)
ilgen.Emit(OpCodes.Pop)
ilgen.Emit(OpCodes.Ret)
```

3. Complete the dynamic method by calling its CreateDelegate method. Use the **add** accessor to add the delegate to the invocation list for the event.

VB

```
Dim dEmitted As [Delegate] = handler.CreateDelegate(tDelegate)
miAddHandler.Invoke(exFormAsObj, New Object() { dEmitted })
```

4. Test the event. The following code loads the form defined in the code example. Clicking the form invokes both the predefined event handler and the emitted event handler.

VB

```
Application.Run(CType(exFormAsObj, Form))
```

Example

The following code example shows how to hook up an existing method to an event using reflection, and also how to use the DynamicMethod class to emit a method at run time and hook it up to an event.

VB

```
Imports System.Reflection
Imports System.Reflection.Emit
Imports System.Windows.Forms

Class ExampleForm
    Inherits Form

    Public Sub New()
        Me.Text = "Click me"

    End Sub 'NewNew
End Class 'ExampleForm

Class Example
    Public Shared Sub Main()
        Dim ex As New Example()
        ex.HookUpDelegate()
    End Sub 'Main

    Private Sub HookUpDelegate()
        ' Load an assembly, for example using the Assembly.Load
        ' method. In this case, the executing assembly is loaded, to
        ' keep the demonstration simple.
        '
        Dim assem As Assembly = GetType(Example).Assembly

        ' Get the type that is to be loaded, and create an instance
        ' of it. Activator.CreateInstance also has an overload that
        ' takes an array of types representing the types of the
        ' constructor parameters, if the type you are creating does
        ' not have a parameterless constructor. The new instance
        ' is stored as type Object, to maintain the fiction that
        ' nothing is known about the assembly. (Note that you can
        ' get the types in an assembly without knowing their names
        ' in advance.)
        '
        Dim tExForm As Type = assem.GetType("ExampleForm")
        Dim exFormAsObj As Object = _
            Activator.CreateInstance(tExForm)

        ' Get an EventInfo representing the Click event, and get the
        ' type of delegate that handles the event.
        '
        Dim evClick As EventInfo = tExForm.GetEvent("Click")
        Dim tDelegate As Type = evClick.EventHandlerType

        ' If you already have a method with the correct signature,
        ' you can simply get a MethodInfo for it.
        '
        Dim miHandler As MethodInfo = _
            GetType(Example).GetMethod("LuckyHandler", _
                BindingFlags.NonPublic Or BindingFlags.Instance)
        ' Create an instance of the delegate. Using the overloads
        ' of CreateDelegate that take MethodInfo is recommended.
        '
        Dim d As [Delegate] = _
            [Delegate].CreateDelegate(tDelegate, Me, miHandler)

        ' Get the "add" accessor of the event and invoke it late-
        ' bound, passing in the delegate instance. This is equivalent
        ' to using the += operator in C#, or AddHandler in Visual
        ' Basic. The instance on which the "add" accessor is invoked
        ' is the form; the arguments must be passed as an array.
        '
        Dim miAddHandler As MethodInfo = evClick.GetAddMethod()
        Dim addHandlerArgs() As Object = { d }
        miAddHandler.Invoke(exFormAsObj, addHandlerArgs)

        ' Event handler methods can also be generated at run time,
        ' using lightweight dynamic methods and Reflection.Emit.
        ' To construct an event handler, you need the return type
        ' and parameter types of the delegate. These can be obtained
        ' by examining the delegate's Invoke method.
        '
        ' It is not necessary to name dynamic methods, so the empty
        ' string can be used. The last argument associates the
        ' dynamic method with the current type, giving the delegate
        ' access to all the public and private members of Example,
        ' as if it were an instance method.
        '
        Dim returnType As Type = GetDelegateReturnType(tDelegate)
        If returnType IsNot GetType(Void) Then
            Throw New ApplicationException("Delegate has a return type.")
        End If

        Dim handler As New DynamicMethod( _
            "", _
            Nothing, _
            GetDelegateParameterTypes(tDelegate), _
            GetType(Example) _
        )

        ' Generate a method body. This method loads a string, calls
        ' the Show method overload that takes a string, pops the
        ' return value off the stack (because the handler has no
        ' return type), and returns.
        '
        Dim ilgen As ILGenerator = handler.GetILGenerator()

        Dim showParameters As Type() = { GetType(String) }
        Dim simpleShow As MethodInfo = _
```

```
            GetType(MessageBox).GetMethod("Show", showParameters)

        ilgen.Emit(OpCodes.Ldstr, _
            "This event handler was constructed at run time.")
        ilgen.Emit(OpCodes.Call, simpleShow)
        ilgen.Emit(OpCodes.Pop)
        ilgen.Emit(OpCodes.Ret)

        ' Complete the dynamic method by calling its CreateDelegate
        ' method. Use the "add" accessor to add the delegate to
        ' the invocation list for the event.
        '
        Dim dEmitted As [Delegate] = handler.CreateDelegate(tDelegate)
        miAddHandler.Invoke(exFormAsObj, New Object() { dEmitted })

        ' Show the form. Clicking on the form causes the two
        ' delegates to be invoked.
        '
        Application.Run(CType(exFormAsObj, Form))

    End Sub

    Private Sub LuckyHandler(ByVal sender As [Object], _
        ByVal e As EventArgs)

        MessageBox.Show("This event handler just happened to be lying around.")
    End Sub

    Private Function GetDelegateParameterTypes(ByVal d As Type) _
        As Type()

        If d.BaseType IsNot GetType(MulticastDelegate) Then
            Throw New ApplicationException("Not a delegate.")
        End If

        Dim invoke As MethodInfo = d.GetMethod("Invoke")
        If invoke Is Nothing Then
            Throw New ApplicationException("Not a delegate.")
        End If

        Dim parameters As ParameterInfo() = invoke.GetParameters()
        ' Dimension this array Length - 1, because VB adds an extra
        ' element to zero-based arrays.
        Dim typeParameters(parameters.Length - 1) As Type
        For i As Integer = 0 To parameters.Length - 1
            typeParameters(i) = parameters(i).ParameterType
        Next i

        Return typeParameters

    End Function


    Private Function GetDelegateReturnType(ByVal d As Type) As Type
```

```
        If d.BaseType IsNot GetType(MulticastDelegate) Then
            Throw New ApplicationException("Not a delegate.")
        End If

        Dim invoke As MethodInfo = d.GetMethod("Invoke")
        If invoke Is Nothing Then
            Throw New ApplicationException("Not a delegate.")
        End If

        Return invoke.ReturnType

    End Function
End Class
```

Compiling the Code

- The code contains the C# **using** statements (**Imports** in Visual Basic) necessary for compilation.
- No additional assembly references are required for compiling from the command line. In Visual Studio you must add a reference to System.Windows.Forms.dll because this example is a console application.
- Compile the code at the command line using csc.exe, vbc.exe, or cl.exe. To compile the code in Visual Studio, place it in a console application project template.

See Also

Assembly.Load
DynamicMethod
CreateInstance
CreateDelegate
How to: Define and Execute Dynamic Methods
Reflection in the .NET Framework

Reflection in the .NET Framework for Windows Store Apps

.NET Framework (current version)

Starting with the .NET Framework 4.5, the .NET Framework includes a set of reflection types and members for use in Windows 8.x Store apps. These types and members are available in the full .NET Framework as well as in the .NET for Windows Store apps. This document explains the major differences between these and their counterparts in the .NET Framework 4 and earlier versions.

If you are creating a Windows 8.x Store app, you must use the reflection types and members in the .NET for Windows 8.x Store apps. These types and members are also available, but not required, for use in desktop apps, so you can use the same code for both types of apps.

TypeInfo and Assembly Loading

In the .NET for Windows 8.x Store apps, the TypeInfo class contains some of the functionality of the .NET Framework 4 Type class. A Type object represents a reference to a type definition, whereas a TypeInfo object represents the type definition itself. This enables you to manipulate Type objects without necessarily requiring the runtime to load the assembly they reference. Getting the associated TypeInfo object forces the assembly to load.

TypeInfo contains many of the members available on Type, and many of the reflection properties in the .NET for Windows 8.x Store apps return collections of TypeInfo objects. To get a TypeInfo object from a Type object, use the GetTypeInfo method.

Query Methods

In the .NET for Windows 8.x Store apps, you use the reflection properties that return IEnumerable(Of T) collections instead of methods that return arrays. Reflection contexts can implement lazy traversal of these collections for large assemblies or types.

The reflection properties return only the declared methods on a particular object instead of traversing the inheritance tree. Moreover, they do not use BindingFlags parameters for filtering. Instead, filtering takes place in user code, by using LINQ queries on the returned collections. For reflection objects that originate with the runtime (for example, as the result of `typeof(Object)`), traversing the inheritance tree is best accomplished by using the helper methods of the RuntimeReflectionExtensions class. Consumers of objects from customized reflection contexts cannot use these methods, and must traverse the inheritance tree themselves.

Restrictions

In a Windows 8.x Store app, access to some .NET Framework types and members is restricted. For example, you cannot call .NET Framework methods that are not included in .NET for Windows 8.x Store apps, by using a MethodInfo object. In addition, certain types and members that are not considered safe within the context of a Windows 8.x Store app are blocked, as are Marshal and WindowsRuntimeMarshal members. This restriction affects only .NET Framework types and members; you can call your code or third-party code as you normally would.

Example

This example uses the reflection types and members in the .NET for Windows 8.x Store apps to retrieve the methods and properties of the Calendar type, including inherited methods and properties. To run this code, paste it into the code file for a Windows 8.x Store page that contains a Windows.UI.Xaml.Controls.Textblock control named `textblock1` in a project named Reflection. If you paste this code inside a project with a different name, just make sure you change the namespace name to match your project.

VB

```vb
Imports Windows.UI.Xaml.Navigation
Imports System.Reflection
Imports System.Globalization
Imports System.Text
Imports System

Public NotInheritable Class MainPage
    Inherits Page

    Protected Overrides Sub OnNavigatedTo(e As NavigationEventArgs)
        Dim t As TypeInfo = GetType(Calendar).GetTypeInfo()
        Dim pList As IEnumerable(Of PropertyInfo) = t.DeclaredProperties
        Dim mList As IEnumerable(Of MethodInfo) = t.DeclaredMethods

        Dim sb As New StringBuilder()

        sb.Append("Properties:")
        For Each p As PropertyInfo In pList

            sb.Append((vbLf + p.DeclaringType.Name & ": ") + p.Name)
        Next
        sb.Append(vbLf & "Methods:")
        For Each m As MethodInfo In mList
            sb.Append((vbLf + m.DeclaringType.Name & ": ") + m.Name)
        Next

        textblock1.Text = sb.ToString()

    End Sub
End Class
```

See Also

Reflection in the .NET Framework

.NET for Windows Store apps – supported APIs

Emitting Dynamic Methods and Assemblies

.NET Framework (current version)

This section describes a set of managed types in the System.Reflection.Emit namespace that allow a compiler or tool to emit metadata and Microsoft intermediate language (MSIL) at run time and optionally generate a portable executable (PE) file on disk. Script engines and compilers are the primary users of this namespace. In this section, the functionality provided by the System.Reflection.Emit namespace is referred to as reflection emit.

Reflection emit provides the following capabilities:

- Define lightweight global methods at run time, using the DynamicMethod class, and execute them using delegates.
- Define assemblies at run time and then run them and/or save them to disk.
- Define assemblies at run time, run them, and then unload them and allow garbage collection to reclaim their resources.
- Define modules in new assemblies at run time and then run and/or save them to disk.
- Define types in modules at run time, create instances of these types, and invoke their methods.
- Define symbolic information for defined modules that can be used by tools such as debuggers and code profilers.

In addition to the managed types in the System.Reflection.Emit namespace, there are unmanaged metadata interfaces which are described in the Metadata Interfaces reference documentation. Managed reflection emit provides stronger semantic error checking and a higher level of abstraction of the metadata than the unmanaged metadata interfaces.

Another useful resource for working with metadata and MSIL is the Common Language Infrastructure (CLI) documentation, especially "Partition II: Metadata Definition and Semantics" and "Partition III: CIL Instruction Set". The documentation is available online on MSDN and at the Ecma Web site.

In This Section

Security Issues in Reflection Emit
: Describes security issues related to creating dynamic assemblies using reflection emit.

Reference

OpCodes
: Catalogs the MSIL instruction codes you can use to build method bodies.

System.Reflection.Emit
: Contains managed classes used to emit dynamic methods, assemblies, and types.

Type
: Describes the Type class, which represents types in managed reflection and reflection emit, and which is key to the use

of these technologies.

System.Reflection
: Contains managed classes used to explore metadata and managed code.

Related Sections

Reflection in the .NET Framework
: Explains how to explore metadata and managed code.

Assemblies in the Common Language Runtime
: Provides an overview of assemblies in the .NET Framework.

Security Issues in Reflection Emit

.NET Framework (current version)

The .NET Framework provides three ways to emit Microsoft intermediate language (MSIL), each with its own security issues:

- Dynamic assemblies
- Anonymously hosted dynamic methods
- Dynamic methods associated with existing assemblies

Regardless of the way you generate dynamic code, executing the generated code requires all the permissions that are required by the types and methods the generated code uses.

Note

The permissions that are required for reflecting on code and emitting code have changed with succeeding releases of the .NET Framework. See Version Information, later in this topic.

Dynamic Assemblies

Dynamic assemblies are created by using overloads of the AppDomain.DefineDynamicAssembly method. Most overloads of this method are deprecated in the .NET Framework 4, because of the elimination of machine-wide security policy. (See Security Changes in the .NET Framework.) The remaining overloads can be executed by any code, regardless of trust level. These overloads fall into two groups: those that specify a list of attributes to apply to the dynamic assembly when it is created, and those that do not. If you do not specify the transparency model for the assembly, by applying the SecurityRulesAttribute attribute when you create it, the transparency model is inherited from the emitting assembly.

Note

Attributes that you apply to the dynamic assembly after it is created, by using the SetCustomAttribute method, do not take effect until the assembly has been saved to disk and loaded into memory again.

Code in a dynamic assembly can access visible types and members in other assemblies.

Note

Dynamic assemblies do not use the ReflectionPermissionFlag.MemberAccess and ReflectionPermissionFlag.RestrictedMemberAccess flags that allow dynamic methods to access nonpublic types and members.

Transient dynamic assemblies are created in memory and never saved to disk, so they require no file access permissions. Saving a dynamic assembly to disk requires FileIOPermission with the appropriate flags.

Generating Dynamic Assemblies from Partially Trusted Code

Consider the conditions in which an assembly with Internet permissions can generate a transient dynamic assembly and execute its code:

- The dynamic assembly uses only public types and members of other assemblies.
- The permissions demanded by those types and members are included in the grant set of the partially trusted assembly.
- The assembly is not saved to disk.
- Debug symbols are not generated. (**Internet** and **LocalIntranet** permission sets do not include the necessary permissions.)

Anonymously Hosted Dynamic Methods

Anonymously hosted dynamic methods are created by using the two DynamicMethod constructors that do not specify an associated type or module, DynamicMethod(String, Type, Type()) and DynamicMethod(String, Type, Type(), Boolean). These constructors place the dynamic methods in a system-provided, fully trusted, security-transparent assembly. No permissions are required to use these constructors or to emit code for the dynamic methods.

Instead, when an anonymously hosted dynamic method is created, the call stack is captured. When the method is constructed, security demands are made against the captured call stack.

Note

Conceptually, demands are made during the construction of the method. That is, demands could be made as each MSIL instruction is emitted. In the current implementation, all demands are made when the DynamicMethod.CreateDelegate method is called or when the just-in-time (JIT) compiler is invoked, if the method is invoked without calling CreateDelegate.

If the application domain permits it, anonymously hosted dynamic methods can skip JIT visibility checks, subject to the following restriction: The nonpublic types and members accessed by an anonymously hosted dynamic method must be in assemblies whose grant sets are equal to, or subsets of, the grant set of the emitting call stack. This restricted ability to skip JIT visibility checks is enabled if the application domain grants ReflectionPermission with the ReflectionPermissionFlag.RestrictedMemberAccess flag.

- If your method uses only public types and members, no permissions are required during construction.

- If you specify that JIT visibility checks should be skipped, the demand that is made when the method is constructed includes ReflectionPermission with the ReflectionPermissionFlag.RestrictedMemberAccess flag and the grant set of the assembly that contains the nonpublic member that is being accessed.

Because the grant set of the nonpublic member is taken into consideration, partially trusted code that has been granted ReflectionPermissionFlag.RestrictedMemberAccess cannot elevate its privileges by executing nonpublic members of trusted assemblies.

As with any other emitted code, executing the dynamic method requires whatever permissions are demanded by the methods the dynamic method uses.

The system assembly that hosts anonymously-hosted dynamic methods uses the SecurityRuleSet.Level1 transparency model, which is the transparency model that was used in the .NET Framework before the .NET Framework 4.

For more information, see the DynamicMethod class.

Generating Anonymously Hosted Dynamic Methods from Partially Trusted Code

Consider the conditions in which an assembly with Internet permissions can generate an anonymously hosted dynamic method and execute it:

- The dynamic method uses only public types and members. If its grant set includes ReflectionPermissionFlag.RestrictedMemberAccess, it can use nonpublic types and members of any assembly whose grant set is equal to, or a subset of, the grant set of the emitting assembly.

- The permissions that are required by all the types and members used by the dynamic method are included in the grant set of the partially trusted assembly.

Note

Dynamic methods do not support debug symbols.

Dynamic Methods Associated with Existing Assemblies

To associate a dynamic method with a type or module in an existing assembly, use any of the DynamicMethod constructors that specify the associated type or module. The permissions that are required to call these constructors vary, because associating a dynamic method with an existing type or module gives the dynamic method access to nonpublic types and members:

- A dynamic method that is associated with a type has access to all members of that type, even private members, and to all internal types and members in the assembly that contains the associated type.

- A dynamic method that is associated with a module has access to all the **internal** types and members (**Friend** in Visual Basic, **assembly** in common language runtime metadata) in the module.

In addition, you can use a constructor that specifies the ability to skip the visibility checks of the JIT compiler. Doing so gives your dynamic method access to all types and members in all assemblies, regardless of access level.

The permissions demanded by the constructor depend on how much access you decide to give your dynamic method:

- If your method uses only public types and members, and you associate it with your own type or your own module, no permissions are required.
- If you specify that JIT visibility checks should be skipped, the constructor demands ReflectionPermission with the ReflectionPermissionFlag.MemberAccess flag.
- If you associate the dynamic method with another type, even another type in your own assembly, the constructor demands ReflectionPermission with the ReflectionPermissionFlag.MemberAccess flag and SecurityPermission with the SecurityPermissionFlag.ControlEvidence flag.
- If you associate the dynamic method with a type or module in another assembly, the constructor demands two things: ReflectionPermission with the ReflectionPermissionFlag.RestrictedMemberAccess flag, and the grant set of the assembly that contains the other module. That is, your call stack must include all the permissions in the grant set of the target module, plus ReflectionPermissionFlag.RestrictedMemberAccess.

> **Note**
>
> For backward compatibility, if the demand for the target grant set plus ReflectionPermissionFlag.RestrictedMemberAccess fails, the constructor demands SecurityPermission with the SecurityPermissionFlag.ControlEvidence flag.

Although the items in this list are described in terms of the grant set of the emitting assembly, remember that the demands are made against the full call stack, including the application domain boundary.

For more information, see the DynamicMethod class.

Generating Dynamic Methods from Partially Trusted Code

> **Note**
>
> The recommended way to generate dynamic methods from partially trusted code is to use anonymously hosted dynamic methods.

Consider the conditions in which an assembly with Internet permissions can generate a dynamic method and execute it:

- Either the dynamic method is associated with the module or type that emits it, or its grant set includes ReflectionPermissionFlag.RestrictedMemberAccess and it is associated with a module in an assembly whose grant set is equal to, or a subset of, the grant set of the emitting assembly.
- The dynamic method uses only public types and members. If its grant set includes ReflectionPermissionFlag.RestrictedMemberAccess and it is associated with a module in an assembly whose grant set is equal to, or a subset of, the grant set of the emitting assembly, it can use types and members marked

internal (**Friend** in Visual Basic, **assembly** in common language runtime metadata) in the associated module.

- The permissions demanded by all the types and members used by the dynamic method are included in the grant set of the partially trusted assembly.
- The dynamic method does not skip JIT visibility checks.

Note

Dynamic methods do not support debug symbols.

Version Information

Starting with the .NET Framework 4, machine-wide security policy is eliminated and security transparency becomes the default enforcement mechanism. See Security Changes in the .NET Framework.

Starting with the .NET Framework 2.0 Service Pack 1, ReflectionPermission with the ReflectionPermissionFlag.ReflectionEmit flag is no longer required when emitting dynamic assemblies and dynamic methods. This flag is required in all earlier versions of the .NET Framework.

Note

ReflectionPermission with the ReflectionPermissionFlag.ReflectionEmit flag is included by default in the **FullTrust** and **LocalIntranet** named permission sets, but not in the **Internet** permission set. Therefore, in earlier versions of the .NET Framework, a library can be used with Internet permissions only if it executes an Assert for ReflectionEmit. Such libraries require careful security review because coding errors could result in security holes. The .NET Framework 2.0 SP1 allows code to be emitted in partial trust scenarios without issuing any security demands, because generating code is not inherently a privileged operation. That is, the generated code has no more permissions than the assembly that emits it. This allows libraries that emit code to be security transparent and removes the need to assert ReflectionEmit, which simplifies the task of writing a secure library.

In addition, the .NET Framework 2.0 SP1 introduces the ReflectionPermissionFlag.RestrictedMemberAccess flag for accessing nonpublic types and members from partially trusted dynamic methods. Earlier versions of the .NET Framework require the ReflectionPermissionFlag.MemberAccess flag for dynamic methods that access nonpublic types and members; this is a permission that should never be granted to partially trusted code.

Finally, the .NET Framework 2.0 SP1 introduces anonymously hosted methods.

Obtaining Information on Types and Members

Starting with the .NET Framework 2.0, no permissions are required to obtain information about nonpublic types and members. Reflection is used to obtain information needed to emit dynamic methods. For example, MethodInfo objects are used to emit method calls. Earlier versions of the .NET Framework require ReflectionPermission with the ReflectionPermissionFlag.TypeInformation flag. For more information, see Security Considerations for Reflection.

See Also

Security Considerations for Reflection
Emitting Dynamic Methods and Assemblies

Walkthrough: Emitting Code in Partial Trust Scenarios

.NET Framework (current version)

Reflection emit uses the same API set in full or partial trust, but some features require special permissions in partially trusted code. In addition, reflection emit has a feature, anonymously hosted dynamic methods, that is designed to be used with partial trust and by security-transparent assemblies.

Note

Before .NET Framework 3.5, emitting code required ReflectionPermission with the ReflectionPermissionFlag.ReflectionEmit flag. This permission is included by default in the **FullTrust** and **Intranet** named permission sets, but not in the **Internet** permission set. Therefore, a library could be used from partial trust only if it had the SecurityCriticalAttribute attribute and also executed an Assert method for ReflectionEmit. Such libraries require careful security review because coding errors could result in security holes. The .NET Framework 3.5 allows code to be emitted in partial trust scenarios without issuing any security demands, because generating code is not inherently a privileged operation. That is, the generated code has no more permissions than the assembly that emits it. This enables libraries that emit code to be security-transparent and removes the need to assert ReflectionEmit, so that writing a secure library does not require such a thorough security review.

This walkthrough illustrates the following tasks:

- Setting up a simple sandbox for testing partially trusted code.

  Important

  This is a simple way to experiment with code in partial trust. To run code that actually comes from untrusted locations, see How to: Run Partially Trusted Code in a Sandbox.

- Running code in partially trusted application domains.
- Using anonymously hosted dynamic methods to emit and execute code in partial trust.

For more information about emitting code in partial trust scenarios, see Security Issues in Reflection Emit.

For a complete listing of the code shown in these procedures, see the Example section at the end of this walkthrough.

Setting up Partially Trusted Locations

The following two procedures show how to set up locations from which you can test code with partial trust.

- The first procedure shows how to create a sandboxed application domain in which code is granted Internet permissions.
- The second procedure shows how to add ReflectionPermission with the ReflectionPermissionFlag.RestrictedMemberAccess flag to a partially trusted application domain, to enable access to private data in assemblies of equal or lesser trust.

Creating Sandboxed Application Domains

To create an application domain in which your assemblies run with partial trust, you must specify the set of permissions to be granted to the assemblies by using the AppDomain.CreateDomain(String, Evidence, AppDomainSetup, PermissionSet, StrongName()) method overload to create the application domain. The easiest way to specify the grant set is to retrieve a named permission set from security policy.

The following procedure creates a sandboxed application domain that runs your code with partial trust, to test scenarios in which emitted code can access only public members of public types. A subsequent procedure shows how to add RestrictedMemberAccess, to test scenarios in which emitted code can access nonpublic types and members in assemblies that are granted equal or lesser permissions.

To create an application domain with partial trust

1. Create a permission set to grant to the assemblies in the sandboxed application domain. In this case, the permission set of the Internet zone is used.

VB

