# Chapter 19. Reflection and Metadata

As we saw in Chapter 18, a C# program compiles into an assembly that includes metadata, compiled code, and resources. Inspecting the metadata and compiled code at runtime is called *reflection*.

The compiled code in an assembly contains almost all of the content of the original source code. Some information is lost, such as local variable names, comments, and preprocessor directives. However, reflection can access pretty much everything else, even making it possible to write a decompiler.

Many of the services available in .NET and exposed via C# (such as dynamic binding, serialization, and data binding) depend on the presence of metadata. Your own programs can also take advantage of this metadata and even extend it with new information using custom attributes. The `System.Reflection` namespace houses the reflection API. It is also possible at runtime to dynamically create new metadata and executable instructions in IL via the classes in the `System.Reflection.Emit` namespace.

The examples in this chapter assume that you import the `System` and `System.Reflection` as well as `System.Reflection.Emit` namespaces.

**NOTE**

When we use the term "dynamically" in this chapter, we mean using reflection to perform some task whose type safety is enforced only at runtime. This is similar in principle to *dynamic binding* via C#'s `dynamic` keyword, although the mechanism and functionality are different.

Dynamic binding is much easier to use and employs the Dynamic Language Runtime (DLR) for dynamic language interoperability. Reflection is relatively clumsy to use, but it is more flexible in terms of what you can do with the CLR. For instance, reflection lets you obtain lists of types and members, instantiate an object whose name comes from a string, and build assemblies on the fly.

# Reflecting and Activating Types

In this section, we examine how to obtain a `Type`, inspect its metadata, and use it to dynamically instantiate an object.

## Obtaining a Type

An instance of `System.Type` represents the metadata for a type. Because `Type` is widely used, it lives in the `System` namespace rather than the `System.Reflection` namespace.

You can get an instance of a `System.Type` by calling `GetType` on any object or with C#'s `typeof` operator:

```
Type t1 = DateTime.Now.GetType();     // Type obtained at runtime
Type t2 = typeof (DateTime);          // Type obtained at compile time
```

You can use `typeof` to obtain array types and generic types, as follows:

```
Type t3 = typeof (DateTime[]);          // 1-d Array type
Type t4 = typeof (DateTime[,]);         // 2-d Array type
Type t5 = typeof (Dictionary<int,int>); // Closed generic type
Type t6 = typeof (Dictionary<,>);       // Unbound generic type
```

You can also retrieve a `Type` by name. If you have a reference to its `Assembly`, call `Assembly.GetType` (we describe this further in the section "Reflecting Assemblies"):

```
Type t = Assembly.GetExecutingAssembly().GetType ("Demos.TestProgram");
```

If you don't have an `Assembly` object, you can obtain a type through its *assembly qualified name* (the type's full name followed by the assembly's fully or partially qualified name). The assembly implicitly loads as if you called `Assembly.Load(string)`:

```
Type t = Type.GetType ("System.Int32, System.Private.CoreLib");
```

After you have a `System.Type` object, you can use its properties to access the type's name, assembly, base type, visibility, and so on:

```
Type stringType = typeof (string);
string name     = stringType.Name;          // String
Type baseType   = stringType.BaseType;      // typeof(Object)
Assembly assem  = stringType.Assembly;      // System.Private.CoreLib
bool isPublic   = stringType.IsPublic;      // true
```

A `System.Type` instance is a window into the entire metadata for the type—and the assembly in which it's defined.

> **NOTE**
>
> `System.Type` is abstract, so the `typeof` operator must actually give you a subclass of `Type`. The subclass that the CLR uses is internal to .NET and is called `RuntimeType`.

### TYPEINFO

Should you target .NET Core 1.x (or an older Windows Store profile), you'll find most of `Type`'s members are missing. These missing members are exposed instead on a class called `TypeInfo`, which you obtain by calling `GetTypeInfo`. So, to get our previous example to run, you would do this:

```
Type stringType = typeof(string);
string name = stringType.Name;
Type baseType = stringType.GetTypeInfo().BaseType;
Assembly assem = stringType.GetTypeInfo().Assembly;
bool isPublic = stringType.GetTypeInfo().IsPublic;
```

`TypeInfo` also exists in .NET Core 2 and 3 (and .NET Framework 4.5+, and all .NET Standard versions), so the preceding code works almost universally. `TypeInfo` also includes additional properties and methods for reflecting over members.

### OBTAINING ARRAY TYPES

As we just saw, `typeof` and `GetType` work with array types. You can also obtain an array type by calling `MakeArrayType` on the *element* type:

```
Type simpleArrayType = typeof (int).MakeArrayType();
Console.WriteLine (simpleArrayType == typeof (int[]));     // True
```

You can create multidimensional arrays by passing an integer argument to `MakeArrayType`:

```
Type cubeType = typeof (int).MakeArrayType (3);       // cube shaped
Console.WriteLine (cubeType == typeof (int[,,]));     // True
```

`GetElementType` does the reverse: it retrieves an array type's element type:

```
Type e = typeof (int[]).GetElementType();     // e == typeof (int)
```

`GetArrayRank` returns the number of dimensions of a rectangular array:

```
int rank = typeof (int[,,]).GetArrayRank();   // 3
```

## OBTAINING NESTED TYPES

To retrieve nested types, call `GetNestedTypes` on the containing type:

```
foreach (Type t in typeof (System.Environment).GetNestedTypes())
```

```
  Console.WriteLine (t.FullName);

OUTPUT: System.Environment+SpecialFolder
```

Or:

```
foreach (TypeInfo t in typeof (System.Environment).GetTypeInfo()
                                                   .DeclaredNestedTypes)
  Debug.WriteLine (t.FullName);
```

The one caveat with nested types is that the CLR treats a nested type as having special “nested” accessibility levels:

```
Type t = typeof (System.Environment.SpecialFolder);
Console.WriteLine (t.IsPublic);                      // False
Console.WriteLine (t.IsNestedPublic);                // True
```

## Type Names

A type has `Namespace`, `Name`, and `FullName` properties. In most cases, `FullName` is a composition of the former two:

```
Type t = typeof (System.Text.StringBuilder);

Console.WriteLine (t.Namespace);      // System.Text
Console.WriteLine (t.Name);           // StringBuilder
Console.WriteLine (t.FullName);       // System.Text.StringBuilder
```

There are two exceptions to this rule: nested types and closed generic types.

> **NOTE**
>
> `Type` also has a property called `AssemblyQualifiedName`, which returns `FullName` followed by a comma and then the full name of its assembly. This is the same string that you can pass to `Type.GetType`, and it uniquely identifies a type within the default loading context.

### NESTED TYPE NAMES

With nested types, the containing type appears only in `FullName`:

```
Type t = typeof (System.Environment.SpecialFolder);

Console.WriteLine (t.Namespace);      // System
Console.WriteLine (t.Name);           // SpecialFolder
Console.WriteLine (t.FullName);       //
System.Environment+SpecialFolder
```

The `+` symbol differentiates the containing type from a nested namespace.

### GENERIC TYPE NAMES

Generic type names are suffixed with the `'` symbol, followed by the number of type parameters. If the generic type is unbound, this rule applies to both `Name` and `FullName`:

```
Type t = typeof (Dictionary<,>); // Unbound
Console.WriteLine (t.Name);      // Dictionary'2
Console.WriteLine (t.FullName);  //
System.Collections.Generic.Dictionary'2
```

If the generic type is closed, however, `FullName` (only) acquires a substantial extra appendage. Each type parameter's full *assembly qualified name* is enumerated:

```
Console.WriteLine (typeof (Dictionary<int,string>).FullName);

// OUTPUT:
System.Collections.Generic.Dictionary`2[[System.Int32,
System.Private.CoreLib, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=7cec85d7bea7798e],[System.String, System.Private.CoreLib,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=7cec85d7bea7798e]]
```

This ensures that `AssemblyQualifiedName` (a combination of the type's full name and assembly name) contains enough information to fully identify both the generic type and its type parameters.

### ARRAY AND POINTER TYPE NAMES

Arrays present with the same suffix that you use in a `typeof` expression:

```
Console.WriteLine (typeof ( int[]  ).Name);      // Int32[]
Console.WriteLine (typeof ( int[,] ).Name);      // Int32[,]
Console.WriteLine (typeof ( int[,] ).FullName);  // System.Int32[,]
```

Pointer types are similar:

```
Console.WriteLine (typeof (byte*).Name);     // Byte*
```

### REF AND OUT PARAMETER TYPE NAMES

A `Type` describing a `ref` or `out` parameter has an `&` suffix:

```
public void RefMethod (ref int p)
{
  Type t = MethodInfo.GetCurrentMethod().GetParameters()
[0].ParameterType;
  Console.WriteLine (t.Name);    // Int32&
}
```

More on this later, in the section “Reflecting and Invoking Members”.

## Base Types and Interfaces

`Type` exposes a `BaseType` property:

```
Type base1 = typeof (System.String).BaseType;
Type base2 = typeof (System.IO.FileStream).BaseType;

Console.WriteLine (base1.Name);     // Object
Console.WriteLine (base2.Name);     // Stream
```

The `GetInterfaces` method returns the interfaces that a type implements:

```
foreach (Type iType in typeof (Guid).GetInterfaces())
  Console.WriteLine (iType.Name);

IFormattable
IComparable
IComparable'1
IEquatable'1
```

Reflection provides two dynamic equivalents to C#’s static `is` operator:

`IsInstanceOfType`
: Accepts a type and instance

`IsAssignableFrom`
: Accepts two types

Here’s an example of the first:

```
object obj  = Guid.NewGuid();
Type target = typeof (IFormattable);

bool isTrue   = obj is IFormattable;             // Static C# operator
bool alsoTrue = target.IsInstanceOfType (obj);   // Dynamic
equivalent
```

`IsAssignableFrom` is more versatile:

```
Type target = typeof (IComparable), source = typeof (string);
Console.WriteLine (target.IsAssignableFrom (source));         //
True
```

The `IsSubclassOf` method works on the same principle as `IsAssignableFrom` but excludes interfaces.

## Instantiating Types

There are two ways to dynamically instantiate an object from its type:

- Call the static `Activator.CreateInstance` method
- Call `Invoke` on a `ConstructorInfo` object obtained from calling `GetConstructor` on a `Type` (advanced scenarios)

`Activator.CreateInstance` accepts a `Type` and optional arguments that it passes to the constructor:

```
int i = (int) Activator.CreateInstance (typeof (int));

DateTime dt = (DateTime) Activator.CreateInstance (typeof (DateTime),
                                                   2000, 1, 1);
```

`CreateInstance` lets you specify many other options such as the assembly from which to load the type and whether to bind to a nonpublic constructor. A `MissingMethodException` is thrown if the runtime can't find a suitable constructor.

Calling `Invoke` on a `ConstructorInfo` is necessary when your argument values can't disambiguate between overloaded constructors. For example, suppose that class `X` has two constructors: one accepting a parameter of type `string`, and another accepting a parameter of type `StringBuilder`. The target is ambiguous should you pass a `null` argument into `Activator.CreateInstance`. This is when you need to use a `ConstructorInfo` instead:

```
// Fetch the constructor that accepts a single parameter of type
string:
ConstructorInfo ci = typeof (X).GetConstructor (new[] { typeof (string)
});
```

```
// Construct the object using that overload, passing in null:
object foo = ci.Invoke (new object[] { null });
```

Or, if you're targeting .NET Core 1, an older Windows Store profile:

```
ConstructorInfo ci = typeof (X).GetTypeInfo().DeclaredConstructors
  .FirstOrDefault (c =>
     c.GetParameters().Length == 1 &&
     c.GetParameters()[0].ParameterType == typeof (string));
```

To obtain a nonpublic constructor, you need to specify `BindingFlags`—see "Accessing Nonpublic Members" in the later section "Reflecting and Invoking Members".

> **NOTE**
>
> Dynamic instantiation adds a few microseconds onto the time taken to construct the object. This is quite a lot in relative terms because the CLR is ordinarily very fast in instantiating objects (a simple `new` on a small class takes in the region of tens of nanoseconds).

To dynamically instantiate arrays based on just element type, first call `MakeArrayType`. You can also instantiate generic types: we describe this in the next section.

To dynamically instantiate a delegate, call `Delegate.CreateDelegate`. The following example demonstrates instantiating both an instance delegate and a static delegate:

```
class Program
```

```
{
  delegate int IntFunc (int x);

  static int Square (int x) => x * x;        // Static method
  int        Cube   (int x) => x * x * x;    // Instance method

  static void Main()
  {
    Delegate staticD = Delegate.CreateDelegate
      (typeof (IntFunc), typeof (Program), "Square");

    Delegate instanceD = Delegate.CreateDelegate
      (typeof (IntFunc), new Program(), "Cube");

    Console.WriteLine (staticD.DynamicInvoke (3));      // 9
    Console.WriteLine (instanceD.DynamicInvoke (3));    // 27
  }
}
```

You can invoke the `Delegate` object that's returned by calling `DynamicInvoke`, as we did in this example, or by casting to the typed delegate:

```
IntFunc f = (IntFunc) staticD;
Console.WriteLine (f(3));         // 9 (but much faster!)
```

You can pass a `MethodInfo` into `CreateDelegate` instead of a method name. We describe `MethodInfo` shortly, in "Reflecting and Invoking Members", along with the rationale for casting a dynamically created delegate back to the static delegate type.