```
Dim ev As New Evidence()
ev.AddHostEvidence(new Zone(SecurityZone.Internet))
Dim pset As New NamedPermissionSet("Internet",
SecurityManager.GetStandardSandbox(ev))
```

2. Create an AppDomainSetup object to initialize the application domain with an application path.

Important

For simplicity, this code example uses the current folder. To run code that actually comes from the Internet, use a separate folder for the untrusted code, as described in How to: Run Partially Trusted Code in a Sandbox.

VB

```
Dim adSetup As New AppDomainSetup()
adSetup.ApplicationBase = "."
```

3. Create the application domain, specifying the application domain setup information and the grant set for all assemblies that execute in the application domain.

VB

```
Dim ad As AppDomain = AppDomain.CreateDomain("Sandbox", ev, adSetup, pset,
Nothing)
```

The last parameter of the AppDomain.CreateDomain(String, Evidence, AppDomainSetup, PermissionSet, StrongName()) method overload enables you to specify a set of assemblies that are to be granted full trust, instead of the grant set of the application domain. You do not have to specify the .NET Framework assemblies that your application uses, because those assemblies are in the global assembly cache. Assemblies in the global assembly cache are always fully trusted. You can use this parameter to specify strong-named assemblies that are not in the global assembly cache.

Adding RestrictedMemberAccess to Sandboxed Domains

Host applications can allow anonymously hosted dynamic methods to have access to private data in assemblies that have trust levels equal to or less than the trust level of the assembly that emits the code. To enable this restricted ability to skip just-in-time (JIT) visibility checks, the host application adds a ReflectionPermission object with the ReflectionPermissionFlag.RestrictedMemberAccess (RMA) flag to the grant set.

For example, a host might grant Internet applications Internet permissions plus RMA, so that an Internet application can emit code that accesses private data in its own assemblies. Because the access is limited to assemblies of equal or lesser trust, an Internet application cannot access members of fully trusted assemblies such as .NET Framework assemblies.

Note

To prevent elevation of privilege, stack information for the emitting assembly is included when anonymously hosted dynamic methods are constructed. When the method is invoked, the stack information is checked. Thus, an anonymously hosted dynamic method that is invoked from fully trusted code is still limited to the trust level of the emitting assembly.

To create an application domain with partial trust plus RMA

1. Create a new ReflectionPermission object with the RestrictedMemberAccess (RMA) flag, and use the PermissionSet.SetPermission method to add the permission to the grant set.

VB

```
pset.SetPermission( _
    New ReflectionPermission( _
        ReflectionPermissionFlag.RestrictedMemberAccess))
```

The AddPermission method adds the permission to the grant set if it is not already included. If the permission is already included in the grant set, the specified flags are added to the existing permission.

Note

RMA is a feature of anonymously hosted dynamic methods. When ordinary dynamic methods skip JIT visibility checks, the emitted code requires full trust.

2. Create the application domain, specifying the application domain setup information and the grant set.

VB

```
ad = AppDomain.CreateDomain("Sandbox2", ev, adSetup, pset, Nothing)
```

Running Code in Sandboxed Application Domains

The following procedure explains how to define a class by using methods that can be executed in an application domain, how to create an instance of the class in the domain, and how to execute its methods.

To define and execute a method in an application domain

1. Define a class that derives from MarshalByRefObject. This enables you to create instances of the class in other application domains and to make method calls across application domain boundaries. The class in this example is named `Worker`.

VB

```
Public Class Worker
    Inherits MarshalByRefObject
```

2. Define a public method that contains the code you want to execute. In this example, the code emits a simple dynamic method, creates a delegate to execute the method, and invokes the delegate.

VB

```
Public Sub SimpleEmitDemo()

    Dim meth As DynamicMethod = new DynamicMethod("", Nothing, Nothing)
    Dim il As ILGenerator = meth.GetILGenerator()
    il.EmitWriteLine("Hello, World!")
    il.Emit(OpCodes.Ret)

    Dim t1 As Test1 = CType(meth.CreateDelegate(GetType(Test1)), Test1)
    t1()
End Sub
```

3. In your main program, get the display name of your assembly. This name is used when you create instances of the `Worker` class in the sandboxed application domain.

VB

```
Dim asmName As String = GetType(Worker).Assembly.FullName
```

4. In your main program, create a sandboxed application domain, as described in the first procedure in this walkthrough. You do not have to add any permissions to the **Internet** permission set, because the `SimpleEmitDemo` method uses only public methods.

5. In your main program, create an instance of the `Worker` class in the sandboxed application domain.

VB

```
Dim w As Worker = _
    CType(ad.CreateInstanceAndUnwrap(asmName, "Worker"), Worker)
```

The CreateInstanceAndUnwrap method creates the object in the target application domain and returns a proxy that can be used to call the properties and methods of the object.

Note

If you use this code in Visual Studio, you must change the name of the class to include the namespace. By default, the namespace is the name of the project. For example, if the project is "PartialTrust", the class name must be "PartialTrust.Worker".

6. Add code to call the `SimpleEmitDemo` method. The call is marshaled across the application domain boundary, and the code is executed in the sandboxed application domain.

VB

```
w.SimpleEmitDemo()
```

Using Anonymously Hosted Dynamic Methods

Anonymously hosted dynamic methods are associated with a transparent assembly that is provided by the system. Therefore, the code they contain is transparent. Ordinary dynamic methods, on the other hand, must be associated with an existing module (whether directly specified or inferred from an associated type), and take their security level from that module.

Note

The only way to associate a dynamic method with the assembly that provides anonymous hosting is to use the constructors that are described in the following procedure. You cannot explicitly specify a module in the anonymous hosting assembly.

Ordinary dynamic methods have access to the internal members of the module they are associated with, or to the private members of the type they are associated with. Because anonymously hosted dynamic methods are isolated from other code, they do not have access to private data. However, they do have a restricted ability to skip JIT visibility checks to gain access to private data. This ability is limited to assemblies that have trust levels equal to or less than the trust level of the assembly that emits the code.

To prevent elevation of privilege, stack information for the emitting assembly is included when anonymously hosted dynamic methods are constructed. When the method is invoked, the stack information is checked. An anonymously hosted dynamic method that is invoked from fully trusted code is still limited to the trust level of the assembly that emitted it.

To use anonymously hosted dynamic methods

- Create an anonymously hosted dynamic method by using a constructor that does not specify an associated module or type.

  VB

  ```
  Dim meth As DynamicMethod = new DynamicMethod("", Nothing, Nothing)
  Dim il As ILGenerator = meth.GetILGenerator()
  il.EmitWriteLine("Hello, World!")
  il.Emit(OpCodes.Ret)
  ```

  If an anonymously hosted dynamic method uses only public types and methods, it does not require restricted member access and does not have to skip JIT visibility checks.

  No special permissions are required to emit a dynamic method, but the emitted code requires the permissions that are demanded by the types and methods it uses. For example, if the emitted code calls a method that accesses a file, it requires FileIOPermission. If the trust level does not include that permission, a security exception is thrown when the emitted code is executed. The code shown here emits a dynamic method that uses only the Console.WriteLine method. Therefore, the code can be executed from partially trusted locations.

- Alternatively, create an anonymously hosted dynamic method with restricted ability to skip JIT visibility checks, by using the DynamicMethod(String, Type, Type(), Boolean) constructor and specifying **true** for the *restrictedSkipVisibility* parameter.

  VB

  ```
  Dim meth As New DynamicMethod("", _
                                 GetType(Char), _
                                 New Type() {GetType(String)}, _
                                 True)
  ```

  The restriction is that the anonymously hosted dynamic method can access private data only in assemblies with trust levels equal to or less than the trust level of the emitting assembly. For example, if the dynamic method is executing with Internet trust, it can access private data in other assemblies that are also executing with Internet trust, but it cannot access private data of .NET Framework assemblies. .NET Framework assemblies are installed in the global assembly cache and are always fully trusted.

  Anonymously hosted dynamic methods can use this restricted ability to skip JIT visibility checks only if the host application grants ReflectionPermission with the ReflectionPermissionFlag.RestrictedMemberAccess flag. The demand for this permission is made when the method is invoked.

Note

Call stack information for the emitting assembly is included when the dynamic method is constructed. Therefore, the demand is made against the permissions of the emitting assembly instead of the assembly that invokes the method. This prevents the emitted code from being executed with elevated permissions.

The complete code example at the end of this walkthrough demonstrates the use and limitations of restricted member access. Its `Worker` class includes a method that can create anonymously hosted dynamic methods with or without the restricted ability to skip visibility checks, and the example shows the result of executing this method in application domains that have different trust levels.

Note

The restricted ability to skip visibility checks is a feature of anonymously hosted dynamic methods. When ordinary dynamic methods skip JIT visibility checks, they must be granted full trust.

Example

Description

The following code example demonstrates the use of the RestrictedMemberAccess flag to allow anonymously hosted dynamic methods to skip JIT visibility checks, but only when the target member is at an equal or lower level of trust than the assembly that emits the code.

The example defines a `Worker` class that can be marshaled across application domain boundaries. The class has two `AccessPrivateMethod` method overloads that emit and execute dynamic methods. The first overload emits a dynamic method that calls the private `PrivateMethod` method of the `Worker` class, and it can emit the dynamic method with or without JIT visibility checks. The second overload emits a dynamic method that accesses an **internal** property (**Friend** property in Visual Basic) of the String class.

The example uses a helper method to create a grant set limited to **Internet** permissions, and then creates an application domain, using the AppDomain.CreateDomain(String, Evidence, AppDomainSetup, PermissionSet, StrongName()) method overload to specify that all code that executes in the domain uses this grant set. The example creates an instance of the `Worker` class in the application domain, and executes the `AccessPrivateMethod` method two times.

- The first time the `AccessPrivateMethod` method is executed, JIT visibility checks are enforced. The dynamic method fails when it is invoked, because JIT visibility checks prevent it from accessing the private method.
- The second time the `AccessPrivateMethod` method is executed, JIT visibility checks are skipped. The dynamic method fails when it is compiled, because the **Internet** grant set does not grant sufficient permissions to skip visibility checks.

The example adds ReflectionPermission with ReflectionPermissionFlag.RestrictedMemberAccess to the grant set. The example then creates a second domain, specifying that all code that executes in the domain is granted the permissions in the new grant set. The example creates an instance of the `Worker` class in the new application domain, and executes both overloads of the `AccessPrivateMethod` method.

- The first overload of the `AccessPrivateMethod` method is executed, and JIT visibility checks are skipped. The dynamic method compiles and executes successfully, because the assembly that emits the code is the same as the assembly that contains the private method. Therefore, the trust levels are equal. If the application that contains the `Worker` class had several assemblies, the same process would succeed for any one of those assemblies, because they would all be at the same trust level.
- The second overload of the `AccessPrivateMethod` method is executed, and again JIT visibility checks are skipped. This time the dynamic method fails when it is compiled, because it tries to access the **internal** `FirstChar` property of the String class. The assembly that contains the String class is fully trusted. Therefore, it is at a higher level of trust than the assembly that emits the code.

This comparison shows how ReflectionPermissionFlag.RestrictedMemberAccess enables partially trusted code to skip visibility checks for other partially trusted code without compromising the security of trusted code.

Code

VB

```
Imports System.Reflection.Emit
Imports System.Reflection
Imports System.Security
Imports System.Security.Permissions
Imports System.Security.Policy
Imports System.Collections
Imports System.Diagnostics

' This code example works properly only if it is run from a fully
' trusted location, such as your local computer.