## Generic Types

A `Type` can represent a closed or unbound generic type. Just as at compile time, a closed generic type can be instantiated, whereas an unbound type cannot:

```
Type closed = typeof (List<int>);
List<int> list = (List<int>) Activator.CreateInstance (closed);  // OK

Type unbound   = typeof (List<>);
object anError = Activator.CreateInstance (unbound);    // Runtime error
```

The `MakeGenericType` method converts an unbound into a closed generic type. Simply pass in the desired type arguments:

```
Type unbound = typeof (List<>);
Type closed = unbound.MakeGenericType (typeof (int));
```

The `GetGenericTypeDefinition` method does the opposite:

```
Type unbound2 = closed.GetGenericTypeDefinition();  // unbound ==
unbound2
```

The `IsGenericType` property returns `true` if a `Type` is generic, and the `IsGenericTypeDefinition` property returns `true` if the generic type is unbound. The following tests whether a type is a nullable value type:

```
Type nullable = typeof (bool?);
Console.WriteLine (
  nullable.IsGenericType &&
  nullable.GetGenericTypeDefinition() == typeof (Nullable<>));   // True
```

`GetGenericArguments` returns the type arguments for closed generic types:

```
Console.WriteLine (closed.GetGenericArguments()[0]);     // System.Int32
Console.WriteLine (nullable.GetGenericArguments()[0]);   //
System.Boolean
```

For unbound generic types, `GetGenericArguments` returns pseudotypes that represent the placeholder types specified in the generic type definition:

```
Console.WriteLine (unbound.GetGenericArguments()[0]);      // T
```

**NOTE**

At runtime, all generic types are either *unbound* or *closed*. They're unbound in the (relatively unusual) case of an expression such as `typeof(Foo<>)`; otherwise, they're closed. There's no such thing as an *open* generic type at runtime: all open types are closed by the compiler. The method in the following class always prints `False`:

```
class Foo<T>
{
  public void Test()
    => Console.Write (GetType().IsGenericTypeDefinition);
}
```

# Reflecting and Invoking Members

The `GetMembers` method returns the members of a type. Consider the following:

```
class Walnut
{
  private bool cracked;
  public void Crack() { cracked = true; }
}
```

We can reflect on its public members, as follows:

```
MemberInfo[] members = typeof (Walnut).GetMembers();
foreach (MemberInfo m in members)
  Console.WriteLine (m);
```

This is the result:

```
Void Crack()
System.Type GetType()
System.String ToString()
Boolean Equals(System.Object)
Int32 GetHashCode()
Void .ctor()
```

### REFLECTING MEMBERS WITH TYPEINFO

`TypeInfo` exposes a different (and somewhat simpler) protocol for reflecting over members. Using this API is optional in .NET Core 2+, but mandatory in .NET Core 1 and older Windows Store apps given that there's no exact equivalent to the `GetMembers` method.

Instead of exposing methods like `GetMembers` that return arrays, `TypeInfo` exposes *properties* that return `IEnumerable<T>`, upon which you typically run LINQ. The broadest is `DeclaredMembers`:

```
IEnumerable<MemberInfo> members =
```

```
  typeof(Walnut).GetTypeInfo().DeclaredMembers;
```

Unlike with `GetMembers()`, the result excludes inherited members:

```
Void Crack()
Void .ctor()
Boolean cracked
```

There are also properties for returning specific kinds of members (`Declared Properties`, `DeclaredMethods`, `DeclaredEvents`, and so on) and methods for returning a specific member by name (e.g., `GetDeclaredMethod`). The latter cannot be used on overloaded methods (because there's no way to specify parameter types). Instead, you run a LINQ query over `DeclaredMethods`:

```
MethodInfo method = typeof (int).GetTypeInfo().DeclaredMethods
  .FirstOrDefault (m => m.Name == "ToString" &&
                        m.GetParameters().Length == 0);
```

When called with no arguments, `GetMembers` returns all the public members for a type (and its base types). `GetMember` retrieves a specific member by name—although it still returns an array because members can be overloaded:

```
MemberInfo[] m = typeof (Walnut).GetMember ("Crack");
Console.WriteLine (m[0]);                              // Void Crack()
```

`MemberInfo` also has a property called `MemberType` of type `MemberTypes`. This is a flags enum with these values:

```
All           Custom        Field        NestedType     TypeInfo
Constructor   Event         Method       Property
```

When calling `GetMembers`, you can pass in a `MemberTypes` instance to restrict the kinds of members that it returns. Alternatively, you can restrict the result set by calling `GetMethods`, `GetFields`, `GetProperties`, `GetEvents`, `GetConstructors`, or `GetNestedTypes`. There are also singular versions of each of these to hone in on a specific member.

**NOTE**

It pays to be as specific as possible when retrieving a type member so that your code doesn't break if additional members are added later. If you're retrieving a method by name, specifying all parameter types ensures that your code will still work if the method is later overloaded (we provide examples shortly, in "Method Parameters").

A `MemberInfo` object has a `Name` property and two `Type` properties:

`DeclaringType`
: Returns the `Type` that defines the member

`ReflectedType`
: Returns the `Type` upon which `GetMembers` was called

The two differ when called on a member that's defined in a base type: `DeclaringType` returns the base type, whereas `ReflectedType` returns the subtype. The following example highlights this:

```
class Program
{
  static void Main()
  {
    // MethodInfo is a subclass of MemberInfo; see Figure 19-1.

    MethodInfo test = typeof (Program).GetMethod ("ToString");
    MethodInfo obj  = typeof (object) .GetMethod ("ToString");

    Console.WriteLine (test.DeclaringType);      // System.Object
    Console.WriteLine (obj.DeclaringType);       // System.Object

    Console.WriteLine (test.ReflectedType);      // Program
    Console.WriteLine (obj.ReflectedType);       // System.Object

    Console.WriteLine (test == obj);             // False
  }
}
```

Because they have different `ReflectedType`s, the `test` and `obj` objects are not equal. Their difference, however, is purely a fabrication of the reflection API; our `Program` type has no distinct `ToString` method in the underlying type system. We can verify that the two `MethodInfo` objects refer to the same method in either of two ways:

```
Console.WriteLine (test.MethodHandle == obj.MethodHandle);    // True

Console.WriteLine (test.MetadataToken == obj.MetadataToken    // True
                   && test.Module == obj.Module);
```

A `MethodHandle` is unique to each (genuinely distinct) method within a process; a `MetadataToken` is unique across all types and members within an assembly module.

`MemberInfo` also defines methods to return custom attributes (see "Retrieving Attributes at Runtime").

> **NOTE**
>
> You can obtain the `MethodBase` of the currently executing method by calling `MethodBase.GetCurrentMethod`.

## Member Types

`MemberInfo` itself is light on members because it's an abstract base for the types shown in Figure 19-1.

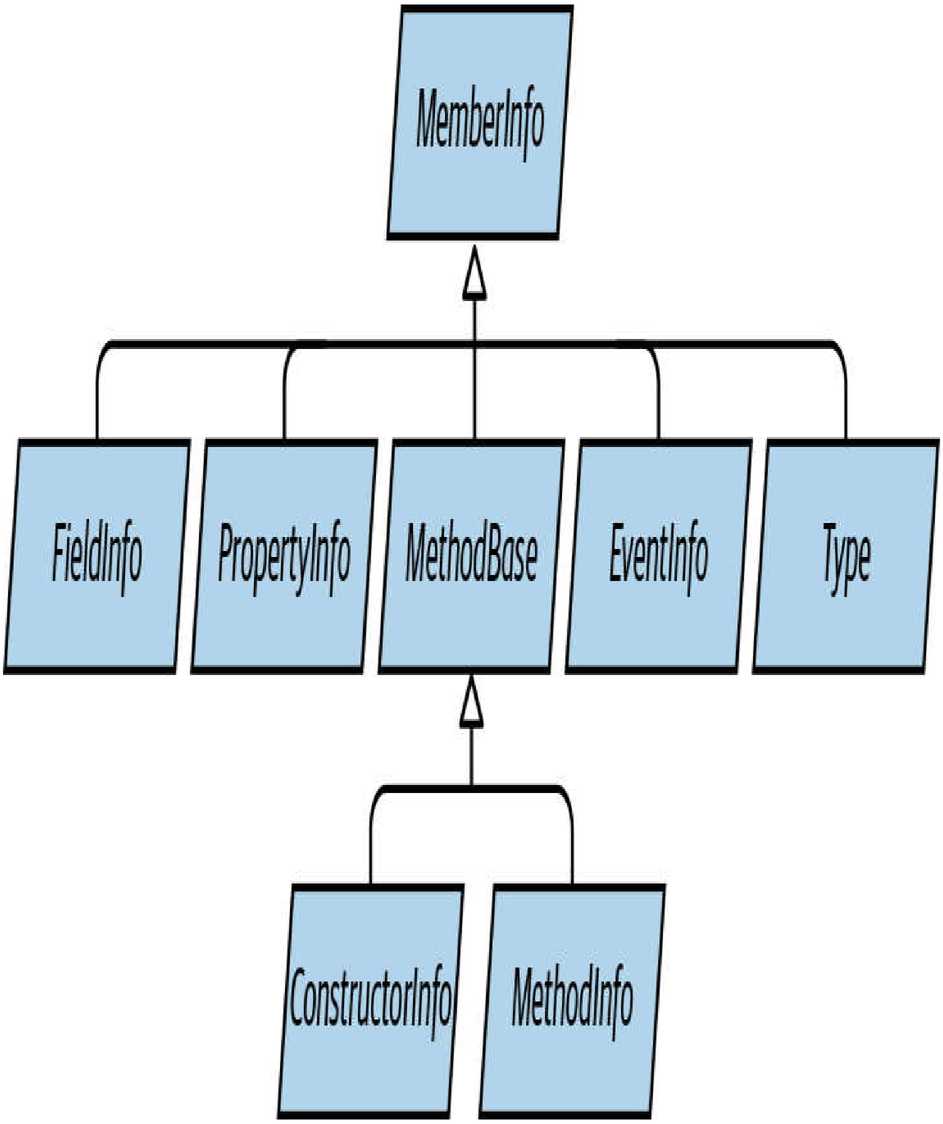

*Figure 19-1. Member types*

You can cast a `MemberInfo` to its subtype, based on its `MemberType` property. If you obtained a member via `GetMethod`, `GetField`, `GetProperty`, `GetEvent`, `GetConstructor`, or `GetNestedType` (or their plural versions), a cast isn't necessary. Table 19-1 summarizes what methods to use for each kind of C# construct.

*T*

*able 19-1. Retrieving member metadata*

| C# construct | Method to use | Name to use | Result |
|---|---|---|---|
| Method | `GetMethod` | (method name) | `MethodInfo` |
| Property | `GetProperty` | (property name) | `PropertyInfo` |
| Indexer | `GetDefaultMembers` | | `MemberInfo[]` (containing `PropertyInfo` objects if compiled in C#) |
| Field | `GetField` | (field name) | `FieldInfo` |
| Enum member | `GetField` | (member name) | `FieldInfo` |
| Event | `GetEvent` | (event name) | `EventInfo` |
| Constructor | `GetConstructor` | | `ConstructorInfo` |
| Finalizer | `GetMethod` | `"Finalize"` | `MethodInfo` |
| Operator | `GetMethod` | `"op_"` + operator name | `MethodInfo` |
| Nested type | `GetNestedType` | (type name) | `Type` |

Each `MemberInfo` subclass has a wealth of properties and methods, exposing all aspects of the member’s metadata. This includes such things as visibility, modifiers, generic type arguments, parameters, return type, and custom attributes.

Here is an example of using `GetMethod`:

```
MethodInfo m = typeof (Walnut).GetMethod ("Crack");
Console.WriteLine (m);                              // Void Crack()
Console.WriteLine (m.ReturnType);                   // System.Void
```

All `*Info` instances are cached by the reflection API on first use:

```
MethodInfo method = typeof (Walnut).GetMethod ("Crack");
MemberInfo member = typeof (Walnut).GetMember ("Crack") [0];

Console.Write (method == member);       // True
```

As well as preserving object identity, caching improves the performance of what is otherwise a fairly slow API.

## C# Members versus CLR Members

The preceding table illustrates that some of C#’s functional constructs don’t have a 1:1 mapping with CLR constructs. This makes sense because the CLR and reflection API were designed with all .NET languages in mind—you can use reflection even from Visual Basic.

Some C# constructs—namely indexers, enums, operators, and finalizers—are contrivances as far as the CLR is concerned. Specifically:

- A C# indexer translates to a property accepting one or more arguments, marked as the type’s `[DefaultMember]`.
- A C# enum translates to a subtype of `System.Enum` with a static field for each member.

- A C# operator translates to a specially named static method, starting in “op_”; for example, `"op_Addition"`.

- A C# finalizer translates to a method that overrides `Finalize`.

Another complication is that properties and events actually comprise two things:

- Metadata describing the property or event (encapsulated by `PropertyInfo` or `EventInfo`)

- One or two backing methods

In a C# program, the backing methods are encapsulated within the property or event definition. But when compiled to IL, the backing methods present as ordinary methods that you can call like any other. This means that `GetMethods` returns property and event backing methods alongside ordinary methods:

```
class Test { public int X { get { return 0; } set {} } }

void Demo()
{
  foreach (MethodInfo mi in typeof (Test).GetMethods())
    Console.Write (mi.Name + "  ");
}

// OUTPUT:
get_X  set_X  GetType  ToString  Equals  GetHashCode
```

You can identify these methods through the `IsSpecialName` property in `MethodInfo`. `IsSpecialName` returns `true` for property, indexer, and event accessors, as well as operators. It returns `false` only for conventional C# methods—and the `Finalize` method if a finalizer is defined.

Here are the backing methods that C# generates:

| C# construct | Member type | Methods in IL |
|---|---|---|
| Property | `Property` | `get_`*`XXX`* and `set_`*`XXX`* |
| Indexer | `Property` | `get_Item` and `set_Item` |
| Event | `Event` | `add_`*`XXX`* and `remove_`*`XXX`* |

Each backing method has its own associated `MethodInfo` object. You can access these as follows:

```
PropertyInfo pi = typeof (Console).GetProperty ("Title");
MethodInfo getter = pi.GetGetMethod();                    // get_Title
MethodInfo setter = pi.GetSetMethod();                    // set_Title
MethodInfo[] both = pi.GetAccessors();                    // Length==2
```

`GetAddMethod` and `GetRemoveMethod` perform a similar job for `EventInfo`.

To go in the reverse direction—from a `MethodInfo` to its associated `PropertyInfo` or `EventInfo`—you need to perform a query. LINQ is ideal for this job:

```
PropertyInfo p = mi.DeclaringType.GetProperties()
                   .First (x => x.GetAccessors (true).Contains (mi));
```

## Generic Type Members

You can obtain member metadata for both unbound and closed generic types:

```
PropertyInfo unbound = typeof (IEnumerator<>)  .GetProperty
("Current");
PropertyInfo closed = typeof (IEnumerator<int>).GetProperty
("Current");

Console.WriteLine (unbound);   // T Current
Console.WriteLine (closed);    // Int32 Current

Console.WriteLine (unbound.PropertyType.IsGenericParameter);  // True
Console.WriteLine (closed.PropertyType.IsGenericParameter);   // False
```

The `MemberInfo` objects returned from unbound and closed generic types are always distinct, even for members whose signatures don't feature generic type parameters:

```
PropertyInfo unbound = typeof (List<>)  .GetProperty ("Count");
PropertyInfo closed = typeof (List<int>).GetProperty ("Count");

Console.WriteLine (unbound);   // Int32 Count
Console.WriteLine (closed);    // Int32 Count

Console.WriteLine (unbound == closed);   // False

Console.WriteLine (unbound.DeclaringType.IsGenericTypeDefinition); //
True
```

```
Console.WriteLine (closed.DeclaringType.IsGenericTypeDefinition); //
False
```

Members of unbound generic types cannot be *dynamically invoked*.