' Delegates used to execute the dynamic methods.
'
Public Delegate Sub Test(ByVal w As Worker)
Public Delegate Sub Test1()
Public Delegate Function Test2(ByVal instance As String) As Char

' The Worker class must inherit MarshalByRefObject so that its public
' methods can be invoked across application domain boundaries.
'
Public Class Worker
    Inherits MarshalByRefObject

    Private Sub PrivateMethod()
        Console.WriteLine("Worker.PrivateMethod()")
    End Sub

    Public Sub SimpleEmitDemo()
```

```
        Dim meth As DynamicMethod = new DynamicMethod("", Nothing, Nothing)
        Dim il As ILGenerator = meth.GetILGenerator()
        il.EmitWriteLine("Hello, World!")
        il.Emit(OpCodes.Ret)

        Dim t1 As Test1 = CType(meth.CreateDelegate(GetType(Test1)), Test1)
        t1()
    End Sub

    ' This overload of AccessPrivateMethod emits a dynamic method and
    ' specifies whether to skip JIT visiblity checks. It creates a
    ' delegate for the method and invokes the delegate. The dynamic
    ' method calls a private method of the Worker class.
    Overloads Public Sub AccessPrivateMethod( _
                       ByVal restrictedSkipVisibility As Boolean)

        ' Create an unnamed dynamic method that has no return type,
        ' takes one parameter of type Worker, and optionally skips JIT
        ' visiblity checks.
        Dim meth As New DynamicMethod("", _
                                      Nothing, _
                                      New Type() { GetType(Worker) }, _
                                      restrictedSkipVisibility)

        ' Get a MethodInfo for the private method.
        Dim pvtMeth As MethodInfo = GetType(Worker).GetMethod( _
            "PrivateMethod", _
            BindingFlags.NonPublic Or BindingFlags.Instance)

        ' Get an ILGenerator and emit a body for the dynamic method.
        Dim il As ILGenerator = meth.GetILGenerator()

        ' Load the first argument, which is the target instance, onto the
        ' execution stack, call the private method, and return.
        il.Emit(OpCodes.Ldarg_0)
        il.EmitCall(OpCodes.Call, pvtMeth, Nothing)
        il.Emit(OpCodes.Ret)

        ' Create a delegate that represents the dynamic method, and
        ' invoke it.
        Try
            Dim t As Test = CType(meth.CreateDelegate(GetType(Test)), Test)
            Try
                t(Me)
            Catch ex As Exception
                Console.WriteLine("{0} was thrown when the delegate was invoked.", _
                    ex.GetType().Name)
            End Try
        Catch ex As Exception
            Console.WriteLine("{0} was thrown when the delegate was compiled.", _
                ex.GetType().Name)
        End Try
    End Sub


    ' This overload of AccessPrivateMethod emits a dynamic method that takes
    ' a string and returns the first character, using a private field of the
    ' String class. The dynamic method skips JIT visiblity checks.
    Overloads Public Sub AccessPrivateMethod()

        Dim meth As New DynamicMethod("", _
                                      GetType(Char), _
                                      New Type() {GetType(String)}, _
                                      True)

        ' Get a MethodInfo for the 'get' accessor of the private property.
        Dim pi As PropertyInfo = GetType(String).GetProperty( _
            "FirstChar", _
            BindingFlags.NonPublic Or BindingFlags.Instance)
        Dim pvtMeth As MethodInfo = pi.GetGetMethod(True)

        ' Get an ILGenerator and emit a body for the dynamic method.
        Dim il As ILGenerator = meth.GetILGenerator()

        ' Load the first argument, which is the target string, onto the
        ' execution stack, call the 'get' accessor to put the result onto
        ' the execution stack, and return.
        il.Emit(OpCodes.Ldarg_0)
        il.EmitCall(OpCodes.Call, pvtMeth, Nothing)
        il.Emit(OpCodes.Ret)

        ' Create a delegate that represents the dynamic method, and
        ' invoke it.
        Try
            Dim t As Test2 = CType(meth.CreateDelegate(GetType(Test2)), Test2)
            Dim first As Char = t("Hello, World!")
            Console.WriteLine("{0} is the first character.", first)
        Catch ex As Exception
            Console.WriteLine("{0} was thrown when the delegate was compiled.", _
                ex.GetType().Name)
        End Try

    End Sub
End Class

Friend Class Example

    ' The entry point for the code example.
    Shared Sub Main()

        ' Get the display name of the executing assembly, to use when
        ' creating objects to run code in application domains.
        Dim asmName As String = GetType(Worker).Assembly.FullName

        ' Create the permission set to grant to other assemblies. In this
        ' case they are the permissions found in the Internet zone.
        Dim ev As New Evidence()
        ev.AddHostEvidence(new Zone(SecurityZone.Internet))
        Dim pset As New NamedPermissionSet("Internet",
SecurityManager.GetStandardSandbox(ev))

        ' For simplicity, set up the application domain to use the
        ' current path as the application folder, so the same executable
        ' can be used in both trusted and untrusted scenarios. Normally
        ' you would not do this with real untrusted code.
        Dim adSetup As New AppDomainSetup()
        adSetup.ApplicationBase = "."

        ' Create an application domain in which all code that executes is
        ' granted the permissions of an application run from the Internet.
        Dim ad As AppDomain = AppDomain.CreateDomain("Sandbox", ev, adSetup, pset,
Nothing)

        ' Create an instance of the Worker class in the partially trusted
        ' domain. Note: If you build this code example in Visual Studio,
        ' you must change the name of the class to include the default
        ' namespace, which is the project name. For example, if the project
        ' is "AnonymouslyHosted", the class is "AnonymouslyHosted.Worker".
        Dim w As Worker = _
            CType(ad.CreateInstanceAndUnwrap(asmName, "Worker"), Worker)

        ' Emit a simple dynamic method that prints "Hello, World!"
        w.SimpleEmitDemo()

        ' Emit and invoke a dynamic method that calls a private method
        ' of Worker, with JIT visibility checks enforced. The call fails
        ' when the delegate is invoked.
        w.AccessPrivateMethod(False)

        ' Emit and invoke a dynamic method that calls a private method
        ' of Worker, skipping JIT visibility checks. The call fails when
        ' the method is compiled.
        w.AccessPrivateMethod(True)


        ' Unload the application domain. Add RestrictedMemberAccess to the
        ' grant set, and use it to create an application domain in which
        ' partially trusted code can call private members, as long as the
        ' trust level of those members is equal to or lower than the trust
        ' level of the partially trusted code.
        AppDomain.Unload(ad)
        pset.SetPermission( _
            New ReflectionPermission( _
                ReflectionPermissionFlag.RestrictedMemberAccess))
        ad = AppDomain.CreateDomain("Sandbox2", ev, adSetup, pset, Nothing)

        ' Create an instance of the Worker class in the partially trusted
        ' domain.
        w = CType(ad.CreateInstanceAndUnwrap(asmName, "Worker"), Worker)
```

```
            ' Again, emit and invoke a dynamic method that calls a private method
            ' of Worker, skipping JIT visibility checks. This time compilation
            ' succeeds because of the grant for RestrictedMemberAccess.
            w.AccessPrivateMethod(True)

            ' Finally, emit and invoke a dynamic method that calls an internal
            ' method of the String class. The call fails, because the trust level
            ' of the assembly that contains String is higher than the trust level
            ' of the assembly that emits the dynamic method.
            w.AccessPrivateMethod()

        End Sub
    End Class

    ' This code example produces the following output:
    '
    'Hello, World!
    'MethodAccessException was thrown when the delegate was invoked.
    'MethodAccessException was thrown when the delegate was invoked.
    'Worker.PrivateMethod()
    'MethodAccessException was thrown when the delegate was compiled.
    '
```

Compiling the Code

- If you build this code example in Visual Studio, you must change the name of the class to include the namespace when you pass it to the CreateInstanceAndUnwrap method. By default, the namespace is the name of the project. For example, if the project is "PartialTrust", the class name must be "PartialTrust.Worker".

See Also

Security Issues in Reflection Emit
How to: Run Partially Trusted Code in a Sandbox

How to: Define and Execute Dynamic Methods

.NET Framework (current version)

The following procedures show how to define and execute a simple dynamic method and a dynamic method bound to an instance of a class. For more information on dynamic methods, see the DynamicMethod class and Reflection Emit Dynamic Method Scenarios.

To define and execute a dynamic method

1. Declare a delegate type to execute the method. Consider using a generic delegate to minimize the number of delegate types you need to declare. The following code declares two delegate types that could be used for the `SquareIt` method, and one of them is generic.

VB

```
Private Delegate Function _
    SquareItInvoker(ByVal input As Integer) As Long

Private Delegate Function _
    OneParameter(Of TReturn, TParameter0) _
    (ByVal p0 As TParameter0) As TReturn
```

2. Create an array that specifies the parameter types for the dynamic method. In this example, the only parameter is an **int** (**Integer** in Visual Basic), so the array has only one element.

VB

```
Dim methodArgs As Type() = { GetType(Integer) }
```

3. Create a DynamicMethod. In this example the method is named `SquareIt`.

Note

It is not necessary to give dynamic methods names, and they cannot be invoked by name. Multiple dynamic methods can have the same name. However, the name appears in call stacks and can be useful for debugging.

The type of the return value is specified as **long**. The method is associated with the module that contains the `Example` class, which contains the example code. Any loaded module could be specified. The dynamic method acts like a module-level **static** method (**Shared** in Visual Basic).

VB

```
Dim squareIt As New DynamicMethod( _
    "SquareIt", _
    GetType(Long), _
    methodArgs, _
    GetType(Example).Module)
```

4. Emit the method body. In this example, an ILGenerator object is used to emit the Microsoft intermediate language (MSIL). Alternatively, a DynamicILInfo object can be used in conjunction with unmanaged code generators to emit the method body for a DynamicMethod.

   The MSIL in this example loads the argument, which is an **int**, onto the stack, converts it to a **long**, duplicates the **long**, and multiplies the two numbers. This leaves the squared result on the stack, and all the method has to do is return.

   VB

```
Dim il As ILGenerator = squareIt.GetILGenerator()
il.Emit(OpCodes.Ldarg_0)
il.Emit(OpCodes.Conv_I8)
il.Emit(OpCodes.Dup)
il.Emit(OpCodes.Mul)
il.Emit(OpCodes.Ret)
```

5. Create an instance of the delegate (declared in step 1) that represents the dynamic method by calling the CreateDelegate method. Creating the delegate completes the method, and any further attempts to change the method — for example, adding more MSIL — are ignored. The following code creates the delegate and invokes it, using a generic delegate.

   VB

```
Dim invokeSquareIt As OneParameter(Of Long, Integer) = _
    CType( _
        squareIt.CreateDelegate( _
            GetType(OneParameter(Of Long, Integer))), _
        OneParameter(Of Long, Integer) _
    )

Console.WriteLine("123456789 squared = {0}", _
    invokeSquareIt(123456789))
```

To define and execute a dynamic method that is bound to an object

1. Declare a delegate type to execute the method. Consider using a generic delegate to minimize the number of delegate types you need to declare. The following code declares a generic delegate type that can be used to execute any method with one parameter and a return value, or a method with two parameters and a return value if the delegate is bound to an object.

   VB

```
Private Delegate Function _
    OneParameter(Of TReturn, TParameter0) _
    (ByVal p0 As TParameter0) As TReturn
```

2. Create an array that specifies the parameter types for the dynamic method. If the delegate representing the method is to be bound to an object, the first parameter must match the type the delegate is bound to. In this example, there are two parameters, of type `Example` and type **int** (**Integer** in Visual Basic).

VB

```
Dim methodArgs2 As Type() = _
    { GetType(Example), GetType(Integer) }
```

3. Create a DynamicMethod. In this example the method has no name. The type of the return value is specified as **int** (**Integer** in Visual Basic). The method has access to the private and protected members of the `Example` class.

VB

```
Dim multiplyPrivate As New DynamicMethod( _
    "", _
    GetType(Integer), _
    methodArgs2, _
    GetType(Example))
```

4. Emit the method body. In this example, an ILGenerator object is used to emit the Microsoft intermediate language (MSIL). Alternatively, a DynamicILInfo object can be used in conjunction with unmanaged code generators to emit the method body for a DynamicMethod.

   The MSIL in this example loads the first argument, which is an instance of the `Example` class, and uses it to load the value of a private instance field of type **int**. The second argument is loaded, and the two numbers are multiplied. If the result is larger than **int**, the value is truncated and the most significant bits are discarded. The method returns, with the return value on the stack.

VB

```
Dim ilMP As ILGenerator = multiplyPrivate.GetILGenerator()
ilMP.Emit(OpCodes.Ldarg_0)

Dim testInfo As FieldInfo = _
    GetType(Example).GetField("test", _
        BindingFlags.NonPublic Or BindingFlags.Instance)

ilMP.Emit(OpCodes.Ldfld, testInfo)
ilMP.Emit(OpCodes.Ldarg_1)
ilMP.Emit(OpCodes.Mul)
ilMP.Emit(OpCodes.Ret)
```

5. Create an instance of the delegate (declared in step 1) that represents the dynamic method by calling the CreateDelegate(Type, Object) method overload. Creating the delegate completes the method, and any further

attempts to change the method — for example, adding more MSIL — are ignored.

Note

You can call the CreateDelegate method multiple times to create delegates bound to other instances of the target type.

The following code binds the method to a new instance of the `Example` class whose private test field is set to 42. That is, each time the delegate is invoked the instance of `Example` is passed to the first parameter of the method.

The delegate `OneParameter` is used because the first parameter of the method always receives the instance of `Example`. When the delegate is invoked, only the second parameter is required.

VB

```
Dim invoke As OneParameter(Of Integer, Integer) = _
    CType( _
        multiplyPrivate.CreateDelegate( _
            GetType(OneParameter(Of Integer, Integer)), _
            new Example(42) _
        ), _
        OneParameter(Of Integer, Integer) _
    )

Console.WriteLine("3 * test = {0}", invoke(3))
```

Example

The following code example demonstrates a simple dynamic method and a dynamic method bound to an instance of a class.

The simple dynamic method takes one argument, a 32-bit integer, and returns the 64-bit square of that integer. A generic delegate is used to invoke the method.

The second dynamic method has two parameters, of type `Example` and type **int** (**Integer** in Visual Basic). When the dynamic method has been created, it is bound to an instance of `Example`, using a generic delegate that has one argument of type **int**. The delegate does not have an argument of type `Example` because the first parameter of the method always receives the bound instance of `Example`. When the delegate is invoked, only the **int** argument is supplied. This dynamic method accesses a private field of the `Example` class and returns the product of the private field and the **int** argument.

The code example defines delegates that can be used to execute the methods.

VB

```
Imports System
Imports System.Reflection
Imports System.Reflection.Emit

Public Class Example

    ' The following constructor and private field are used to
    ' demonstrate a method bound to an object.
    '
    Private test As Integer
    Public Sub New(ByVal test As Integer)
        Me.test = test
    End Sub

    ' Declare delegates that can be used to execute the completed
    ' SquareIt dynamic method. The OneParameter delegate can be
    ' used to execute any method with one parameter and a return
    ' value, or a method with two parameters and a return value
    ' if the delegate is bound to an object.
    '
    Private Delegate Function _
        SquareItInvoker(ByVal input As Integer) As Long

    Private Delegate Function _
        OneParameter(Of TReturn, TParameter0) _
        (ByVal p0 As TParameter0) As TReturn

    Public Shared Sub Main()

        ' Example 1: A simple dynamic method.
        '
        ' Create an array that specifies the parameter types for the
        ' dynamic method. In this example the only parameter is an
        ' Integer, so the array has only one element.
        '
        Dim methodArgs As Type() = { GetType(Integer) }

        ' Create a DynamicMethod. In this example the method is
        ' named SquareIt. It is not necessary to give dynamic
        ' methods names. They cannot be invoked by name, and two
        ' dynamic methods can have the same name. However, the
        ' name appears in calls stacks and can be useful for
        ' debugging.
        '
        ' In this example the return type of the dynamic method
        ' is Long. The method is associated with the module that
        ' contains the Example class. Any loaded module could be
        ' specified. The dynamic method is like a module-level
        ' Shared method.
        '
        Dim squareIt As New DynamicMethod( _
            "SquareIt", _
            GetType(Long), _
            methodArgs, _
            GetType(Example).Module)

        ' Emit the method body. In this example ILGenerator is used
        ' to emit the MSIL. DynamicMethod has an associated type
        ' DynamicILInfo that can be used in conjunction with
        ' unmanaged code generators.
        '
```

```
' The MSIL loads the argument, which is an Integer, onto the
' stack, converts the Integer to a Long, duplicates the top
' item on the stack, and multiplies the top two items on the
' stack. This leaves the squared number on the stack, and
' all the method has to do is return.
'
Dim il As ILGenerator = squareIt.GetILGenerator()
il.Emit(OpCodes.Ldarg_0)
il.Emit(OpCodes.Conv_I8)
il.Emit(OpCodes.Dup)
il.Emit(OpCodes.Mul)
il.Emit(OpCodes.Ret)

' Create a delegate that represents the dynamic method.
' Creating the delegate completes the method, and any further
' attempts to change the method (for example, by adding more
' MSIL) are ignored. The following code uses a generic
' delegate that can produce delegate types matching any
' single-parameter method that has a return type.
'
Dim invokeSquareIt As OneParameter(Of Long, Integer) = _
    CType( _
        squareIt.CreateDelegate( _
            GetType(OneParameter(Of Long, Integer))), _
        OneParameter(Of Long, Integer) _
    )

Console.WriteLine("123456789 squared = {0}", _
    invokeSquareIt(123456789))


' Example 2: A dynamic method bound to an instance.
'
' Create an array that specifies the parameter types for a
' dynamic method. If the delegate representing the method
' is to be bound to an object, the first parameter must
' match the type the delegate is bound to. In the following
' code the bound instance is of the Example class.
'
Dim methodArgs2 As Type() = _
    { GetType(Example), GetType(Integer) }

' Create a DynamicMethod. In this example the method has no
' name. The return type of the method is Integer. The method
' has access to the protected and private members of the
' Example class.
'
Dim multiplyPrivate As New DynamicMethod( _
    "", _
    GetType(Integer), _
    methodArgs2, _
    GetType(Example))

' Emit the method body. In this example ILGenerator is used
```

```
        ' to emit the MSIL. DynamicMethod has an associated type
        ' DynamicILInfo that can be used in conjunction with
        ' unmanaged code generators.
        '
        ' The MSIL loads the first argument, which is an instance of
        ' the Example class, and uses it to load the value of a
        ' private instance field of type Integer. The second argument
        ' is loaded, and the two numbers are multiplied. If the result
        ' is larger than Integer, the value is truncated and the most
        ' significant bits are discarded. The method returns, with
        ' the return value on the stack.
        '
        Dim ilMP As ILGenerator = multiplyPrivate.GetILGenerator()
        ilMP.Emit(OpCodes.Ldarg_0)

        Dim testInfo As FieldInfo = _
            GetType(Example).GetField("test", _
                BindingFlags.NonPublic Or BindingFlags.Instance)

        ilMP.Emit(OpCodes.Ldfld, testInfo)
        ilMP.Emit(OpCodes.Ldarg_1)
        ilMP.Emit(OpCodes.Mul)
        ilMP.Emit(OpCodes.Ret)

        ' Create a delegate that represents the dynamic method.
        ' Creating the delegate completes the method, and any further
        ' attempts to change the method  for example, by adding more
        ' MSIL  are ignored.
        '
        ' The following code binds the method to a new instance
        ' of the Example class whose private test field is set to 42.
        ' That is, each time the delegate is invoked the instance of
        ' Example is passed to the first parameter of the method.
        '
        ' The delegate OneParameter is used, because the first
        ' parameter of the method receives the instance of Example.
        ' When the delegate is invoked, only the second parameter is
        ' required.
        '
        Dim invoke As OneParameter(Of Integer, Integer) = _
            CType( _
                multiplyPrivate.CreateDelegate( _
                    GetType(OneParameter(Of Integer, Integer)), _
                    new Example(42) _
                ), _
                OneParameter(Of Integer, Integer) _
            )

        Console.WriteLine("3 * test = {0}", invoke(3))

    End Sub

End Class
```

```
' This code example produces the following output:
'
'123456789 squared = 15241578750190521
'3 * test = 126
'
```

Compiling the Code

- The code contains the C# **using** statements (**Imports** in Visual Basic) necessary for compilation.
- No additional assembly references are required.
- Compile the code at the command line using csc.exe, vbc.exe, or cl.exe. To compile the code in Visual Studio, place it in a console application project template.

See Also

DynamicMethod
Using Reflection Emit
Reflection Emit Dynamic Method Scenarios

How to: Define a Generic Type with Reflection Emit

.NET Framework (current version)

This topic shows how to create a simple generic type with two type parameters, how to apply class constraints, interface constraints, and special constraints to the type parameters, and how to create members that use the type parameters of the class as parameter types and return types.

Important

A method is not generic just because it belongs to a generic type and uses the type parameters of that type. A method is generic only if it has its own type parameter list. Most methods on generic types are not generic, as in this example. For an example of emitting a generic method, see How to: Define a Generic Method with Reflection Emit.

To define a generic type

1. Define a dynamic assembly named `GenericEmitExample1`. In this example, the assembly is executed and saved to disk, so AssemblyBuilderAccess.RunAndSave is specified.

VB

```
Dim myDomain As AppDomain = AppDomain.CurrentDomain
Dim myAsmName As New AssemblyName("GenericEmitExample1")
Dim myAssembly As AssemblyBuilder = myDomain.DefineDynamicAssembly( _
    myAsmName, _
    AssemblyBuilderAccess.RunAndSave)
```

2. Define a dynamic module. An assembly is made up of executable modules. For a single-module assembly, the module name is the same as the assembly name, and the file name is the module name plus an extension.

VB

```
Dim myModule As ModuleBuilder = myAssembly.DefineDynamicModule( _
    myAsmName.Name, _
    myAsmName.Name & ".dll")
```

3. Define a class. In this example, the class is named `Sample`.

VB

```
Dim myType As TypeBuilder = myModule.DefineType( _
    "Sample", _
```

```
        TypeAttributes.Public)
```

4. Define the generic type parameters of `Sample` by passing an array of strings containing the names of the parameters to the TypeBuilder.DefineGenericParameters method. This makes the class a generic type. The return value is an array of GenericTypeParameterBuilder objects representing the type parameters, which can be used in your emitted code.

   In the following code, `Sample` becomes a generic type with type parameters `TFirst` and `TSecond`. To make the code easier to read, each GenericTypeParameterBuilder is placed in a variable with the same name as the type parameter.

   VB

```
Dim typeParamNames() As String = {"TFirst", "TSecond"}
Dim typeParams() As GenericTypeParameterBuilder = _
    myType.DefineGenericParameters(typeParamNames)

Dim TFirst As GenericTypeParameterBuilder = typeParams(0)
Dim TSecond As GenericTypeParameterBuilder = typeParams(1)
```

5. Add special constraints to the type parameters. In this example, type parameter `TFirst` is constrained to types that have parameterless constructors, and to reference types.

   VB

```
TFirst.SetGenericParameterAttributes( _
    GenericParameterAttributes.DefaultConstructorConstraint _
    Or GenericParameterAttributes.ReferenceTypeConstraint)
```

6. Optionally add class and interface constraints to the type parameters. In this example, type parameter `TFirst` is constrained to types that derive from the base class represented by the Type object contained in the variable `baseType`, and that implement the interfaces whose types are contained in the variables `interfaceA` and `interfaceB`. See the code example for the declaration and assignment of these variables.

   VB

```
TSecond.SetBaseTypeConstraint(baseType)
Dim interfaceTypes() As Type = {interfaceA, interfaceB}
TSecond.SetInterfaceConstraints(interfaceTypes)
```

7. Define a field. In this example, the type of the field is specified by type parameter `TFirst`. GenericTypeParameterBuilder derives from Type, so you can use generic type parameters anywhere a type can be used.

   VB

```
Dim exField As FieldBuilder = _
    myType.DefineField("ExampleField", TFirst, _
        FieldAttributes.Private)
```

8. Define a method that uses the type parameters of the generic type. Note that such methods are not generic unless

they have their own type parameter lists. The following code defines a **static** method (**Shared** in Visual Basic) that takes an array of `TFirst` and returns a `List<TFirst>` (`List(Of TFirst)` in Visual Basic) containing all the elements of the array. To define this method, it is necessary to create the type `List<TFirst>` by calling MakeGenericType on the generic type definition, `List<T>`. (The `T` is omitted when you use the **typeof** operator (**GetType** in Visual Basic) to get the generic type definition.) The parameter type is created by using the MakeArrayType method.

VB

```
Dim listOf As Type = GetType(List(Of ))
Dim listOfTFirst As Type = listOf.MakeGenericType(TFirst)
Dim mParamTypes() As Type = { TFirst.MakeArrayType() }

Dim exMethod As MethodBuilder = _
    myType.DefineMethod("ExampleMethod", _
        MethodAttributes.Public Or MethodAttributes.Static, _
        listOfTFirst, _
        mParamTypes)
```

9. Emit the method body. The method body consists of three opcodes that load the input array onto the stack, call the `List<TFirst>` constructor that takes `IEnumerable<TFirst>` (which does all the work of putting the input elements into the list), and return (leaving the new List(Of T) object on the stack). The difficult part of emitting this code is getting the constructor.

   The GetConstructor method is not supported on a GenericTypeParameterBuilder, so it is not possible to get the constructor of `List<TFirst>` directly. First, it is necessary to get the constructor of the generic type definition `List<T>` and then to call a method that converts it to the corresponding constructor of `List<TFirst>`.

   The constructor used for this code example takes an `IEnumerable<T>`. Note, however, that this is not the generic type definition of the IEnumerable(Of T) generic interface; instead, the type parameter `T` from `List<T>` must be substituted for the type parameter `T` of `IEnumerable<T>`. (This seems confusing only because both types have type parameters named `T`. That is why this code example uses the names `TFirst` and `TSecond`.) To get the type of the constructor argument, start with the generic type definition `IEnumerable<T>` and call MakeGenericType with the first generic type parameter of `List<T>`. The constructor argument list must be passed as an array, with just one argument in this case.

   Note

   The generic type definition is expressed as `IEnumerable<>` when you use the **typeof** operator in C#, or `IEnumerable(Of )` when you use the **GetType** operator in Visual Basic.

   Now it is possible to get the constructor of `List<T>` by calling GetConstructor on the generic type definition. To convert this constructor to the corresponding constructor of `List<TFirst>`, pass `List<TFirst>` and the constructor from `List<T>` to the static TypeBuilder.GetConstructor(Type, ConstructorInfo) method.

   VB

```
Dim ilgen As ILGenerator = exMethod.GetILGenerator()

Dim ienumOf As Type = GetType(IEnumerable(Of ))
Dim listOfTParams() As Type = listOf.GetGenericArguments()
Dim TfromListOf As Type = listOfTParams(0)
Dim ienumOfT As Type = ienumOf.MakeGenericType(TfromListOf)
Dim ctorArgs() As Type = { ienumOfT }

Dim ctorPrep As ConstructorInfo = _
    listOf.GetConstructor(ctorArgs)
Dim ctor As ConstructorInfo = _
    TypeBuilder.GetConstructor(listOfTFirst, ctorPrep)

ilgen.Emit(OpCodes.Ldarg_0)
ilgen.Emit(OpCodes.Newobj, ctor)
ilgen.Emit(OpCodes.Ret)
```

10. Create the type and save the file.

VB

```
Dim finished As Type = myType.CreateType()
myAssembly.Save(myAsmName.Name & ".dll")
```

11. Invoke the method. `ExampleMethod` is not generic, but the type it belongs to is generic, so in order to get a MethodInfo that can be invoked it is necessary to create a constructed type from the type definition for `Sample`. The constructed type uses the `Example` class, which satisfies the constraints on `TFirst` because it is a reference type and has a default parameterless constructor, and the `ExampleDerived` class which satisfies the constraints on `TSecond`. (The code for `ExampleDerived` can be found in the example code section.) These two types are passed to MakeGenericType to create the constructed type. The MethodInfo is then obtained using the GetMethod method.

VB

```
Dim typeArgs() As Type = _
    { GetType(Example), GetType(ExampleDerived) }
Dim constructed As Type = finished.MakeGenericType(typeArgs)
Dim mi As MethodInfo = constructed.GetMethod("ExampleMethod")
```

12. The following code creates an array of `Example` objects, places that array in an array of type Object representing the arguments of the method to be invoked, and passes them to the Invoke(Object, Object()) method. The first argument of the Invoke method is a null reference because the method is **static**.

VB

```
Dim input() As Example = { New Example(), New Example() }
Dim arguments() As Object = { input }

Dim listX As List(Of Example) = mi.Invoke(Nothing, arguments)

Console.WriteLine(vbLf & _
    "There are {0} elements in the List(Of Example).", _
    listX.Count _
)
```

Example

The following code example defines a class named Sample, along with a base class and two interfaces. The program defines two generic type parameters for Sample, turning it into a generic type. Type parameters are the only thing that makes a type generic. The program shows this by displaying a test message before and after the definition of the type parameters.

The type parameter TSecond is used to demonstrate class and interface constraints, using the base class and interfaces, and the type parameter TFirst is used to demonstrate special constraints.

The code example defines a field and a method using the class's type parameters for the field type and for the parameter and return type of the method.

After the Sample class has been created, the method is invoked.

The program includes a method that lists information about a generic type, and a method that lists the special constraints on a type parameter. These methods are used to display information about the finished Sample class.

The program saves the finished module to disk as GenericEmitExample1.dll, so you can open it with the Ildasm.exe (IL Disassembler) and examine the MSIL for the Sample class.

VB

```
Imports System
Imports System.Reflection
Imports System.Reflection.Emit
Imports System.Collections.Generic

' Define a trivial base class and two trivial interfaces
' to use when demonstrating constraints.
'
Public Class ExampleBase
End Class

Public Interface IExampleA
End Interface

Public Interface IExampleB
End Interface

' Define a trivial type that can substitute for type parameter
' TSecond.
'
Public Class ExampleDerived
    Inherits ExampleBase
    Implements IExampleA, IExampleB
End Class

Public Class Example
    Public Shared Sub Main()
        ' Define a dynamic assembly to contain the sample type. The
        ' assembly will not be run, but only saved to disk, so
        ' AssemblyBuilderAccess.Save is specified.
        '
```

```
Dim myDomain As AppDomain = AppDomain.CurrentDomain
Dim myAsmName As New AssemblyName("GenericEmitExample1")
Dim myAssembly As AssemblyBuilder = myDomain.DefineDynamicAssembly( _
    myAsmName, _
    AssemblyBuilderAccess.RunAndSave)

' An assembly is made up of executable modules. For a single-
' module assembly, the module name and file name are the same
' as the assembly name.
'
Dim myModule As ModuleBuilder = myAssembly.DefineDynamicModule( _
    myAsmName.Name, _
    myAsmName.Name & ".dll")

' Get type objects for the base class trivial interfaces to
' be used as constraints.
'
Dim baseType As Type = GetType(ExampleBase)
Dim interfaceA As Type = GetType(IExampleA)
Dim interfaceB As Type = GetType(IExampleB)

' Define the sample type.
'
Dim myType As TypeBuilder = myModule.DefineType( _
    "Sample", _
    TypeAttributes.Public)

Console.WriteLine("Type 'Sample' is generic: {0}", _
    myType.IsGenericType)

' Define type parameters for the type. Until you do this,
' the type is not generic, as the preceding and following
' WriteLine statements show. The type parameter names are
' specified as an array of strings. To make the code
' easier to read, each GenericTypeParameterBuilder is placed
' in a variable with the same name as the type parameter.
'
Dim typeParamNames() As String = {"TFirst", "TSecond"}
Dim typeParams() As GenericTypeParameterBuilder = _
    myType.DefineGenericParameters(typeParamNames)

Dim TFirst As GenericTypeParameterBuilder = typeParams(0)
Dim TSecond As GenericTypeParameterBuilder = typeParams(1)

Console.WriteLine("Type 'Sample' is generic: {0}", _
    myType.IsGenericType)

' Apply constraints to the type parameters.
'
' A type that is substituted for the first parameter, TFirst,
' must be a reference type and must have a parameterless
' constructor.
TFirst.SetGenericParameterAttributes( _
    GenericParameterAttributes.DefaultConstructorConstraint _
```

```
            Or GenericParameterAttributes.ReferenceTypeConstraint)

        ' A type that is substituted for the second type
        ' parameter must implement IExampleA and IExampleB, and
        ' inherit from the trivial test class ExampleBase. The
        ' interface constraints are specified as an array
        ' containing the interface types.
        TSecond.SetBaseTypeConstraint(baseType)
        Dim interfaceTypes() As Type = {interfaceA, interfaceB}
        TSecond.SetInterfaceConstraints(interfaceTypes)

        ' The following code adds a private field named ExampleField,
        ' of type TFirst.
        Dim exField As FieldBuilder = _
            myType.DefineField("ExampleField", TFirst, _
                FieldAttributes.Private)

        ' Define a Shared method that takes an array of TFirst and
        ' returns a List(Of TFirst) containing all the elements of
        ' the array. To define this method it is necessary to create
        ' the type List(Of TFirst) by calling MakeGenericType on the
        ' generic type definition, List(Of T). (The T is omitted with
        ' the GetType operator when you get the generic type
        ' definition.) The parameter type is created by using the
        ' MakeArrayType method.
        '
        Dim listOf As Type = GetType(List(Of ))
        Dim listOfTFirst As Type = listOf.MakeGenericType(TFirst)
        Dim mParamTypes() As Type = { TFirst.MakeArrayType() }

        Dim exMethod As MethodBuilder = _
            myType.DefineMethod("ExampleMethod", _
                MethodAttributes.Public Or MethodAttributes.Static, _
                listOfTFirst, _
                mParamTypes)

        ' Emit the method body.
        ' The method body consists of just three opcodes, to load
        ' the input array onto the execution stack, to call the
        ' List(Of TFirst) constructor that takes IEnumerable(Of TFirst),
        ' which does all the work of putting the input elements into
        ' the list, and to return, leaving the list on the stack. The
        ' hard work is getting the constructor.
        '
        ' The GetConstructor method is not supported on a
        ' GenericTypeParameterBuilder, so it is not possible to get
        ' the constructor of List(Of TFirst) directly. There are two
        ' steps, first getting the constructor of List(Of T) and then
        ' calling a method that converts it to the corresponding
        ' constructor of List(Of TFirst).
        '
        ' The constructor needed here is the one that takes an
        ' IEnumerable(Of T). Note, however, that this is not the
        ' generic type definition of IEnumerable(Of T); instead, the
```

```
' T from List(Of T) must be substituted for the T of
' IEnumerable(Of T). (This seems confusing only because both
' types have type parameters named T. That is why this example
' uses the somewhat silly names TFirst and TSecond.) To get
' the type of the constructor argument, take the generic
' type definition IEnumerable(Of T) (expressed as
' IEnumerable(Of ) when you use the GetType operator) and
' call MakeGenericType with the first generic type parameter
' of List(Of T). The constructor argument list must be passed
' as an array, with just one argument in this case.
'
' Now it is possible to get the constructor of List(Of T),
' using GetConstructor on the generic type definition. To get
' the constructor of List(Of TFirst), pass List(Of TFirst) and
' the constructor from List(Of T) to the static
' TypeBuilder.GetConstructor method.
'
Dim ilgen As ILGenerator = exMethod.GetILGenerator()

Dim ienumOf As Type = GetType(IEnumerable(Of ))
Dim listOfTParams() As Type = listOf.GetGenericArguments()
Dim TfromListOf As Type = listOfTParams(0)
Dim ienumOfT As Type = ienumOf.MakeGenericType(TfromListOf)
Dim ctorArgs() As Type = { ienumOfT }

Dim ctorPrep As ConstructorInfo = _
    listOf.GetConstructor(ctorArgs)
Dim ctor As ConstructorInfo = _
    TypeBuilder.GetConstructor(listOfTFirst, ctorPrep)

ilgen.Emit(OpCodes.Ldarg_0)
ilgen.Emit(OpCodes.Newobj, ctor)
ilgen.Emit(OpCodes.Ret)

' Create the type and save the assembly.
Dim finished As Type = myType.CreateType()
myAssembly.Save(myAsmName.Name & ".dll")

' Invoke the method.
' ExampleMethod is not generic, but the type it belongs to is
' generic, so in order to get a MethodInfo that can be invoked
' it is necessary to create a constructed type. The Example
' class satisfies the constraints on TFirst, because it is a
' reference type and has a default constructor. In order to
' have a class that satisfies the constraints on TSecond,
' this code example defines the ExampleDerived type. These
' two types are passed to MakeGenericMethod to create the
' constructed type.
'
Dim typeArgs() As Type = _
    { GetType(Example), GetType(ExampleDerived) }
Dim constructed As Type = finished.MakeGenericType(typeArgs)
Dim mi As MethodInfo = constructed.GetMethod("ExampleMethod")
```

```
        ' Create an array of Example objects, as input to the generic
        ' method. This array must be passed as the only element of an
        ' array of arguments. The first argument of Invoke is
        ' Nothing, because ExampleMethod is Shared. Display the count
        ' on the resulting List(Of Example).
        '
        Dim input() As Example = { New Example(), New Example() }
        Dim arguments() As Object = { input }

        Dim listX As List(Of Example) = mi.Invoke(Nothing, arguments)

        Console.WriteLine(vbLf & _
            "There are {0} elements in the List(Of Example).", _
            listX.Count _
        )

        DisplayGenericParameters(finished)
    End Sub

    Private Shared Sub DisplayGenericParameters(ByVal t As Type)

        If Not t.IsGenericType Then
            Console.WriteLine("Type '{0}' is not generic.")
            Return
        End If
        If Not t.IsGenericTypeDefinition Then _
            t = t.GetGenericTypeDefinition()

        Dim typeParameters() As Type = t.GetGenericArguments()
        Console.WriteLine(vbCrLf & _
            "Listing {0} type parameters for type '{1}'.", _
            typeParameters.Length, t)

        For Each tParam As Type In typeParameters

            Console.WriteLine(vbCrLf & "Type parameter {0}:", _
                tParam.ToString())

            For Each c As Type In tParam.GetGenericParameterConstraints()
                If c.IsInterface Then
                    Console.WriteLine("    Interface constraint: {0}", c)
                Else
                    Console.WriteLine("    Base type constraint: {0}", c)
                End If
            Next

            ListConstraintAttributes(tParam)
        Next tParam
    End Sub

    ' List the constraint flags. The GenericParameterAttributes
    ' enumeration contains two sets of attributes, variance and
    ' constraints. For this example, only constraints are used.
    '
```

```
        Private Shared Sub ListConstraintAttributes(ByVal t As Type)

            ' Mask off the constraint flags.
            Dim constraints As GenericParameterAttributes = _
                t.GenericParameterAttributes And _
                GenericParameterAttributes.SpecialConstraintMask

            If (constraints And GenericParameterAttributes.ReferenceTypeConstraint) _
                    <> GenericParameterAttributes.None Then _
                Console.WriteLine("    ReferenceTypeConstraint")

            If (constraints And GenericParameterAttributes.NotNullableValueTypeConstraint) _
                    <> GenericParameterAttributes.None Then _
                Console.WriteLine("    NotNullableValueTypeConstraint")

            If (constraints And GenericParameterAttributes.DefaultConstructorConstraint) _
                    <> GenericParameterAttributes.None Then _
                Console.WriteLine("    DefaultConstructorConstraint")

        End Sub

    End Class

    ' This code example produces the following output:
    '
    'Type 'Sample' is generic: False
    'Type 'Sample' is generic: True
    '
    'There are 2 elements in the List(Of Example).
    '
    'Listing 2 type parameters for type 'Sample[TFirst,TSecond]'.
    '
    'Type parameter TFirst:
    '    ReferenceTypeConstraint
    '    DefaultConstructorConstraint
    '
    'Type parameter TSecond:
    '    Interface constraint: IExampleA
    '    Interface constraint: IExampleB
    '    Base type constraint: ExampleBase
```

Compiling the Code

- The code contains the C# **using** statements (**Imports** in Visual Basic) necessary for compilation.
- No additional assembly references are required.
- Compile the code at the command line using csc.exe, vbc.exe, or cl.exe. To compile the code in Visual Studio, place it in a console application project template.

See Also

GenericTypeParameterBuilder
Using Reflection Emit
Reflection Emit Dynamic Assembly Scenarios

How to: Define a Generic Method with Reflection Emit

.NET Framework (current version)

The first procedure shows how to create a simple generic method with two type parameters, and how to apply class constraints, interface constraints, and special constraints to the type parameters.

The second procedure shows how to emit the method body, and how to use the type parameters of the generic method to create instances of generic types and to call their methods.

The third procedure shows how to invoke the generic method.

Important

A method is not generic just because it belongs to a generic type and uses the type parameters of that type. A method is generic only if it has its own type parameter list. A generic method can appear on a nongeneric type, as in this example. For an example of a nongeneric method on a generic type, see How to: Define a Generic Type with Reflection Emit.

To define a generic method

1. Before beginning, it is useful to look at how the generic method appears when written using a high-level language. The following code is included in the example code for this topic, along with code to call the generic method. The method has two type parameters, `TInput` and `TOutput`, the second of which must be a reference type (**class**), must have a parameterless constructor (**new**), and must implement `ICollection(Of TInput)` (`ICollection<TInput>` in C#). This interface constraint ensures that the ICollection(Of T).Add method can be used to add elements to the `TOutput` collection that the method creates. The method has one formal parameter, `input`, which is an array of `TInput`. The method creates a collection of type `TOutput` and copies the elements of `input` to the collection.

VB