## Dynamically Invoking a Member

After you have a `MethodInfo`, `PropertyInfo`, or `FieldInfo` object, you can dynamically call it or get/set its value. This is called *late binding* because you choose which member to invoke at runtime rather than compile time.

To illustrate, the following uses ordinary *static binding*:

```
string s = "Hello";
int length = s.Length;
```

Here's the same thing performed dynamically with late binding:

```
object s = "Hello";
PropertyInfo prop = s.GetType().GetProperty ("Length");
int length = (int) prop.GetValue (s, null);              // 5
```

`GetValue` and `SetValue` get and set the value of a `PropertyInfo` or `FieldInfo`. The first argument is the instance, which can be `null` for a static member. Accessing an indexer is just like accessing a property called *Item*, except that you provide indexer values as the second argument when calling `GetValue` or `SetValue`.

To dynamically call a method, call `Invoke` on a `MethodInfo`, providing an array of arguments to pass to that method. If you get any of the argument types wrong, an exception is thrown at runtime. With dynamic invocation, you lose compile-time type safety, but you still have runtime type safety (just as with the `dynamic` keyword).

## Method Parameters

Suppose that we want to dynamically call `string`'s `Substring` method. Statically, we would do this as follows:

```
Console.WriteLine ("stamp".Substring(2));                  // "amp"
```

Here's the dynamic equivalent with reflection and late binding:

```
Type type = typeof (string);
Type[] parameterTypes = { typeof (int) };
MethodInfo method = type.GetMethod ("Substring", parameterTypes);

object[] arguments = { 2 };
object returnValue = method.Invoke ("stamp", arguments);
Console.WriteLine (returnValue);                           // "amp"
```

Because the `Substring` method is overloaded, we had to pass an array of parameter types to `GetMethod` to indicate which version we wanted. Without the parameter types, `GetMethod` would throw an `AmbiguousMatchException`.

The `GetParameters` method, defined on `MethodBase` (the base class for `MethodInfo` and `ConstructorInfo`), returns parameter metadata.

We can continue our previous example as follows:

```
ParameterInfo[] paramList = method.GetParameters();
foreach (ParameterInfo x in paramList)
{
  Console.WriteLine (x.Name);                 // startIndex
  Console.WriteLine (x.ParameterType);        // System.Int32
}
```

## DEALING WITH REF AND OUT PARAMETERS

To pass `ref` or `out` parameters, call `MakeByRefType` on the type before obtaining the method. For instance, you can dynamically execute this code:

```
int x;
bool successfulParse = int.TryParse ("23", out x);
```

as follows:

```
object[] args = { "23", 0 };
Type[] argTypes = { typeof (string), typeof (int).MakeByRefType() };
MethodInfo tryParse = typeof (int).GetMethod ("TryParse", argTypes);
bool successfulParse = (bool) tryParse.Invoke (null, args);

Console.WriteLine (successfulParse + " " + args[1]);       // True 23
```

This same approach works for both `ref` and `out` parameter types.

## RETRIEVING AND INVOKING GENERIC METHODS

Explicitly specifying parameter types when calling `GetMethod` can be essential in disambiguating overloaded methods. However, it's impossible to specify generic parameter types. For instance, consider the `System.Linq.Enumerable` class, which overloads the `Where` method, as follows:

```
public static IEnumerable<TSource> Where<TSource>
 (this IEnumerable<TSource> source, Func<TSource, bool> predicate);

public static IEnumerable<TSource> Where<TSource>
 (this IEnumerable<TSource> source, Func<TSource, int, bool> predicate);
```

To retrieve a specific overload, we must retrieve all methods and then manually find the desired overload. The following query retrieves the former overload of `Where`:

```
from m in typeof (Enumerable).GetMethods()
where m.Name == "Where" && m.IsGenericMethod
let parameters = m.GetParameters()
where parameters.Length == 2
let genArg = m.GetGenericArguments().First()
let enumerableOfT = typeof (IEnumerable<>).MakeGenericType (genArg)
let funcOfTBool = typeof (Func<,>).MakeGenericType (genArg, typeof
(bool))
where parameters[0].ParameterType == enumerableOfT
   && parameters[1].ParameterType == funcOfTBool
select m
```

Calling `.Single()` on this query gives the correct `MethodInfo` object with unbound type parameters. The next step is to close the type parameters by calling `MakeGenericMethod`:

```
var closedMethod = unboundMethod.MakeGenericMethod (typeof (int));
```

In this case, we’ve closed `TSource` with `int`, allowing us to call `Enumerable.Where` with a source of type `IEnumerable<int>` and a predicate of type `Func<int,bool>`:

```
int[] source = { 3, 4, 5, 6, 7, 8 };
Func<int, bool> predicate = n => n % 2 == 1;   // Odd numbers only
```

We can now invoke the closed generic method:

```
var query = (IEnumerable<int>) closedMethod.Invoke
  (null, new object[] { source, predicate });

foreach (int element in query) Console.Write (element + "|");  // 3|5|7|
```

> **NOTE**
>
> If you're using the `System.Linq.Expressions` API to dynamically build expressions (Chapter 8), you don't need to go to this trouble to specify a generic method. The `Expression.Call` method is overloaded to let you specify the closed type arguments of the method that you want to call:
>
> ```
> int[] source = { 3, 4, 5, 6, 7, 8 };
> Func<int, bool> predicate = n => n % 2 == 1;
>
> var sourceExpr = Expression.Constant (source);
> var predicateExpr = Expression.Constant (predicate);
>
> var callExpression = Expression.Call (
>   typeof (Enumerable), "Where",
>   new[] { typeof (int) },  // Closed generic arg type.
>   sourceExpr, predicateExpr);
> ```

## Using Delegates for Performance

Dynamic invocations are relatively inefficient, with an overhead typically in the few-microseconds region. If you're calling a method repeatedly in a loop, you can shift the per-call overhead into the nanoseconds region by instead calling a dynamically instantiated delegate that targets your dynamic method. In the following example, we dynamically call `string`'s `Trim` method a million times without significant overhead:

```
delegate string StringToString (string s);

static void Main()
```

```
{
  MethodInfo trimMethod = typeof (string).GetMethod ("Trim", new
Type[0]);
  var trim = (StringToString) Delegate.CreateDelegate
                                    (typeof (StringToString),
trimMethod);
  for (int i = 0; i < 1000000; i++)
    trim ("test");
}
```

This is faster because the costly late binding (shown in bold) happens just once.

## Accessing Nonpublic Members

All of the methods on types used to probe metadata (e.g., `GetProperty`, `GetField`, etc.) have overloads that take a `BindingFlags` enum. This enum serves as a metadata filter and allows you to change the default selection criteria. The most common use for this is to retrieve nonpublic members (this works only in desktop apps).