```
Public Shared Function Factory(Of TInput, _
    TOutput As {ICollection(Of TInput), Class, New}) _
    (ByVal input() As TInput) As TOutput

    Dim retval As New TOutput()
    Dim ic As ICollection(Of TInput) = retval

    For Each t As TInput In input
        ic.Add(t)
    Next

    Return retval
End Function
```

2. Define a dynamic assembly and a dynamic module to contain the type the generic method belongs to. In this case, the assembly has only one module, named `DemoMethodBuilder1`, and the module name is the same as the assembly name plus an extension. In this example, the assembly is saved to disk and also executed, so AssemblyBuilderAccess.RunAndSave is specified. You can use the Ildasm.exe (IL Disassembler) to examine DemoMethodBuilder1.dll and to compare it to the Microsoft intermediate language (MSIL) for the method shown in step 1.

VB

```
Dim asmName As New AssemblyName("DemoMethodBuilder1")
Dim domain As AppDomain = AppDomain.CurrentDomain
Dim demoAssembly As AssemblyBuilder = _
    domain.DefineDynamicAssembly(asmName, _
        AssemblyBuilderAccess.RunAndSave)

' Define the module that contains the code. For an
' assembly with one module, the module name is the
' assembly name plus a file extension.
Dim demoModule As ModuleBuilder = _
    demoAssembly.DefineDynamicModule( _
        asmName.Name, _
        asmName.Name & ".dll")
```

3. Define the type the generic method belongs to. The type does not have to be generic. A generic method can belong to either a generic or nongeneric type. In this example, the type is a class, is not generic, and is named `DemoType`.

VB

```
Dim demoType As TypeBuilder = demoModule.DefineType( _
    "DemoType", _
    TypeAttributes.Public)
```

4. Define the generic method. If the types of a generic method's formal parameters are specified by generic type parameters of the generic method, use the DefineMethod(String, MethodAttributes) method overload to define the method. The generic type parameters of the method are not yet defined, so you cannot specify the types of the method's formal parameters in the call to DefineMethod. In this example, the method is named `Factory`. The method is public and **static** (**Shared** in Visual Basic).

VB

```
Dim factory As MethodBuilder = _
    demoType.DefineMethod("Factory", _
        MethodAttributes.Public Or MethodAttributes.Static)
```

5. Define the generic type parameters of `DemoMethod` by passing an array of strings containing the names of the parameters to the MethodBuilder.DefineGenericParameters method. This makes the method a generic method. The following code makes `Factory` a generic method with type parameters `TInput` and `TOutput`. To make the code easier to read, variables with these names are created to hold the GenericTypeParameterBuilder objects representing the two type parameters.

VB

```
Dim typeParameterNames() As String = {"TInput", "TOutput"}
Dim typeParameters() As GenericTypeParameterBuilder = _
    factory.DefineGenericParameters(typeParameterNames)

Dim TInput As GenericTypeParameterBuilder = typeParameters(0)
Dim TOutput As GenericTypeParameterBuilder = typeParameters(1)
```

6. Optionally add special constraints to the type parameters. Special constraints are added using the SetGenericParameterAttributes method. In this example, `TOutput` is constrained to be a reference type and to have a parameterless constructor.

VB

```
TOutput.SetGenericParameterAttributes( _
    GenericParameterAttributes.ReferenceTypeConstraint Or _
    GenericParameterAttributes.DefaultConstructorConstraint)
```

7. Optionally add class and interface constraints to the type parameters. In this example, type parameter `TOutput` is constrained to types that implement the `ICollection(Of TInput)` (`ICollection<TInput>` in C#) interface. This ensures that the Add method can be used to add elements.

VB

```
Dim icoll As Type = GetType(ICollection(Of ))
Dim icollOfTInput As Type = icoll.MakeGenericType(TInput)
Dim constraints() As Type = { icollOfTInput }
TOutput.SetInterfaceConstraints(constraints)
```

8. Define the formal parameters of the method, using the SetParameters method. In this example, the `Factory` method has one parameter, an array of `TInput`. This type is created by calling the MakeArrayType method on the GenericTypeParameterBuilder that represents `TInput`. The argument of SetParameters is an array of Type objects.

VB

```
Dim params() As Type = { TInput.MakeArrayType() }
factory.SetParameters(params)
```

9. Define the return type for the method, using the SetReturnType method. In this example, an instance of `TOutput` is returned.

VB

```
factory.SetReturnType(TOutput)
```

10. Emit the method body, using ILGenerator. For details, see the accompanying procedure for emitting the method body.

Important

When you emit calls to methods of generic types, and the type arguments of those types are type parameters of the generic method, you must use the **static** GetConstructor(Type, ConstructorInfo), GetMethod(Type, MethodInfo), and GetField(Type, FieldInfo) method overloads of the TypeBuilder class to obtain constructed forms of the methods. The accompanying procedure for emitting the method body demonstrates this.

11. Complete the type that contains the method and save the assembly. The accompanying procedure for invoking the generic method shows two ways to invoke the completed method.

VB

```
' Complete the type.
Dim dt As Type = demoType.CreateType()
' Save the assembly, so it can be examined with Ildasm.exe.
demoAssembly.Save(asmName.Name & ".dll")
```

To emit the method body

1. Get a code generator and declare local variables and labels. The DeclareLocal method is used to declare local variables. The `Factory` method has four local variables: `retVal` to hold the new `TOutput` that is returned by the method, `ic` to hold the `TOutput` when it is cast to `ICollection(Of TInput)` (`ICollection<TInput>` in C#), `input` to hold the input array of `TInput` objects, and `index` to iterate through the array. The method also has two labels, one to enter the loop (`enterLoop`) and one for the top of the loop (`loopAgain`), defined using the DefineLabel method.

   The first thing the method does is to load its argument using Ldarg_0 opcode and to store it in the local variable `input` using Stloc_S opcode.

VB

```
Dim ilgen As ILGenerator = factory.GetILGenerator()

Dim retVal As LocalBuilder = ilgen.DeclareLocal(TOutput)
Dim ic As LocalBuilder = ilgen.DeclareLocal(icollOfTInput)
Dim input As LocalBuilder = _
    ilgen.DeclareLocal(TInput.MakeArrayType())
Dim index As LocalBuilder = _
    ilgen.DeclareLocal(GetType(Integer))

Dim enterLoop As Label = ilgen.DefineLabel()
Dim loopAgain As Label = ilgen.DefineLabel()

ilgen.Emit(OpCodes.Ldarg_0)
ilgen.Emit(OpCodes.Stloc_S, input)
```

2. Emit code to create an instance of `TOutput`, using the generic method overload of the Activator.CreateInstance(Of T) method. Using this overload requires the specified type to have a parameterless constructor, which is the reason for

adding that constraint to `TOutput`. Create the constructed generic method by passing `TOutput` to MakeGenericMethod. After emitting code to call the method, emit code to store it in the local variable `retVal` using Stloc_S

VB

```
Dim createInst As MethodInfo = _
    GetType(Activator).GetMethod("CreateInstance", Type.EmptyTypes)
Dim createInstOfTOutput As MethodInfo = _
    createInst.MakeGenericMethod(TOutput)

ilgen.Emit(OpCodes.Call, createInstOfTOutput)
ilgen.Emit(OpCodes.Stloc_S, retVal)
```

3. Emit code to cast the new `TOutput` object to `ICollection(Of TInput)` and store it in the local variable `ic`.

VB

```
ilgen.Emit(OpCodes.Ldloc_S, retVal)
ilgen.Emit(OpCodes.Box, TOutput)
ilgen.Emit(OpCodes.Castclass, icollOfTInput)
ilgen.Emit(OpCodes.Stloc_S, ic)
```

4. Get a MethodInfo representing the ICollection(Of T).Add method. The method is acting on an `ICollection(Of TInput)` (`ICollection<TInput>` in C#), so it is necessary to get the **Add** method specific to that constructed type. You cannot use the GetMethod method to get this MethodInfo directly from `icollOfTInput`, because GetMethod is not supported on a type that has been constructed with a GenericTypeParameterBuilder. Instead, call GetMethod on `icoll`, which contains the generic type definition for the ICollection(Of T) generic interface. Then use the GetMethod(Type, MethodInfo) **static** method to produce the MethodInfo for the constructed type. The following code demonstrates this.

VB

```
Dim mAddPrep As MethodInfo = icoll.GetMethod("Add")
Dim mAdd As MethodInfo = _
    TypeBuilder.GetMethod(icollOfTInput, mAddPrep)
```

5. Emit code to initialize the `index` variable, by loading a 32-bit integer 0 and storing it in the variable. Emit code to branch to the label `enterLoop`. This label has not yet been marked, because it is inside the loop. Code for the loop is emitted in the next step.

VB

```
' Initialize the count and enter the loop.
ilgen.Emit(OpCodes.Ldc_I4_0)
ilgen.Emit(OpCodes.Stloc_S, index)
ilgen.Emit(OpCodes.Br_S, enterLoop)
```

6. Emit code for the loop. The first step is to mark the top of the loop, by calling MarkLabel with the `loopAgain` label.

Branch statements that use the label will now branch to this point in the code. The next step is to push the `TOutput` object, cast to `ICollection(Of TInput)`, onto the stack. It is not needed immediately, but needs to be in position for calling the **Add** method. Next the input array is pushed onto the stack, then the `index` variable containing the current index into the array. The Ldelem opcode pops the index and the array off the stack and pushes the indexed array element onto the stack. The stack is now ready for the call to the ICollection(Of T).Add method, which pops the collection and the new element off the stack and adds the element to the collection.

The rest of the code in the loop increments the index and tests to see whether the loop is finished: The index and a 32-bit integer 1 are pushed onto the stack and added, leaving the sum on the stack; the sum is stored in `index`. MarkLabel is called to set this point as the entry point for the loop. The index is loaded again. The input array is pushed on the stack, and Ldlen is emitted to get its length. The index and the length are now on the stack, and Clt is emitted to compare them. If the index is less than the length, Brtrue_S branches back to the beginning of the loop.

VB

```
ilgen.MarkLabel(loopAgain)

ilgen.Emit(OpCodes.Ldloc_S, ic)
ilgen.Emit(OpCodes.Ldloc_S, input)
ilgen.Emit(OpCodes.Ldloc_S, index)
ilgen.Emit(OpCodes.Ldelem, TInput)
ilgen.Emit(OpCodes.Callvirt, mAdd)

ilgen.Emit(OpCodes.Ldloc_S, index)
ilgen.Emit(OpCodes.Ldc_I4_1)
ilgen.Emit(OpCodes.Add)
ilgen.Emit(OpCodes.Stloc_S, index)

ilgen.MarkLabel(enterLoop)
ilgen.Emit(OpCodes.Ldloc_S, index)
ilgen.Emit(OpCodes.Ldloc_S, input)
ilgen.Emit(OpCodes.Ldlen)
ilgen.Emit(OpCodes.Conv_I4)
ilgen.Emit(OpCodes.Clt)
ilgen.Emit(OpCodes.Brtrue_S, loopAgain)
```

7. Emit code to push the `TOutput` object onto the stack and return from the method. The local variables `retVal` and `ic` both contain references to the new `TOutput`; `ic` is used only to access the ICollection(Of T).Add method.

VB

```
ilgen.Emit(OpCodes.Ldloc_S, retVal)
ilgen.Emit(OpCodes.Ret)
```

To invoke the generic method

1. `Factory` is a generic method definition. In order to invoke it, you must assign types to its generic type parameters. Use the MakeGenericMethod method to do this. The following code creates a constructed generic method, specifying String for `TInput` and `List(Of String)` (`List<string>` in C#) for `TOutput`, and displays a string representation of the method.

VB

```
Dim m As MethodInfo = dt.GetMethod("Factory")
Dim bound As MethodInfo = m.MakeGenericMethod( _
    GetType(String), GetType(List(Of String)))

' Display a string representing the bound method.
Console.WriteLine(bound)
```

2. To invoke the method late-bound, use the Invoke method. The following code creates an array of Object, containing as its only element an array of strings, and passes it as the argument list for the generic method. The first parameter of Invoke is a null reference because the method is **static**. The return value is cast to `List(Of String)`, and its first element is displayed.

VB

```
Dim o As Object = bound.Invoke(Nothing, New Object() { arr })
Dim list2 As List(Of String) = CType(o, List(Of String))

Console.WriteLine("The first element is: {0}", list2(0))
```

3. To invoke the method using a delegate, you must have a delegate that matches the signature of the constructed generic method. An easy way to do this is to create a generic delegate. The following code creates an instance of the generic delegate `D` defined in the example code, using the Delegate.CreateDelegate(Type, MethodInfo) method overload, and invokes the delegate. Delegates perform better than late-bound calls.

VB

```
Dim dType As Type = GetType(D(Of String, List(Of String)))
Dim test As D(Of String, List(Of String))
test = CType( _
    [Delegate].CreateDelegate(dType, bound), _
    D(Of String, List(Of String)))

Dim list3 As List(Of String) = test(arr)
Console.WriteLine("The first element is: {0}", list3(0))
```

4. The emitted method can also be called from a program that refers to the saved assembly.

Example

The following code example creates a nongeneric type, `DemoType`, with a generic method, `Factory`. This method has two generic type parameters, `TInput` to specify an input type and `TOutput` to specify an output type. The `TOutput` type parameter is constrained to implement `ICollection<TInput>` (`ICollection(Of TInput)` in Visual Basic), to be a reference type, and to have a parameterless constructor.

The method has one formal parameter, which is an array of `TInput`. The method returns an instance of `TOutput` that contains all the elements of the input array. `TOutput` can be any generic collection type that implements the ICollection(Of T) generic interface.

When the code is executed, the dynamic assembly is saved as DemoGenericMethod1.dll, and can be examined using the Ildasm.exe (IL Disassembler).

Note

A good way to learn how to emit code is to write a Visual Basic, C#, or Visual C++ program that performs the task you are trying to emit, and use the disassembler to examine the MSIL produced by the compiler.

The code example includes source code that is equivalent to the emitted method. The emitted method is invoked late-bound and also by using a generic delegate declared in the code example.

VB

```
Imports System
Imports System.Collections.Generic
Imports System.Reflection
Imports System.Reflection.Emit

' Declare a generic delegate that can be used to execute the
' finished method.
'
Delegate Function D(Of TIn, TOut)(ByVal input() As TIn) As TOut

Class GenericMethodBuilder

    ' This method shows how to declare, in Visual Basic, the generic
    ' method this program emits. The method has two type parameters,
    ' TInput and TOutput, the second of which must be a reference type
    ' (Class), must have a parameterless constructor (New), and must
    ' implement ICollection(Of TInput). This interface constraint
    ' ensures that ICollection(Of TInput).Add can be used to add
    ' elements to the TOutput object the method creates. The method
    ' has one formal parameter, input, which is an array of TInput.
    ' The elements of this array are copied to the new TOutput.
    '
    Public Shared Function Factory(Of TInput, _
        TOutput As {ICollection(Of TInput), Class, New}) _
        (ByVal input() As TInput) As TOutput

        Dim retval As New TOutput()
        Dim ic As ICollection(Of TInput) = retval

        For Each t As TInput In input
            ic.Add(t)
        Next

        Return retval
    End Function


    Public Shared Sub Main()
```

```
' The following shows the usage syntax of the Visual Basic
' version of the generic method emitted by this program.
' Note that the generic parameters must be specified
' explicitly, because the compiler does not have enough
' context to infer the type of TOutput. In this case, TOutput
' is a generic List containing strings.
'
Dim arr() As String = {"a", "b", "c", "d", "e"}
Dim list1 As List(Of String) = _
    GenericMethodBuilder.Factory(Of String, List(Of String))(arr)
Console.WriteLine("The first element is: {0}", list1(0))


' Creating a dynamic assembly requires an AssemblyName
' object, and the current application domain.
'
Dim asmName As New AssemblyName("DemoMethodBuilder1")
Dim domain As AppDomain = AppDomain.CurrentDomain
Dim demoAssembly As AssemblyBuilder = _
    domain.DefineDynamicAssembly(asmName, _
        AssemblyBuilderAccess.RunAndSave)

' Define the module that contains the code. For an
' assembly with one module, the module name is the
' assembly name plus a file extension.
Dim demoModule As ModuleBuilder = _
    demoAssembly.DefineDynamicModule( _
        asmName.Name, _
        asmName.Name & ".dll")

' Define a type to contain the method.
Dim demoType As TypeBuilder = demoModule.DefineType( _
    "DemoType", _
    TypeAttributes.Public)

' Define a Shared, Public method with standard calling
' conventions. Do not specify the parameter types or the
' return type, because type parameters will be used for
' those types, and the type parameters have not been
' defined yet.
'
Dim factory As MethodBuilder = _
    demoType.DefineMethod("Factory", _
        MethodAttributes.Public Or MethodAttributes.Static)

' Defining generic type parameters for the method makes it a
' generic method. To make the code easier to read, each
' type parameter is copied to a variable of the same name.
'
Dim typeParameterNames() As String = {"TInput", "TOutput"}
Dim typeParameters() As GenericTypeParameterBuilder = _
    factory.DefineGenericParameters(typeParameterNames)

Dim TInput As GenericTypeParameterBuilder = typeParameters(0)
Dim TOutput As GenericTypeParameterBuilder = typeParameters(1)

' Add special constraints.
' The type parameter TOutput is constrained to be a reference
' type, and to have a parameterless constructor. This ensures
' that the Factory method can create the collection type.
'
TOutput.SetGenericParameterAttributes( _
    GenericParameterAttributes.ReferenceTypeConstraint Or _
    GenericParameterAttributes.DefaultConstructorConstraint)

' Add interface and base type constraints.
' The type parameter TOutput is constrained to types that
' implement the ICollection(Of T) interface, to ensure that
' they have an Add method that can be used to add elements.
'
' To create the constraint, first use MakeGenericType to bind
' the type parameter TInput to the ICollection(Of T) interface,
' returning the type ICollection(Of TInput), then pass
' the newly created type to the SetInterfaceConstraints
' method. The constraints must be passed as an array, even if
' there is only one interface.
'
Dim icoll As Type = GetType(ICollection(Of ))
Dim icollOfTInput As Type = icoll.MakeGenericType(TInput)
Dim constraints() As Type = { icollOfTInput }
TOutput.SetInterfaceConstraints(constraints)

' Set parameter types for the method. The method takes
' one parameter, an array of type TInput.
Dim params() As Type = { TInput.MakeArrayType() }
factory.SetParameters(params)

' Set the return type for the method. The return type is
' the generic type parameter TOutput.
factory.SetReturnType(TOutput)

' Generate a code body for the method.
' -----------------------------------
' Get a code generator and declare local variables and
' labels. Save the input array to a local variable.
'
Dim ilgen As ILGenerator = factory.GetILGenerator()

Dim retVal As LocalBuilder = ilgen.DeclareLocal(TOutput)
Dim ic As LocalBuilder = ilgen.DeclareLocal(icollOfTInput)
Dim input As LocalBuilder = _
    ilgen.DeclareLocal(TInput.MakeArrayType())
Dim index As LocalBuilder = _
    ilgen.DeclareLocal(GetType(Integer))

Dim enterLoop As Label = ilgen.DefineLabel()
Dim loopAgain As Label = ilgen.DefineLabel()
```

```
        ilgen.Emit(OpCodes.Ldarg_0)
        ilgen.Emit(OpCodes.Stloc_S, input)

        ' Create an instance of TOutput, using the generic method
        ' overload of the Activator.CreateInstance method.
        ' Using this overload requires the specified type to have
        ' a parameterless constructor, which is the reason for adding
        ' that constraint to TOutput. Create the constructed generic
        ' method by passing TOutput to MakeGenericMethod. After
        ' emitting code to call the method, emit code to store the
        ' new TOutput in a local variable.
        '
        Dim createInst As MethodInfo = _
            GetType(Activator).GetMethod("CreateInstance", Type.EmptyTypes)
        Dim createInstOfTOutput As MethodInfo = _
            createInst.MakeGenericMethod(TOutput)

        ilgen.Emit(OpCodes.Call, createInstOfTOutput)
        ilgen.Emit(OpCodes.Stloc_S, retVal)

        ' Load the reference to the TOutput object, cast it to
        ' ICollection(Of TInput), and save it.
        ilgen.Emit(OpCodes.Ldloc_S, retVal)
        ilgen.Emit(OpCodes.Box, TOutput)
        ilgen.Emit(OpCodes.Castclass, icollOfTInput)
        ilgen.Emit(OpCodes.Stloc_S, ic)

        ' Loop through the array, adding each element to the new
        ' instance of TOutput. Note that in order to get a MethodInfo
        ' for ICollection(Of TInput).Add, it is necessary to first
        ' get the Add method for the generic type defintion,
        ' ICollection(Of T).Add. This is because it is not possible
        ' to call GetMethod on icollOfTInput. The static overload of
        ' TypeBuilder.GetMethod produces the correct MethodInfo for
        ' the constructed type.
        '
        Dim mAddPrep As MethodInfo = icoll.GetMethod("Add")
        Dim mAdd As MethodInfo = _
            TypeBuilder.GetMethod(icollOfTInput, mAddPrep)

        ' Initialize the count and enter the loop.
        ilgen.Emit(OpCodes.Ldc_I4_0)
        ilgen.Emit(OpCodes.Stloc_S, index)
        ilgen.Emit(OpCodes.Br_S, enterLoop)

        ' Mark the beginning of the loop. Push the ICollection
        ' reference on the stack, so it will be in position for the
        ' call to Add. Then push the array and the index on the
        ' stack, get the array element, and call Add (represented
        ' by the MethodInfo mAdd) to add it to the collection.
        '
        ' The other ten instructions just increment the index
        ' and test for the end of the loop. Note the MarkLabel
        ' method, which sets the point in the code where the
```

```
' loop is entered. (See the earlier Br_S to enterLoop.)
'
ilgen.MarkLabel(loopAgain)

ilgen.Emit(OpCodes.Ldloc_S, ic)
ilgen.Emit(OpCodes.Ldloc_S, input)
ilgen.Emit(OpCodes.Ldloc_S, index)
ilgen.Emit(OpCodes.Ldelem, TInput)
ilgen.Emit(OpCodes.Callvirt, mAdd)

ilgen.Emit(OpCodes.Ldloc_S, index)
ilgen.Emit(OpCodes.Ldc_I4_1)
ilgen.Emit(OpCodes.Add)
ilgen.Emit(OpCodes.Stloc_S, index)

ilgen.MarkLabel(enterLoop)
ilgen.Emit(OpCodes.Ldloc_S, index)
ilgen.Emit(OpCodes.Ldloc_S, input)
ilgen.Emit(OpCodes.Ldlen)
ilgen.Emit(OpCodes.Conv_I4)
ilgen.Emit(OpCodes.Clt)
ilgen.Emit(OpCodes.Brtrue_S, loopAgain)

ilgen.Emit(OpCodes.Ldloc_S, retVal)
ilgen.Emit(OpCodes.Ret)

' Complete the type.
Dim dt As Type = demoType.CreateType()
' Save the assembly, so it can be examined with Ildasm.exe.
demoAssembly.Save(asmName.Name & ".dll")

' To create a constructed generic method that can be
' executed, first call the GetMethod method on the completed
' type to get the generic method definition. Call MakeGenericType
' on the generic method definition to obtain the constructed
' method, passing in the type arguments. In this case, the
' constructed method has String for TInput and List(Of String)
' for TOutput.
'
Dim m As MethodInfo = dt.GetMethod("Factory")
Dim bound As MethodInfo = m.MakeGenericMethod( _
    GetType(String), GetType(List(Of String)))

' Display a string representing the bound method.
Console.WriteLine(bound)


' Once the generic method is constructed,
' you can invoke it and pass in an array of objects
' representing the arguments. In this case, there is only
' one element in that array, the argument 'arr'.
'
Dim o As Object = bound.Invoke(Nothing, New Object() { arr })
Dim list2 As List(Of String) = CType(o, List(Of String))
```

```
        Console.WriteLine("The first element is: {0}", list2(0))


        ' You can get better performance from multiple calls if
        ' you bind the constructed method to a delegate. The
        ' following code uses the generic delegate D defined
        ' earlier.
        '
        Dim dType As Type = GetType(D(Of String, List(Of String)))
        Dim test As D(Of String, List(Of String))
        test = CType( _
            [Delegate].CreateDelegate(dType, bound), _
            D(Of String, List(Of String)))

        Dim list3 As List(Of String) = test(arr)
        Console.WriteLine("The first element is: {0}", list3(0))

    End Sub
End Class

' This code example produces the following output:
'
'The first element is: a
'System.Collections.Generic.List`1[System.String] Factory[String,List`1](System.String[])
'The first element is: a
'The first element is: a
```

Compiling the Code

- The code contains the C# **using** statements (**Imports** in Visual Basic) necessary for compilation.
- No additional assembly references are required.
- Compile the code at the command line using csc.exe, vbc.exe, or cl.exe. To compile the code in Visual Studio, place it in a console application project template.

See Also

MethodBuilder
How to: Define a Generic Type with Reflection Emit

Dynamic Language Runtime Overview

.NET Framework (current version)

The *dynamic language runtime* (DLR) is a runtime environment that adds a set of services for dynamic languages to the common language runtime (CLR). The DLR makes it easier to develop dynamic languages to run on the .NET Framework and to add dynamic features to statically typed languages.

Dynamic languages can identify the type of an object at run time, whereas in statically typed languages such as C# and Visual Basic (when you use **Option Explicit On**) you must specify object types at design time. Examples of dynamic languages are Lisp, Smalltalk, JavaScript, PHP, Ruby, Python, ColdFusion, Lua, Cobra, and Groovy.

Most dynamic languages provide the following advantages for developers:

- The ability to use a rapid feedback loop (REPL, or read-evaluate-print loop). This lets you enter several statements and immediately execute them to see the results.
- Support for both top-down development and more traditional bottom-up development. For example, when you use a top-down approach, you can call functions that are not yet implemented and then add underlying implementations when you need them.
- Easier refactoring and code modifications, because you do not have to change static type declarations throughout the code.

Dynamic languages make excellent scripting languages. Customers can easily extend applications created by using dynamic languages with new commands and functionality. Dynamic languages are also frequently used for creating Web sites and test harnesses, maintaining server farms, developing various utilities, and performing data transformations.

The purpose of the DLR is to enable a system of dynamic languages to run on the .NET Framework and give them .NET interoperability. The DLR introduces dynamic objects to C# and Visual Basic in Visual Studio 2010 to support dynamic behavior in these languages and enable their interoperation with dynamic languages.

The DLR also helps you create libraries that support dynamic operations. For example, if you have a library that uses XML or JavaScript Object Notation (JSON) objects, your objects can appear as dynamic objects to languages that use the DLR. This lets library users write syntactically simpler and more natural code for operating with objects and accessing object members.

For example, you might use the following code to increment a counter in XML in C#.

```
Scriptobj.SetProperty("Count", ((int)GetProperty("Count")) + 1);
```

By using the DLR, you could use the following code instead for the same operation.

```
scriptobj.Count += 1;
```

Like the CLR, the DLR is a part of the .NET Framework and is provided with the .NET Framework and Visual Studio installation packages. The open-source version of the DLR is also available for download on the CodePlex Web site.

Note

The open-source version of the DLR has all the features of the DLR that is included in Visual Studio and the .NET Framework. It also provides additional support for language implementers. For more information, see the documentation on the CodePlex Web site.

Examples of languages developed by using the DLR include the following:

- IronPython. Available as open-source software from the CodePlex Web site.
- IronRuby. Available as open-source software from the RubyForge Web site.

Primary DLR Advantages

The DLR provides the following advantages.

Simplifies Porting Dynamic Languages to the .NET Framework

The DLR allows language implementers to avoid creating lexical analyzers, parsers, semantic analyzers, code generators, and other tools that they traditionally had to create themselves. To use the DLR, a language needs to produce *expression trees*, which represent language-level code in a tree-shaped structure, runtime helper routines, and optional dynamic objects that implement the IDynamicMetaObjectProvider interface. The DLR and the .NET Framework automate a lot of code analysis and code generation tasks. This enables language implementers to concentrate on unique language features.

Enables Dynamic Features in Statically Typed Languages

Existing .NET Framework languages such as C# and Visual Basic can create dynamic objects and use them together with statically typed objects. For example, C# and Visual Basic can use dynamic objects for HTML, Document Object Model (DOM), and .NET reflection.

Provides Future Benefits of the DLR and .NET Framework

Languages implemented by using the DLR can benefit from future DLR and .NET Framework improvements. For example, if the .NET Framework releases a new version that has an improved garbage collector or faster assembly loading time, languages implemented by using the DLR immediately get the same benefit. If the DLR adds optimizations such as better compilation, the performance also improves for all languages implemented by using the DLR.

Enables Sharing of Libraries and Objects

The objects and libraries implemented in one language can be used by other languages. The DLR also enables interoperation between statically typed and dynamic languages. For example, C# can declare a dynamic object that uses a library that is written in a dynamic language. At the same time, dynamic languages can use libraries from the

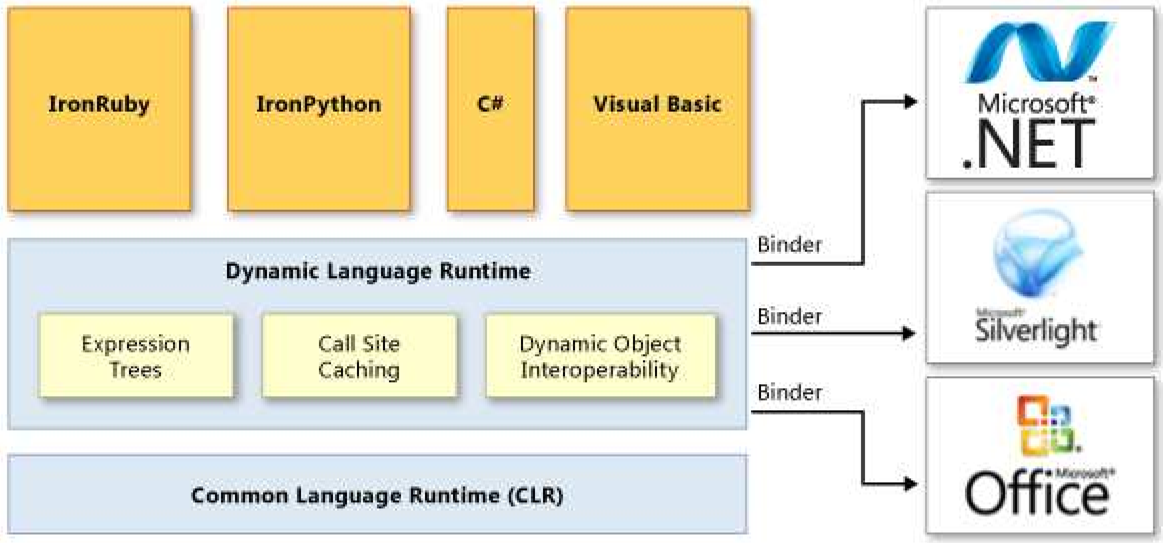
.NET Framework.

Provides Fast Dynamic Dispatch and Invocation

The DLR provides fast execution of dynamic operations by supporting advanced polymorphic caching. The DLR creates rules for binding operations that use objects to the necessary runtime implementations and then caches these rules to avoid resource-exhausting binding computations during successive executions of the same code on the same types of objects.

DLR Architecture

The following illustration shows the architecture of the dynamic language runtime.



DLR architecture

The DLR adds a set of services to the CLR for better supporting dynamic languages. These services include the following:

- Expression trees. The DLR uses expression trees to represent language semantics. For this purpose, the DLR has extended LINQ expression trees to include control flow, assignment, and other language-modeling nodes. For more information, see Expression Trees (C# and Visual Basic).
- Call site caching. A *dynamic call site* is a place in the code where you perform an operation like `a + b` or `a.b()` on dynamic objects. The DLR caches the characteristics of `a` and `b` (usually the types of these objects) and information about the operation. If such an operation has been performed previously, the DLR retrieves all the necessary information from the cache for fast dispatch.
- Dynamic object interoperability. The DLR provides a set of classes and interfaces that represent dynamic objects and operations and can be used by language implementers and authors of dynamic libraries. These classes and interfaces include IDynamicMetaObjectProvider, DynamicMetaObject, DynamicObject, and ExpandoObject.

The DLR uses binders in call sites to communicate not only with the .NET Framework, but with other infrastructures and services, including Silverlight and COM. Binders encapsulate a language's semantics and specify how to perform operations in a call site by using expression trees. This enables dynamic and statically typed languages that use the DLR to share libraries and gain access to all the technologies that the DLR supports.

DLR Documentation

For more information about how to use the open source version of the DLR to add dynamic behavior to a language, or about how to enable the use of a dynamic language with the .NET Framework, see the documentation on the CodePlex Web site.

See Also

ExpandoObject
DynamicObject
Common Language Runtime (CLR)
Expression Trees (C# and Visual Basic)
Walkthrough: Creating and Using Dynamic Objects (C# and Visual Basic)

Dynamic Source Code Generation and Compilation

.NET Framework (current version)

The .NET Framework includes a mechanism called the Code Document Object Model (CodeDOM) that enables developers of programs that emit source code to generate source code in multiple programming languages at run time, based on a single model that represents the code to render.

To represent source code, CodeDOM elements are linked to each other to form a data structure known as a CodeDOM graph, which models the structure of some source code.

The **System.CodeDom** namespace defines types that can represent the logical structure of source code, independent of a specific programming language. The **System.CodeDom.Compiler** namespace defines types for generating source code from CodeDOM graphs and managing the compilation of source code in supported languages. Compiler vendors or developers can extend the set of supported languages.

Language-independent source code modeling can be valuable when a program needs to generate source code for a program model in multiple languages or for an uncertain target language. For example, some designers use the CodeDOM as a language abstraction interface to produce source code in the correct programming language, if CodeDOM support for the language is available.

The .NET Framework includes code generators and code compilers for C#, JScript, and Visual Basic.

In This Section

Using the CodeDOM
    Describes common uses, and demonstrates building a simple object graph using the CodeDOM.

Generating Source Code and Compiling a Program from a CodeDOM Graph
    Describes how to generate source code and compile the generated code with an external compiler using classes defined in the **System.CodeDom.Compiler** namespace.

How to: Create an XML Documentation File Using CodeDOM
    Describes how to use CodeDOM to generate code with XML documentation comments, and compile the generated code so that it creates the XML documentation output.

How to: Create a Class Using CodeDOM
    Describes how to use CodeDOM to generate a class containing fields, properties, a method, a constructor, and an entry point.

Reference

System.CodeDom
    Defines elements that represent code elements in programming languages that target the common language runtime.

System.CodeDom.Compiler
Defines interfaces for generating and compiling code at run time.

Related Sections

CodeDOM Quick Reference
Provides a quick way for developers to find the CodeDOM elements that represent source code elements.

Using the CodeDOM

.NET Framework (current version)

The CodeDOM provides types that represent many common types of source code elements. You can design a program that builds a source code model using CodeDOM elements to assemble an object graph. This object graph can be rendered as source code using a CodeDOM code generator for a supported programming language. The CodeDOM can also be used to compile source code into a binary assembly.

Some common uses for the CodeDOM include:

- Templated code generation: generating code for ASP.NET, XML Web services client proxies, code wizards, designers, or other code-emitting mechanisms.
- Dynamic compilation: supporting code compilation in single or multiple languages.

Building a CodeDOM Graph

The System.CodeDom namespace provides classes for representing the logical structure of source code, independent of language syntax.

The Structure of a CodeDOM Graph

The structure of a CodeDOM graph is like a tree of containers. The top-most, or root, container of each compilable CodeDOM graph is a CodeCompileUnit. Every element of your source code model must be linked into the graph through a property of a CodeObject in the graph.

Building a Source Code Model for a Sample Hello World Program

The following walkthrough provides an example of how to build a CodeDOM object graph that represents the code for a simple Hello World application. For the complete source code for this code example, see the System.CodeDom.Compiler.CodeDomProvider topic.

Creating a compile unit

The CodeDOM defines an object called a CodeCompileUnit, which can reference a CodeDOM object graph that models the source code to compile. A **CodeCompileUnit** has properties for storing references to attributes, namespaces, and assemblies.

The CodeDom providers that derive from the CodeDomProvider class contain methods that process the object graph referenced by a **CodeCompileUnit**.

To create an object graph for a simple application, you must assemble the source code model and reference it from a **CodeCompileUnit**.

You can create a new compile unit with the syntax demonstrated in this example:

VB

```
Dim compileUnit As New CodeCompileUnit()
```

A CodeSnippetCompileUnit can contain a section of source code that is already in the target language, but cannot be rendered to another language.

Defining a namespace

To define a namespace, create a CodeNamespace and assign a name for it using the appropriate constructor or by setting its **Name** property.

VB

```
Dim samples As New CodeNamespace("Samples")
```

Importing a namespace

To add a namespace import directive to the namespace, add a CodeNamespaceImport that indicates the namespace to import to the **CodeNamespace.Imports** collection.

The following code adds an import for the **System** namespace to the **Imports** collection of a **CodeNamespace** named `samples`:

VB

```
samples.Imports.Add(new CodeNamespaceImport("System"))
```

Linking code elements into the object graph

All code elements that form a CodeDOM graph must be linked to the CodeCompileUnit that is the root element of the tree by a series of references between elements directly referenced from the properties of the root object of the graph. Set an object to a property of a container object to establish a reference from the container object.

The following statement adds the `samples` **CodeNamespace** to the **Namespaces** collection property of the root **CodeCompileUnit**.

VB