For instance, consider the following class:

```
class Walnut
{
  private bool cracked;
  public void Crack() { cracked = true; }

  public override string ToString() { return cracked.ToString(); }
}
```

We can *uncrack* the walnut as follows:

```
Type t = typeof (Walnut);
Walnut w = new Walnut();
w.Crack();
FieldInfo f = t.GetField ("cracked", BindingFlags.NonPublic |
                                     BindingFlags.Instance);
f.SetValue (w, false);
Console.WriteLine (w);         // False
```

Using reflection to access nonpublic members is powerful, but it is also dangerous because you can bypass encapsulation, creating an unmanageable dependency on the internal implementation of a type.

### THE BINDINGFLAGS ENUM

`BindingFlags` is intended to be bitwise-combined. To get any matches at all, you need to start with one of the following four combinations:

```
BindingFlags.Public    | BindingFlags.Instance
BindingFlags.Public    | BindingFlags.Static
BindingFlags.NonPublic | BindingFlags.Instance
BindingFlags.NonPublic | BindingFlags.Static
```

`NonPublic` includes `internal`, `protected`, `protected internal`, and `private`.

The following example retrieves all the public static members of type `object`:

```
BindingFlags publicStatic = BindingFlags.Public | BindingFlags.Static;
MemberInfo[] members = typeof (object).GetMembers (publicStatic);
```

The following example retrieves all the nonpublic members of type `object`, both static and instance:

```
BindingFlags nonPublicBinding =
  BindingFlags.NonPublic | BindingFlags.Static | BindingFlags.Instance;

MemberInfo[] members = typeof (object).GetMembers (nonPublicBinding);
```

The `DeclaredOnly` flag excludes functions inherited from base types, unless they are overridden.

> **NOTE**
>
> The `DeclaredOnly` flag is somewhat confusing in that it *restricts* the result set (whereas all the other binding flags *expand* the result set).

## Generic Methods

You cannot directly invoke generic methods; the following throws an exception:

```
class Program
{
  public static T Echo<T> (T x) { return x; }

  static void Main()
  {
    MethodInfo echo = typeof (Program).GetMethod ("Echo");
    Console.WriteLine (echo.IsGenericMethodDefinition);    // True
    echo.Invoke (null, new object[] { 123 } );             //
Exception
```

```
  }
}
```

An extra step is required, which is to call `MakeGenericMethod` on the `MethodInfo`, specifying concrete generic type arguments. This returns another `MethodInfo`, which you can then invoke as follows:

```
MethodInfo echo = typeof (Program).GetMethod ("Echo");
MethodInfo intEcho = echo.MakeGenericMethod (typeof (int));
Console.WriteLine (intEcho.IsGenericMethodDefinition);            //
False
Console.WriteLine (intEcho.Invoke (null, new object[] { 3 } ));   // 3
```

## Anonymously Calling Members of a Generic Interface

Reflection is useful when you need to invoke a member of a generic interface and you don’t know the type parameters until runtime. In theory, the need for this arises rarely if types are perfectly designed; of course, types are not always perfectly designed.

For instance, suppose that we want to write a more powerful version of `ToString` that could expand the result of LINQ queries. We could start out as follows:

```
public static string ToStringEx <T> (IEnumerable<T> sequence)
{
  ...
}
```

This is already quite limiting. What if `sequence` contained *nested* collections that we also want to enumerate? We'd need to overload the method to cope:

```
public static string ToStringEx <T> (IEnumerable<IEnumerable<T>>
sequence)
```

And then what if `sequence` contained groupings, or *projections* of nested sequences? The static solution of method overloading becomes impractical—we need an approach that can scale to handle an arbitrary object graph, such as the following:

```
public static string ToStringEx (object value)
{
  if (value == null) return "<null>";
  StringBuilder sb = new StringBuilder();

  if (value is List<>)                                          //
Error
    sb.Append ("List of " + ((List<>) value).Count + " items");   //
Error

  if (value is IGrouping<,>)                                     //
Error
    sb.Append ("Group with key=" + ((IGrouping<,>) value).Key);   //
Error

  // Enumerate collection elements if this is a collection,
  // recursively calling ToStringEx()
  // ...

  return sb.ToString();
}
```

Unfortunately, this won’t compile: you cannot invoke members of an *unbound* generic type such as `List<>` or `IGrouping<>`. In the case of `List<>`, we can solve the problem by using the nongeneric `IList` interface instead:

```
if (value is IList)
  sb.AppendLine ("A list with " + ((IList) value).Count + " items");
```

> **NOTE**
>
> We can do this because the designers of `List<>` had the foresight to implement `IList` classic (as well as `IList` *generic*). The same principle is worthy of consideration when writing your own generic types: having a nongeneric interface or base class upon which consumers can fall back can be extremely valuable.

The solution is not as simple for `IGrouping<,>`. Here’s how the interface is defined:

```
public interface IGrouping <TKey,TElement> : IEnumerable <TElement>,
                                             IEnumerable
{
  TKey Key { get; }
}
```

There’s no nongeneric type we can use to access the `Key` property, so here we must use reflection. The solution is not to invoke members of an unbound generic type (which is impossible), but to invoke members of a *closed* generic type, whose type arguments we establish at runtime.

> **NOTE**
>
> In the following chapter, we solve this more simply with C#’s `dynamic` keyword. A good indication for dynamic binding is when you would otherwise need to perform *type gymnastics*—as we are doing right now.

The first step is to determine whether `value` implements `IGrouping<,>`, and if so, obtain its closed generic interface. We can do this most easily by executing a LINQ query. Then, we retrieve and invoke the `Key` property:

```
public static string ToStringEx (object value)
{
  if (value == null) return "<null>";
  if (value.GetType().IsPrimitive) return value.ToString();

  StringBuilder sb = new StringBuilder();

  if (value is IList)
    sb.Append ("List of " + ((IList)value).Count + " items: ");

  Type closedIGrouping = value.GetType().GetInterfaces()
    .Where (t => t.IsGenericType &&
                 t.GetGenericTypeDefinition() == typeof
(IGrouping<,>))
    .FirstOrDefault();

  if (closedIGrouping != null)   // Call the Key property on
IGrouping<,>
```

```
  {
    PropertyInfo pi = closedIGrouping.GetProperty ("Key");
    object key = pi.GetValue (value, null);
    sb.Append ("Group with key=" + key + ": ");
  }

  if (value is IEnumerable)
    foreach (object element in ((IEnumerable)value))
      sb.Append (ToStringEx (element) + " ");

  if (sb.Length == 0) sb.Append (value.ToString());

  return "\r\n" + sb.ToString();
}
```

This approach is robust: it works whether `IGrouping<,>` is implemented implicitly or explicitly. The following demonstrates this method:

```
Console.WriteLine (ToStringEx (new List<int> { 5, 6, 7 } ));
Console.WriteLine (ToStringEx ("xyyzzz".GroupBy (c => c) ));

List of 3 items: 5 6 7

Group with key=x: x
Group with key=y: y y
Group with key=z: z z z
```

## Reflecting Assemblies

You can dynamically reflect an assembly by calling `GetType` or `GetTypes` on an `Assembly` object. The following retrieves from the current assembly, the type called `TestProgram` in the `Demos` namespace:

```
Type t = Assembly.GetExecutingAssembly().GetType ("Demos.TestProgram");
```

You can also obtain an assembly from an existing type:

```
typeof (Foo).Assembly.GetType ("Demos.TestProgram");
```

The next example lists all the types in the assembly *mylib.dll* in *e:\demo*:

```
Assembly a = Assembly.LoadFile (@"e:\demo\mylib.dll");

foreach (Type t in a.GetTypes())
  Console.WriteLine (t);
```

Or:

```
Assembly a = typeof (Foo).GetTypeInfo().Assembly;

foreach (Type t in a.ExportedTypes)
  Console.WriteLine (t);
```

`GetTypes` and `ExportedTypes` return only top-level and not nested types.

## Modules

Calling `GetTypes` on a multimodule assembly returns all types in all modules. As a result, you can ignore the existence of modules and treat an assembly as a type’s container. There is one case, though, for which modules are relevant—and that's when dealing with metadata tokens.

A metadata token is an integer that uniquely refers to a type, member, string, or resource within the scope of a module. IL uses metadata tokens, so if you're parsing IL, you'll need to be able to resolve them. The methods for doing this are defined in the `Module` type and are called `ResolveType`, `ResolveMember`, `ResolveString`, and `ResolveSignature`. We revisit this in the final section of this chapter, on writing a disassembler.

You can obtain a list of all the modules in an assembly by calling `GetModules`. You can also access an assembly's main module directly via its `ManifestModule` property.

## Working with Attributes

The CLR allows additional metadata to be attached to types, members, and assemblies through attributes. This is the mechanism by which many CLR functions such as serialization and security are directed, making attributes an indivisible part of an application.

A key characteristic of attributes is that you can write your own and then use them just as you would any other attribute to "decorate" a code element with additional information. This additional information is compiled into the underlying assembly and can be retrieved at runtime using reflection to build services that work declaratively, such as automated unit testing.

## Attribute Basics

There are three kinds of attributes:

- Bit-mapped attributes
- Custom attributes
- Pseudocustom attributes

Of these, only *custom attributes* are extensible.

> **NOTE**
>
> The term “attribute” by itself can refer to any of the three, although in the C# world, it most often refers to custom attributes or pseudocustom attributes.

Bit-mapped attributes (our terminology) map to dedicated bits in a type’s metadata. Most of C#’s modifier keywords, such as `public`, `abstract`, and `sealed`, compile to bit-mapped attributes. These attributes are very efficient because they consume minimal space in the metadata (usually just one bit), and the CLR can locate them with little or no indirection. The reflection API exposes them via dedicated properties on `Type` (and other `MemberInfo` subclasses), such as `IsPublic`, `IsAbstract`, and `IsSealed`. The `Attributes` property returns a flags enum that describes most of them in one hit:

```
static void Main()
{
  TypeAttributes ta = typeof (Console).Attributes;
```

```
    MethodAttributes ma = MethodInfo.GetCurrentMethod().Attributes;
    Console.WriteLine (ta + "\r\n" + ma);
}
```

Here’s the result:

```
AutoLayout, AnsiClass, Class, Public, Abstract, Sealed, BeforeFieldInit
PrivateScope, Private, Static, HideBySig
```

In contrast, *custom attributes* compile to a blob that hangs off the type’s main metadata table. All custom attributes are represented by a subclass of `System.Attribute` and, unlike bit-mapped attributes, are extensible. The blob in the metadata identifies the attribute class, and also stores the values of any positional or named argument that was specified when the attribute was applied. Custom attributes that you define yourself are architecturally identical to those defined in .NET Core.

Chapter 4 described how to attach custom attributes to a type or member in C#. Here, we attach the predefined `Obsolete` attribute to the `Foo` class:

```
[Obsolete] public class Foo {...}
```

This instructs the compiler to incorporate an instance of `ObsoleteAttribute` into the metadata for `Foo`, which then can be reflected at runtime by calling `GetCustomAttributes` on a `Type` or `MemberInfo` object.

*Pseudocustom attributes* look and feel just like standard custom attributes. They are represented by a subclass of `System.Attribute` and are attached in the standard manner:

```
[Serializable] public class Foo {...}
```

The difference is that the compiler or CLR internally optimizes pseudocustom attributes by converting them to bit-mapped attributes. Examples include `[Serializable]` (Chapter 17), `StructLayout`, `In`, and `Out` (Chapter 25). Reflection exposes pseudocustom attributes through dedicated properties such as `IsSerializable`, and in many cases they are also returned as `System.Attribute` objects when you call `GetCustomAttributes` (`SerializableAttribute` included). This means that you can (almost) ignore the difference between pseudo- and non-pseudocustom attributes (a notable exception is when using `Reflection.Emit` to generate types dynamically at runtime; see "Emitting Assemblies and Types").

## The AttributeUsage Attribute

`AttributeUsage` is an attribute applied to attribute classes. It instructs the compiler how the target attribute should be used:

```
public sealed class AttributeUsageAttribute : Attribute
{
  public AttributeUsageAttribute (AttributeTargets validOn);

  public bool AllowMultiple        { get; set; }
  public bool Inherited            { get; set; }
```

```
  public AttributeTargets ValidOn  { get; }
}
```

`AllowMultiple` controls whether the attribute being defined can be applied more than once to the same target; `Inherited` controls whether an attribute applied to a base class also applies to derived classes (or in the case of methods, whether an attribute applied to a virtual method also applies to overriding methods). `ValidOn` determines the set of targets (classes, interfaces, properties, methods, parameters, etc.) to which the attribute can be attached. It accepts any combination of values from the `AttributeTargets` enum, which has the following members:

| | | | |
|---|---|---|---|
| `All` | `Delegate` | `GenericParameter` | `Parameter` |
| `Assembly` | `Enum` | `Interface` | `Property` |
| `Class` | `Event` | `Method` | `ReturnValue` |
| `Constructor` | `Field` | `Module` | `Struct` |

To illustrate, here's how the authors of .NET Core have applied `AttributeUsage` to the `Serializable` attribute:

```
[AttributeUsage (AttributeTargets.Delegate |
                 AttributeTargets.Enum     |
                 AttributeTargets.Struct   |
                 AttributeTargets.Class,     Inherited = false)
]
public sealed class SerializableAttribute : Attribute { }
```

This is, in fact, almost the complete definition of the `Serializable` attribute. Writing an attribute class that has no properties or special constructors is this simple.

## Defining Your Own Attribute

Here's how to write your own attribute:

1. Derive a class from `System.Attribute` or a descendent of `System.Attribute`. By convention, the class name should end with the word "Attribute," although this isn't required.

2. Apply the `AttributeUsage` attribute, described in the preceding section.

   If the attribute requires no properties or arguments in its constructor, the job is done.

3. Write one or more public constructors. The parameters to the constructor define the positional parameters of the attribute and will become mandatory when using the attribute.

4. Declare a public field or property for each named parameter you wish to support. Named parameters are optional when using the attribute.

## NOTE

Attribute properties and constructor parameters must be of the following types:

- A sealed primitive type: in other words, `bool`, `byte`, `char`, `double`, `float`, `int`, `long`, `short`, or `string`
- The `Type` type
- An enum type
- A one-dimensional array of any of these

When an attribute is applied, it must also be possible for the compiler to statically evaluate each of the properties or constructor arguments.