```
compileUnit.Namespaces.Add(samples)
```

Defining a type

To declare a class, structure, interface, or enumeration using the CodeDOM, create a new CodeTypeDeclaration, and assign it a name. The following example demonstrates this using a constructor overload to set the **Name** property:

VB

```
Dim class1 As New CodeTypeDeclaration("Class1")
```

To add a type to a namespace, add a CodeTypeDeclaration that represents the type to add to the namespace to the **Types** collection of a **CodeNamespace**.

The following example demonstrates how to add a class named `class1` to a **CodeNamespace** named `samples`:

VB

```
samples.Types.Add(class1)
```

Adding class members to a class

The System.CodeDom namespace provides a variety of elements that can be used to represent class members. Each class member can be added to the **Members** collection of a CodeTypeDeclaration.

Defining a code entry point method for an executable

If you are building code for an executable program, it is necessary to indicate the entry point of a program by creating a CodeEntryPointMethod to represent the method at which program execution should begin.

The following example demonstrates how to define an entry point method that contains a CodeMethodInvokeExpression that calls **System.Console.WriteLine** to print "Hello World!":

VB

```
Dim start As New CodeEntryPointMethod()
Dim cs1 As New CodeMethodInvokeExpression( _
    New CodeTypeReferenceExpression("System.Console"), _
    "WriteLine", new CodePrimitiveExpression("Hello World!"))
start.Statements.Add(cs1)
```

The following statement adds the entry point method named `Start` to the **Members** collection of `class1`:

VB

```
class1.Members.Add( start)
```

Now the CodeCompileUnit named `compileUnit` contains the CodeDOM graph for a simple Hello World program. For information on generating and compiling code from a CodeDOM graph, see Generating Source Code and Compiling a Program from a CodeDOM Graph.

More information on building a CodeDOM graph

The CodeDOM supports the many common types of code elements found in programming languages that support the common language runtime. The CodeDOM was not designed to provide elements to represent all possible programming language features. Code that cannot be represented easily with CodeDOM elements can be encapsulated in a CodeSnippetExpression, a CodeSnippetStatement, a CodeSnippetTypeMember, or a CodeSnippetCompileUnit. However, snippets cannot be translated to other languages automatically by the CodeDOM.

For documentation for the each of the CodeDOM types, see the reference documentation for the System.CodeDom namespace.

For a quick chart to locate the CodeDOM element that represents a specific type of code element, see the CodeDOM Quick Reference.

Generating and Compiling Source Code from a CodeDOM Graph

.NET Framework (current version)

The System.CodeDom.Compiler namespace provides interfaces for generating source code from CodeDOM object graphs and for managing compilation with supported compilers. A code provider can produce source code in a particular programming language according to a CodeDOM graph. A class that derives from CodeDomProvider can typically provide methods for generating and compiling code for the language the provider supports.

Using a CodeDOM code provider to generate source code

To generate source code in a particular language, you need a CodeDOM graph that represents the structure of the source code to generate.

The following example demonstrate how to create an instance of a CSharpCodeProvider:

VB

```
Dim provider As New CSharpCodeProvider()
```

The graph for code generation is typically contained in a CodeCompileUnit. To generate code for a **CodeCompileUnit** that contains a CodeDOM graph, call the GenerateCodeFromCompileUnit method of the code provider. This method has a parameter for a TextWriter that it uses to generate the source code, so it is sometimes necessary to first create a **TextWriter** that can be written to. The following example demonstrates generating code from a **CodeCompileUnit** and writing the generated source code to a file named HelloWorld.cs.

VB

```
Public Shared Function GenerateCSharpCode(compileunit As CodeCompileUnit) As String
    ' Generate the code with the C# code provider.
    Dim provider As New CSharpCodeProvider()

    ' Build the output file name.
    Dim sourceFile As String
    If provider.FileExtension(0) = "." Then
       sourceFile = "HelloWorld" + provider.FileExtension
    Else
       sourceFile = "HelloWorld." + provider.FileExtension
    End If

    ' Create a TextWriter to a StreamWriter to the output file.
    Using sw As New StreamWriter(sourceFile, false)
        Dim tw As New IndentedTextWriter(sw, "    ")

        ' Generate source code Imports the code provider.
        provider.GenerateCodeFromCompileUnit(compileunit, tw, _
            New CodeGeneratorOptions())

        ' Close the output file.
        tw.Close()
    End Using

    Return sourceFile
End Function
```

Using a CodeDOM code provider to compile assemblies

Invoking compilation

To compile an assembly using a CodeDom provider, you must have either source code to compile in a language for which you have a compiler, or a CodeDOM graph that source code to compile can be generated from.

If you are compiling from a CodeDOM graph, pass the CodeCompileUnit containing the graph to the CompileAssemblyFromDom method of the code provider. If you have a source code file in a language that the compiler understands, pass the name of the file containing the source code to the CompileAssemblyFromFile method of the CodeDom provider. You can also pass a string containing source code in a language that the compiler understands to the CompileAssemblyFromSource method of the CodeDom provider.

Configuring compilation parameters

All of the standard compilation-invoking methods of a CodeDom provider have a parameter of type CompilerParameters that indicates the options to use for compilation.

You can specify a file name for the output assembly in the OutputAssembly property of the **CompilerParameters**. Otherwise, a default output file name will be used.

By default, a new **CompilerParameters** is initialized with its GenerateExecutable property set to **false**. If you are compiling an executable program, you must set the **GenerateExecutable** property to **true**. When the **GenerateExecutable** is set to **false**, the compiler will generate a class library.

If you are compiling an executable from a CodeDOM graph, a CodeEntryPointMethod must be defined in the graph. If there are multiple code entry points, it may be necessary to set the MainClass property of the **CompilerParameters** to the name of the class that defines the entry point to use.

To include debug information in a generated executable, set the IncludeDebugInformation property to **true**.

If your project references any assemblies, you must specify the assembly names as items in a StringCollection as the ReferencedAssemblies property of the **CompilerParameters** you use when invoking compilation.

You can compile an assembly that is written to memory rather than disk by setting the GenerateInMemory property to **true**. When an assembly is generated in memory, your code can obtain a reference to the generated assembly from the CompiledAssembly property of a CompilerResults. If an assembly is written to disk, you can obtain the path to the generated assembly from the PathToAssembly property of a **CompilerResults**.

To specify a custom command-line arguments string to use when invoking the compilation process, set the string in the CompilerOptions property.

If a Win32 security token is required to invoke the compiler process, specify the token in the UserToken property.

To link a Win32 resource file into the compiled assembly, specify the name of the Win32 resource file in the Win32Resource property.

To specify a warning level at which to halt compilation, set the WarningLevel property to an integer that represents the warning level at which to halt compilation. You can also configure the compiler to halt compilation if warnings are encountered by setting the TreatWarningsAsErrors property to **true**.

The following code example demonstrates compiling a source file using a CodeDom provider derived from the CodeDomProvider class.

VB

```
Public Shared Function CompileCSharpCode(sourceFile As String, _
    exeFile As String) As Boolean
    Dim provider As New CSharpCodeProvider()

    ' Build the parameters for source compilation.
    Dim cp As New CompilerParameters()

    ' Add an assembly reference.
    cp.ReferencedAssemblies.Add( "System.dll" )

    ' Generate an executable instead of
    ' a class library.
    cp.GenerateExecutable = true

    ' Set the assembly file name to generate.
    cp.OutputAssembly = exeFile

    ' Save the assembly as a physical file.
    cp.GenerateInMemory = false

    ' Invoke compilation.
    Dim cr As CompilerResults = provider.CompileAssemblyFromFile(cp, sourceFile)

    If cr.Errors.Count > 0 Then
        ' Display compilation errors.
         Console.WriteLine("Errors building {0} into {1}", _
             sourceFile, cr.PathToAssembly)
        For Each ce As CompilerError In cr.Errors
            Console.WriteLine("  {0}", ce.ToString())
            Console.WriteLine()
        Next ce
    Else
        Console.WriteLine("Source {0} built into {1} successfully.", _
            sourceFile, cr.PathToAssembly)
    End If

    ' Return the results of compilation.
    If cr.Errors.Count > 0 Then
        Return False
    Else
        Return True
    End If
    End Function
```

Languages with Initial Support

The .NET Framework provides code compilers and code generators for the following languages: C#, Visual Basic, C++, and JScript. CodeDOM support can be extended to other languages by implementing language-specific code generators and code compilers.

See Also

System.CodeDom
System.CodeDom.Compiler
Dynamic Source Code Generation and Compilation
CodeDOM Quick Reference

How to: Create an XML Documentation File Using CodeDOM

.NET Framework (current version)

CodeDOM can be used to create code that generates XML documentation. The process involves creating the CodeDOM graph that contains the XML documentation comments, generating the code, and compiling the generated code with the compiler option that creates the XML documentation output.

To create a CodeDOM graph that contains XML documentation comments

1. Create a CodeCompileUnit containing the CodeDOM graph for the sample application.

2. Use the CodeCommentStatement constructor with the *docComment* parameter set to **true** to create the XML documentation comment elements and text.

VB

```vb
Dim class1 As New CodeTypeDeclaration("Class1")

class1.Comments.Add(New CodeCommentStatement("<summary>", True))
class1.Comments.Add(New CodeCommentStatement( _
    "Create a Hello World application.", True))
class1.Comments.Add(New CodeCommentStatement( _
    "<seealso cref=" & ControlChars.Quote & "Class1.Main" & _
    ControlChars.Quote & "/>", True))
class1.Comments.Add(New CodeCommentStatement("</summary>", True))

' Add the new type to the namespace type collection.
samples.Types.Add(class1)

' Declare a new code entry point method.
Dim start As New CodeEntryPointMethod()
start.Comments.Add(New CodeCommentStatement("<summary>", True))
start.Comments.Add(New CodeCommentStatement( _
    "Main method for HelloWorld application.", True))
start.Comments.Add(New CodeCommentStatement( _
    "<para>Add a new paragraph to the description.</para>", True))
start.Comments.Add(New CodeCommentStatement("</summary>", True))
```

To generate the code from the CodeCompileUnit

1. Use the GenerateCodeFromCompileUnit method to generate the code and create a source file to be compiled.

VB

```
Dim sourceFile As New StreamWriter(sourceFileName)

LogMessage("Generating code...")
provider.GenerateCodeFromCompileUnit(cu, sourceFile, Nothing)
sourceFile.Close()
```

To compile the code and generate the documentation file

1. Add the **/doc** compiler option to the CompilerOptions property of a CompilerParameters object and pass the object to the CompileAssemblyFromFile method to create the XML documentation file when the code is compiled.

VB

```
Dim opt As New CompilerParameters(New String() {"System.dll"})
opt.GenerateExecutable = True
opt.OutputAssembly = "HelloWorld.exe"
opt.TreatWarningsAsErrors = True
opt.IncludeDebugInformation = True
opt.GenerateInMemory = True
opt.CompilerOptions = "/doc"

Dim results As CompilerResults

LogMessage(("Compiling with " & providerName))
results = provider.CompileAssemblyFromFile(opt, sourceFileName)
```

Example

The following code example creates a CodeDOM graph with documentation comments, generates a code file from the graph, and compiles the file and creates an associated XML documentation file.

VB

```
Imports System
Imports System.CodeDom
Imports System.CodeDom.Compiler
Imports System.IO
Imports System.Text.RegularExpressions



Class Program
    Private Shared providerName As String = "vb"
    Private Shared sourceFileName As String = "test.vb"

    Shared Sub Main(ByVal args() As String)
        Dim provider As CodeDomProvider = _
            CodeDomProvider.CreateProvider(providerName)
        LogMessage("Building CodeDOM graph...")

        Dim cu As New CodeCompileUnit()

        cu = BuildHelloWorldGraph()


        Dim sourceFile As New StreamWriter(sourceFileName)

        LogMessage("Generating code...")
        provider.GenerateCodeFromCompileUnit(cu, sourceFile, Nothing)
        sourceFile.Close()

        Dim opt As New CompilerParameters(New String() {"System.dll"})
        opt.GenerateExecutable = True
        opt.OutputAssembly = "HelloWorld.exe"
        opt.TreatWarningsAsErrors = True
        opt.IncludeDebugInformation = True
        opt.GenerateInMemory = True
        opt.CompilerOptions = "/doc"

        Dim results As CompilerResults

        LogMessage(("Compiling with " & providerName))
        results = provider.CompileAssemblyFromFile(opt, sourceFileName)

        OutputResults(results)
        If results.NativeCompilerReturnValue <> 0 Then
            LogMessage("")
            LogMessage("Compilation failed.")
        Else
            LogMessage("")
            LogMessage("Demo completed successfully.")
        End If
        File.Delete(sourceFileName)

    End Sub 'Main


    ' Build a Hello World program graph using
    ' System.CodeDom types.
    Public Shared Function BuildHelloWorldGraph() As CodeCompileUnit
        ' Create a new CodeCompileUnit to contain
        ' the program graph.
        Dim compileUnit As New CodeCompileUnit()

        ' Declare a new namespace called Samples.
        Dim samples As New CodeNamespace("Samples")
        ' Add the new namespace to the compile unit.
        compileUnit.Namespaces.Add(samples)

        ' Add the new namespace import for the System namespace.
        samples.Imports.Add(New CodeNamespaceImport("System"))
```

```
' Declare a new type called Class1.
Dim class1 As New CodeTypeDeclaration("Class1")

class1.Comments.Add(New CodeCommentStatement("<summary>", True))
class1.Comments.Add(New CodeCommentStatement( _
    "Create a Hello World application.", True))
class1.Comments.Add(New CodeCommentStatement( _
    "<seealso cref=" & ControlChars.Quote & "Class1.Main" & _
    ControlChars.Quote & "/>", True))
class1.Comments.Add(New CodeCommentStatement("</summary>", True))

' Add the new type to the namespace type collection.
samples.Types.Add(class1)

' Declare a new code entry point method.
Dim start As New CodeEntryPointMethod()
start.Comments.Add(New CodeCommentStatement("<summary>", True))
start.Comments.Add(New CodeCommentStatement( _
    "Main method for HelloWorld application.", True))
start.Comments.Add(New CodeCommentStatement( _
    "<para>Add a new paragraph to the description.</para>", True))
start.Comments.Add(New CodeCommentStatement("</summary>", True))
' Create a type reference for the System.Console class.
Dim csSystemConsoleType As New CodeTypeReferenceExpression( _
    "System.Console")

' Build a Console.WriteLine statement.
Dim cs1 As New CodeMethodInvokeExpression(csSystemConsoleType, _
    "WriteLine", New CodePrimitiveExpression("Hello World!"))

' Add the WriteLine call to the statement collection.
start.Statements.Add(cs1)

' Build another Console.WriteLine statement.
Dim cs2 As New CodeMethodInvokeExpression(csSystemConsoleType, _
    "WriteLine", New CodePrimitiveExpression( _
    "Press the ENTER key to continue."))

' Add the WriteLine call to the statement collection.
start.Statements.Add(cs2)

' Build a call to System.Console.ReadLine.
Dim csReadLine As New CodeMethodInvokeExpression( _
    csSystemConsoleType, "ReadLine")

' Add the ReadLine statement.
start.Statements.Add(csReadLine)

' Add the code entry point method to
' the Members collection of the type.
class1.Members.Add(start)

Return compileUnit
```

```
    End Function 'BuildHelloWorldGraph

    Shared Sub LogMessage(ByVal [text] As String)
        Console.WriteLine([text])

    End Sub 'LogMessage


    Shared Sub OutputResults(ByVal results As CompilerResults)
        LogMessage(("NativeCompilerReturnValue=" & _
            results.NativeCompilerReturnValue.ToString()))
        Dim s As String
        For Each s In results.Output
            LogMessage(s)
        Next s

    End Sub 'OutputResults
End Class 'Program
```

The code example creates the following XML documentation in the HelloWorldDoc.xml file.

```
<?xml version="1.0" ?>
<doc>
  <assembly>
    <name>HelloWorld</name>
  </assembly>
  <members>
    <member name="T:Samples.Class1">
      <summary>
        Create a Hello World application.
        <seealso cref="M:Samples.Class1.Main" />
      </summary>
    </member>
    <member name="M:Samples.Class1.Main">
      <summary>
        Main method for HelloWorld application.
        <para>Add a new paragraph to the description.</para>
      </summary>
    </member>
  </members>
</doc>
```

Compiling the Code

- This code example requires the **FullTrust** permission set to execute successfully.

See Also

Documenting Your Code with XML (Visual Basic)
XML Documentation Comments (C# Programming Guide)
XML Documentation (Visual C++)

How to: Create a Class Using CodeDOM

.NET Framework (current version)

The following procedures illustrate how to create and compile a CodeDOM graph that generates a class containing two fields, three properties, a method, a constructor, and an entry point.

1. Create a console application that will use CodeDOM code to generate the source code for a class.

   In this example, the generating class is named `Sample`, and the generated code is a class named `CodeDOMCreatedClass` in a file named SampleCode.

2. In the generating class, initialize the CodeDOM graph and use CodeDOM methods to define the members, constructor, and entry point (`Main` method) of the generated class.

   In this example, the generated class has two fields, three properties, a constructor, a method, and a `Main` method.

3. In the generating class, create a language-specific code provider and call its GenerateCodeFromCompileUnit method to generate the code from the graph.

4. Compile and execute the application to generate the code.

   In this example, the generated code is in a file named SampleCode. Compile and execute that code to see the sample output.

To create the application that will execute the CodeDOM code

- Create a console application class to contain the CodeDOM code. Define the global fields that are to be used in the class to reference the assembly (CodeCompileUnit) and class (CodeTypeDeclaration), specify the name of the generated source file, and declare the `Main` method.

VB

```vb
Imports System
Imports System.Reflection
Imports System.IO
Imports System.CodeDom
Imports System.CodeDom.Compiler
Imports Microsoft.CSharp



Class Sample
    Private targetUnit As CodeCompileUnit
    Private targetClass As CodeTypeDeclaration
    Private Const outputFileName As String = "SampleCode.vb"
```

```
        Shared Sub Main(ByVal args() As String)

        End Sub 'Main
    End Class 'Sample
```

To initialize the CodeDOM graph

- In the constructor for the console application class, initialize the assembly and class, and add the appropriate declarations to the CodeDOM graph.

VB

```
    Public Sub New()
        targetUnit = New CodeCompileUnit()
        Dim samples As New CodeNamespace("CodeDOMSample")
        samples.Imports.Add(New CodeNamespaceImport("System"))
        targetClass = New CodeTypeDeclaration("CodeDOMCreatedClass")
        targetClass.IsClass = True
        targetClass.TypeAttributes = _
            TypeAttributes.Public Or TypeAttributes.Sealed
        samples.Types.Add(targetClass)
        targetUnit.Namespaces.Add(samples)

    End Sub 'NewNew
```

To add members to the CodeDOM graph

- Add fields to the CodeDOM graph by adding CodeMemberField objects to the Members property of the class.

VB

```
    Public Sub AddFields()
        ' Declare the widthValue field.
        Dim widthValueField As New CodeMemberField()
        widthValueField.Attributes = MemberAttributes.Private
        widthValueField.Name = "widthValue"
        widthValueField.Type = _
            New CodeTypeReference(GetType(System.Double))
        widthValueField.Comments.Add(New CodeCommentStatement( _
            "The width of the object."))
        targetClass.Members.Add(widthValueField)

        ' Declare the heightValue field
        Dim heightValueField As New CodeMemberField()
        heightValueField.Attributes = MemberAttributes.Private
        heightValueField.Name = "heightValue"
        heightValueField.Type = _
            New CodeTypeReference(GetType(System.Double))
        heightValueField.Comments.Add(New CodeCommentStatement( _
            "The height of the object."))
        targetClass.Members.Add(heightValueField)

    End Sub 'AddFields
```

- Add properties to the CodeDOM graph by adding CodeMemberProperty objects to the Members property of the class.

VB

```
    Public Sub AddProperties()
        ' Declare the read only Width property.
        Dim widthProperty As New CodeMemberProperty()
        widthProperty.Attributes = _
            MemberAttributes.Public Or MemberAttributes.Final
        widthProperty.Name = "Width"
        widthProperty.HasGet = True
        widthProperty.Type = New CodeTypeReference(GetType(System.Double))
        widthProperty.Comments.Add(New CodeCommentStatement( _
            "The width property for the object."))
        widthProperty.GetStatements.Add(New CodeMethodReturnStatement( _
            New CodeFieldReferenceExpression( _
            New CodeThisReferenceExpression(), "widthValue")))
        targetClass.Members.Add(widthProperty)

        ' Declare the read-only Height property.
        Dim heightProperty As New CodeMemberProperty()
        heightProperty.Attributes = _
            MemberAttributes.Public Or MemberAttributes.Final
        heightProperty.Name = "Height"
        heightProperty.HasGet = True
        heightProperty.Type = New CodeTypeReference(GetType(System.Double))
        heightProperty.Comments.Add(New CodeCommentStatement( _
            "The Height property for the object."))
        heightProperty.GetStatements.Add(New CodeMethodReturnStatement( _
            New CodeFieldReferenceExpression( _
            New CodeThisReferenceExpression(), "heightValue")))
        targetClass.Members.Add(heightProperty)

        ' Declare the read only Area property.
        Dim areaProperty As New CodeMemberProperty()
        areaProperty.Attributes = _
            MemberAttributes.Public Or MemberAttributes.Final
        areaProperty.Name = "Area"
        areaProperty.HasGet = True
        areaProperty.Type = New CodeTypeReference(GetType(System.Double))
        areaProperty.Comments.Add(New CodeCommentStatement( _
            "The Area property for the object."))

        ' Create an expression to calculate the area for the get accessor
        ' of the Area property.
        Dim areaExpression As New CodeBinaryOperatorExpression( _
            New CodeFieldReferenceExpression( _
            New CodeThisReferenceExpression(), "widthValue"), _
            CodeBinaryOperatorType.Multiply, _
            New CodeFieldReferenceExpression( _
            New CodeThisReferenceExpression(), "heightValue"))
        areaProperty.GetStatements.Add( _
            New CodeMethodReturnStatement(areaExpression))
        targetClass.Members.Add(areaProperty)

    End Sub 'AddProperties
```

- Add a method to the CodeDOM graph by adding a CodeMemberMethod object to the Members property of the class.

VB

```
    Public Sub AddMethod()
        ' Declaring a ToString method.
        Dim toStringMethod As New CodeMemberMethod()
        toStringMethod.Attributes = _
            MemberAttributes.Public Or MemberAttributes.Override
        toStringMethod.Name = "ToString"
        toStringMethod.ReturnType = _
            New CodeTypeReference(GetType(System.String))

        Dim widthReference As New CodeFieldReferenceExpression( _
            New CodeThisReferenceExpression(), "Width")
        Dim heightReference As New CodeFieldReferenceExpression( _
            New CodeThisReferenceExpression(), "Height")
        Dim areaReference As New CodeFieldReferenceExpression( _
            New CodeThisReferenceExpression(), "Area")

        ' Declaring a return statement for method ToString.
        Dim returnStatement As New CodeMethodReturnStatement()

        ' This statement returns a string representation of the width,
        ' height, and area.
        Dim formattedOutput As String = "The object:" & Environment.NewLine _
            & " width = {0}," & Environment.NewLine & " height = {1}," _
            & Environment.NewLine & " area = {2}"
        returnStatement.Expression = New CodeMethodInvokeExpression( _
            New CodeTypeReferenceExpression("System.String"), "Format", _
            New CodePrimitiveExpression(formattedOutput), widthReference, _
            heightReference, areaReference)
        toStringMethod.Statements.Add(returnStatement)
        targetClass.Members.Add(toStringMethod)

    End Sub 'AddMethod
```

- Add a constructor to the CodeDOM graph by adding a CodeConstructor object to the Members property of the class.

VB

```
Public Sub AddConstructor()
    ' Declare the constructor
    Dim constructor As New CodeConstructor()
    constructor.Attributes = _
        MemberAttributes.Public Or MemberAttributes.Final

    ' Add parameters.
    constructor.Parameters.Add( _
        New CodeParameterDeclarationExpression( _
        GetType(System.Double), "width"))
    constructor.Parameters.Add( _
        New CodeParameterDeclarationExpression( _
        GetType(System.Double), "height"))

    ' Add field initialization logic
    Dim widthReference As New CodeFieldReferenceExpression( _
        New CodeThisReferenceExpression(), "widthValue")
    constructor.Statements.Add(New CodeAssignStatement( _
        widthReference, New CodeArgumentReferenceExpression("width")))
    Dim heightReference As New CodeFieldReferenceExpression( _
        New CodeThisReferenceExpression(), "heightValue")
    constructor.Statements.Add( _
        New CodeAssignStatement(heightReference, _
        New CodeArgumentReferenceExpression("height")))
    targetClass.Members.Add(constructor)

End Sub 'AddConstructor
```

- Add an entry point to the CodeDOM graph by adding a CodeEntryPointMethod object to the Members property of the class.

VB

```
Public Sub AddEntryPoint()
    Dim start As New CodeEntryPointMethod()
    Dim objectCreate As New CodeObjectCreateExpression( _
        New CodeTypeReference("CodeDOMCreatedClass"), _
        New CodePrimitiveExpression(5.3), _
        New CodePrimitiveExpression(6.9))

    ' Add the statement:
    ' "CodeDOMCreatedClass testClass = _
    '     new CodeDOMCreatedClass(5.3, 6.9);"
    start.Statements.Add(New CodeVariableDeclarationStatement( _
        New CodeTypeReference("CodeDOMCreatedClass"), _
        "testClass", objectCreate))

    ' Creat the expression:
    ' "testClass.ToString()"
    Dim toStringInvoke As New CodeMethodInvokeExpression( _
        New CodeVariableReferenceExpression("testClass"), "ToString")
    ' Add a System.Console.WriteLine statement with the previous
    ' expression as a parameter.
    start.Statements.Add(New CodeMethodInvokeExpression( _
        New CodeTypeReferenceExpression("System.Console"), _
        "WriteLine", toStringInvoke))
    targetClass.Members.Add(start)

End Sub 'AddEntryPoint
```

To generate the code from the CodeDOM graph

- Generate source code from the CodeDOM graph by calling the GenerateCodeFromCompileUnit method.

VB

```
    Public Sub GenerateVBCode(ByVal fileName As String)
        Dim provider As CodeDomProvider
        provider = CodeDomProvider.CreateProvider("VisualBasic")
        Dim options As New CodeGeneratorOptions()
        Dim sourceWriter As New StreamWriter(fileName)
        Try
            provider.GenerateCodeFromCompileUnit( _
                targetUnit, sourceWriter, options)
        Finally
            sourceWriter.Dispose()
        End Try

    End Sub 'GenerateVBCode
```

To create the graph and generate the code

1. Add the methods created in the preceding steps to the `Main` method defined in the first step.

VB

```
        Shared Sub Main()
            Dim sample As New Sample()
            sample.AddFields()
            sample.AddProperties()
            sample.AddMethod()
            sample.AddConstructor()
            sample.AddEntryPoint()
            sample.GenerateVBCode(outputFileName)

        End Sub 'Main
    End Class 'Sample
```

2. Compile and execute the generating class.

Example

The following code example shows the code from the preceding steps.

VB

```
Imports System
Imports System.Reflection
Imports System.IO
Imports System.CodeDom
Imports System.CodeDom.Compiler
Imports Microsoft.VisualBasic

' This code example creates a graph using a CodeCompileUnit and
' generates source code for the graph using the VBCodeProvider.
Class Sample

    ' Define the compile unit to use for code generation.
    Private targetUnit As CodeCompileUnit

    ' The only class in the compile unit. This class contains 2 fields,
    ' 3 properties, a constructor, an entry point, and 1 simple method.
    Private targetClass As CodeTypeDeclaration

    ' The name of the file to contain the source code.
    Private Const outputFileName As String = "SampleCode.vb"

    ' Define the class.
    Public Sub New()
        targetUnit = New CodeCompileUnit()
        Dim samples As New CodeNamespace("CodeDOMSample")
        samples.Imports.Add(New CodeNamespaceImport("System"))
        targetClass = New CodeTypeDeclaration("CodeDOMCreatedClass")
        targetClass.IsClass = True
        targetClass.TypeAttributes = _
            TypeAttributes.Public Or TypeAttributes.Sealed
        samples.Types.Add(targetClass)
        targetUnit.Namespaces.Add(samples)

    End Sub 'NewNew

    ' Adds two fields to the class.
    Public Sub AddFields()
        ' Declare the widthValue field.
        Dim widthValueField As New CodeMemberField()
        widthValueField.Attributes = MemberAttributes.Private
        widthValueField.Name = "widthValue"
        widthValueField.Type = _
            New CodeTypeReference(GetType(System.Double))
        widthValueField.Comments.Add(New CodeCommentStatement( _
            "The width of the object."))
        targetClass.Members.Add(widthValueField)

        ' Declare the heightValue field
        Dim heightValueField As New CodeMemberField()
        heightValueField.Attributes = MemberAttributes.Private
        heightValueField.Name = "heightValue"
        heightValueField.Type = _
            New CodeTypeReference(GetType(System.Double))
        heightValueField.Comments.Add(New CodeCommentStatement( _
            "The height of the object."))
        targetClass.Members.Add(heightValueField)

    End Sub 'AddFields

    ' Add three properties to the class.
    Public Sub AddProperties()
        ' Declare the read only Width property.
        Dim widthProperty As New CodeMemberProperty()
        widthProperty.Attributes = _
            MemberAttributes.Public Or MemberAttributes.Final
        widthProperty.Name = "Width"
        widthProperty.HasGet = True
        widthProperty.Type = New CodeTypeReference(GetType(System.Double))
        widthProperty.Comments.Add(New CodeCommentStatement( _
            "The width property for the object."))
        widthProperty.GetStatements.Add(New CodeMethodReturnStatement( _
            New CodeFieldReferenceExpression( _
            New CodeThisReferenceExpression(), "widthValue")))
        targetClass.Members.Add(widthProperty)

        ' Declare the read-only Height property.
        Dim heightProperty As New CodeMemberProperty()
        heightProperty.Attributes = _
            MemberAttributes.Public Or MemberAttributes.Final
        heightProperty.Name = "Height"
        heightProperty.HasGet = True
        heightProperty.Type = New CodeTypeReference(GetType(System.Double))
        heightProperty.Comments.Add(New CodeCommentStatement( _
            "The Height property for the object."))
        heightProperty.GetStatements.Add(New CodeMethodReturnStatement( _
            New CodeFieldReferenceExpression( _
            New CodeThisReferenceExpression(), "heightValue")))
        targetClass.Members.Add(heightProperty)

        ' Declare the read only Area property.
        Dim areaProperty As New CodeMemberProperty()
        areaProperty.Attributes = _
            MemberAttributes.Public Or MemberAttributes.Final
        areaProperty.Name = "Area"
        areaProperty.HasGet = True
        areaProperty.Type = New CodeTypeReference(GetType(System.Double))
        areaProperty.Comments.Add(New CodeCommentStatement( _
            "The Area property for the object."))

        ' Create an expression to calculate the area for the get accessor
        ' of the Area property.
        Dim areaExpression As New CodeBinaryOperatorExpression( _
            New CodeFieldReferenceExpression( _
            New CodeThisReferenceExpression(), "widthValue"), _
            CodeBinaryOperatorType.Multiply, _
            New CodeFieldReferenceExpression( _
            New CodeThisReferenceExpression(), "heightValue"))
        areaProperty.GetStatements.Add( _
            New CodeMethodReturnStatement(areaExpression))
        targetClass.Members.Add(areaProperty)

    End Sub 'AddProperties

    ' Adds a method to the class. This method multiplies values stored
    ' in both fields.
    Public Sub AddMethod()
        ' Declaring a ToString method.
        Dim toStringMethod As New CodeMemberMethod()
        toStringMethod.Attributes = _
            MemberAttributes.Public Or MemberAttributes.Override
        toStringMethod.Name = "ToString"
        toStringMethod.ReturnType = _
            New CodeTypeReference(GetType(System.String))

        Dim widthReference As New CodeFieldReferenceExpression( _
            New CodeThisReferenceExpression(), "Width")
        Dim heightReference As New CodeFieldReferenceExpression( _
            New CodeThisReferenceExpression(), "Height")
        Dim areaReference As New CodeFieldReferenceExpression( _
            New CodeThisReferenceExpression(), "Area")

        ' Declaring a return statement for method ToString.
        Dim returnStatement As New CodeMethodReturnStatement()

        ' This statement returns a string representation of the width,
        ' height, and area.
        Dim formattedOutput As String = "The object:" & Environment.NewLine _
            & " width = {0}," & Environment.NewLine & " height = {1}," _
            & Environment.NewLine & " area = {2}"
        returnStatement.Expression = New CodeMethodInvokeExpression( _
            New CodeTypeReferenceExpression("System.String"), "Format", _
            New CodePrimitiveExpression(formattedOutput), widthReference, _
            heightReference, areaReference)
        toStringMethod.Statements.Add(returnStatement)
        targetClass.Members.Add(toStringMethod)

    End Sub 'AddMethod

    ' Add a constructor to the class.
    Public Sub AddConstructor()
        ' Declare the constructor
        Dim constructor As New CodeConstructor()
        constructor.Attributes = _
            MemberAttributes.Public Or MemberAttributes.Final

        ' Add parameters.
        constructor.Parameters.Add( _
            New CodeParameterDeclarationExpression( _
            GetType(System.Double), "width"))
        constructor.Parameters.Add( _
            New CodeParameterDeclarationExpression( _
            GetType(System.Double), "height"))

        ' Add field initialization logic
        Dim widthReference As New CodeFieldReferenceExpression( _
            New CodeThisReferenceExpression(), "widthValue")
        constructor.Statements.Add(New CodeAssignStatement( _
            widthReference, New CodeArgumentReferenceExpression("width")))
        Dim heightReference As New CodeFieldReferenceExpression( _
            New CodeThisReferenceExpression(), "heightValue")
        constructor.Statements.Add( _
            New CodeAssignStatement(heightReference, _
            New CodeArgumentReferenceExpression("height")))
        targetClass.Members.Add(constructor)

    End Sub 'AddConstructor

    ' Add an entry point to the class.
    Public Sub AddEntryPoint()
        Dim start As New CodeEntryPointMethod()
        Dim objectCreate As New CodeObjectCreateExpression( _
            New CodeTypeReference("CodeDOMCreatedClass"), _
            New CodePrimitiveExpression(5.3), _
            New CodePrimitiveExpression(6.9))

        ' Add the statement:
        ' "CodeDOMCreatedClass testClass = _
        '     new CodeDOMCreatedClass(5.3, 6.9);"
        start.Statements.Add(New CodeVariableDeclarationStatement( _
            New CodeTypeReference("CodeDOMCreatedClass"), _
            "testClass", objectCreate))

        ' Creat the expression:
        ' "testClass.ToString()"
        Dim toStringInvoke As New CodeMethodInvokeExpression( _
            New CodeVariableReferenceExpression("testClass"), "ToString")

        ' Add a System.Console.WriteLine statement with the previous
        ' expression as a parameter.
        start.Statements.Add(New CodeMethodInvokeExpression( _
            New CodeTypeReferenceExpression("System.Console"), _
            "WriteLine", toStringInvoke))
        targetClass.Members.Add(start)

    End Sub 'AddEntryPoint

    ' Generate Visual Basic source code from the compile unit.
    Public Sub GenerateVBCode(ByVal fileName As String)
        Dim provider As CodeDomProvider
        provider = CodeDomProvider.CreateProvider("VisualBasic")
```

```
            Dim options As New CodeGeneratorOptions()
            Dim sourceWriter As New StreamWriter(fileName)
            Try
                provider.GenerateCodeFromCompileUnit( _
                    targetUnit, sourceWriter, options)
            Finally
                sourceWriter.Dispose()
            End Try

        End Sub 'GenerateVBCode

        ' Create the CodeDOM graph and generate the code.
        Shared Sub Main()
            Dim sample As New Sample()
            sample.AddFields()
            sample.AddProperties()
            sample.AddMethod()
            sample.AddConstructor()
            sample.AddEntryPoint()
            sample.GenerateVBCode(outputFileName)

        End Sub 'Main
    End Class 'Sample
```

When the preceding example is compiled and executed, it produces the following source code.

VB

```
    '------------------------------------------------------------------------------
    ' <auto-generated>
    '     This code was generated by a tool.
    '     Runtime Version:2.0.50727.42
    '
    '     Changes to this file may cause incorrect behavior and will be lost if
    '     the code is regenerated.
    ' </auto-generated>
    '------------------------------------------------------------------------------

    Option Strict Off
    Option Explicit On

    Imports System

    Namespace CodeDOMSample

        Public NotInheritable Class CodeDOMCreatedClass

            'The width of the object.
            Private widthValue As Double

            'The height of the object.
            Private heightValue As Double
            Public Sub New(ByVal width As Double, ByVal height As Double)
                MyBase.New
                Me.widthValue = width
                Me.heightValue = height
            End Sub

            'The width property for the object.
            Public ReadOnly Property Width() As Double
                Get
                    Return Me.widthValue
                End Get
            End Property

            'The Height property for the object.
            Public ReadOnly Property Height() As Double
                Get
                    Return Me.heightValue
                End Get
            End Property

            'The Area property for the object.
            Public ReadOnly Property Area() As Double
                Get
                    Return (Me.widthValue * Me.heightValue)
                End Get
            End Property

            Public Overrides Function ToString() As String
                Return String.Format("The object:"& _
                    Global.Microsoft.VisualBasic.ChrW(13)& _
                    Global.Microsoft.VisualBasic.ChrW(10)& _
                    " width = {0},"&Global.Microsoft.VisualBasic.ChrW(13)& _
                    Global.Microsoft.VisualBasic.ChrW(10)& _
                    " height = {1},"&Global.Microsoft.VisualBasic.ChrW(13)& _
                    Global.Microsoft.VisualBasic.ChrW(10)&" area = {2}", _
                    Me.Width, Me.Height, Me.Area)
            End Function

            Public Shared Sub Main()
                Dim testClass As CodeDOMCreatedClass = _
                    New CodeDOMCreatedClass(5.3, 6.9)
                System.Console.WriteLine(testClass.ToString)
            End Sub
        End Class
    End Namespace
```

The generated source code produces the following output when compiled and executed.

```
The object:
 width = 5.3,
 height = 6.9,
    area = 36.57
```

Compiling the Code

- This code example requires the **FullTrust** permission set to execute successfully.

See Also

Using the CodeDOM
Generating and Compiling Source Code from a CodeDOM Graph