The following class defines an attribute for assisting an automated unit-testing system. It indicates that a method should be tested, the number of test repetitions, and a message in case of failure:

```
[AttributeUsage (AttributeTargets.Method)]
public sealed class TestAttribute : Attribute
{
  public int     Repetitions;
  public string  FailureMessage;

  public TestAttribute () : this (1)     { }
  public TestAttribute (int repetitions) { Repetitions = repetitions; }
}
```

Here’s a `Foo` class with methods decorated in various ways with the `Test` attribute:

```
class Foo
```

```
{
  [Test]
  public void Method1() { ... }

  [Test(20)]
  public void Method2() { ... }

  [Test(20, FailureMessage="Debugging Time!")]
  public void Method3() { ... }
}
```

## Retrieving Attributes at Runtime

There are two standard ways to retrieve attributes at runtime:

- Call `GetCustomAttributes` on any `Type` or `MemberInfo` object
- Call `Attribute.GetCustomAttribute` or `Attribute.GetCustomAttributes`

These latter two methods are overloaded to accept any reflection object that corresponds to a valid attribute target (`Type`, `Assembly`, `Module`, `MemberInfo`, or `ParameterInfo`).

> **NOTE**
>
> You can also call `GetCustomAttributes`**`Data`**`()` on a type or member to obtain attribute information. The difference between this and `GetCustomAttributes()` is that the former lets you know you *how* the attribute was instantiated: it reports the constructor overload that was used, and the value of each constructor argument and named parameter. This is useful when you want to emit code or IL to reconstruct the attribute to the same state (see "Emitting Type Members").

Here’s how we can enumerate each method in the preceding `Foo` class that has a `TestAttribute`:

```
foreach (MethodInfo mi in typeof (Foo).GetMethods())
{
  TestAttribute att = (TestAttribute) Attribute.GetCustomAttribute
    (mi, typeof (TestAttribute));

  if (att != null)
    Console.WriteLine ("Method {0} will be tested; reps={1}; msg={2}",
                        mi.Name, att.Repetitions, att.FailureMessage);
}
```

Or:

```
foreach (MethodInfo mi in typeof
(Foo).GetTypeInfo().DeclaredMethods)
...
```

Here’s the output:

```
Method Method1 will be tested; reps=1; msg=
Method Method2 will be tested; reps=20; msg=
Method Method3 will be tested; reps=20; msg=Debugging Time!
```

To complete the illustration on how we could use this to write a unit-testing system, here’s the same example expanded so that it actually calls the methods decorated with the `Test` attribute:

```
foreach (MethodInfo mi in typeof (Foo).GetMethods())
{
```

```
  TestAttribute att = (TestAttribute) Attribute.GetCustomAttribute
    (mi, typeof (TestAttribute));

  if (att != null)
    for (int i = 0; i < att.Repetitions; i++)
      try
      {
        mi.Invoke (new Foo(), null);    // Call method with no arguments
      }
      catch (Exception ex)       // Wrap exception in att.FailureMessage
      {
        throw new Exception ("Error: " + att.FailureMessage, ex);
      }
}
```

Returning to attribute reflection, here’s an example that lists the attributes present on a specific type:

```
[Serializable, Obsolete]
class Test
{
  static void Main()
  {
    object[] atts = Attribute.GetCustomAttributes (typeof (Test));
    foreach (object att in atts) Console.WriteLine (att);
  }
}
```

And, here’s the output:

```
System.ObsoleteAttribute
System.SerializableAttribute
```

# Dynamic Code Generation

The `System.Reflection.Emit` namespace contains classes for creating metadata and IL at runtime. Generating code dynamically is useful for certain kinds of programming tasks. An example is the regular expressions API, which emits performant types tuned to specific regular expressions. Another example is Entity Framework Core, which uses `Reflection.Emit` to generate proxy classes to enable lazy loading.

## Generating IL with DynamicMethod

The `DynamicMethod` class is a lightweight tool in the `System.Reflection.Emit` namespace for generating methods on the fly. Unlike `TypeBuilder`, it doesn't require that you first set up a dynamic assembly, module, and type in which to contain the method. This makes it suitable for simple tasks—as well as serving as a good introduction to `Reflection.Emit`.

> **NOTE**
>
> A `DynamicMethod` and the associated IL are garbage-collected when no longer referenced. This means you can repeatedly generate dynamic methods without filling up memory. (To do the same with dynamic *assemblies*, you must apply the `AssemblyBuilderAccess.RunAndCollect` flag when creating the assembly.)

Here is a simple use of `DynamicMethod` to create a method that writes `Hello world` to the console:

```
public class Test
```

```
{
  static void Main()
  {
    var dynMeth = new DynamicMethod ("Foo", null, null, typeof (Test));
    ILGenerator gen = dynMeth.GetILGenerator();
    gen.EmitWriteLine ("Hello world");
    gen.Emit (OpCodes.Ret);
    dynMeth.Invoke (null, null);                    // Hello world
  }
}
```

`OpCodes` has a static read-only field for every IL opcode. Most of the functionality is exposed through various opcodes, although `ILGenerator` also has specialized methods for generating labels and local variables and for exception handling. A method always ends in `Opcodes.Ret`, which means "return," or some kind of branching/throwing instruction. The `EmitWriteLine` method on `ILGenerator` is a shortcut for `Emit`ting a number of lower-level opcodes. We would get the same result if we replaced the call to `EmitWriteLine` with this:

```
MethodInfo writeLineStr = typeof (Console).GetMethod ("WriteLine",
                           new Type[] { typeof (string) });
gen.Emit (OpCodes.Ldstr, "Hello world");     // Load a string
gen.Emit (OpCodes.Call, writeLineStr);       // Call a method
```

Note that we passed `typeof(Test)` into `DynamicMethod`'s constructor. This gives the dynamic method access to the nonpublic methods of that type, allowing us to do this:

```
public class Test
{
```

```
  static void Main()
  {
    var dynMeth = new DynamicMethod ("Foo", null, null, typeof
(Test));
    ILGenerator gen = dynMeth.GetILGenerator();

    MethodInfo privateMethod = typeof(Test).GetMethod
("HelloWorld",
      BindingFlags.Static | BindingFlags.NonPublic);

    gen.Emit (OpCodes.Call, privateMethod);     // Call HelloWorld
    gen.Emit (OpCodes.Ret);

    dynMeth.Invoke (null, null);                // Hello world
  }

  static void HelloWorld()       // private method, yet we can call it
  {
    Console.WriteLine ("Hello world");
  }
}
```

Understanding IL requires a considerable investment of time. Rather than understand all the opcodes, it’s much easier to compile a C# program and then examine, copy, and tweak the IL. LINQPad displays the IL for any method or code snippet that you type, and assembly viewing tools such ILSpy are useful for examining existing assemblies.

## The Evaluation Stack

Central to IL is the concept of the *evaluation stack*. To call a method with arguments, you first push (*load*) the arguments onto the evaluation stack and then call the method. The method then pops the arguments it needs from the evaluation stack. We demonstrated this previously, in calling `Console.WriteLine`. Here's a similar example with an integer:

```
var dynMeth = new DynamicMethod ("Foo", null, null, typeof(void));
ILGenerator gen = dynMeth.GetILGenerator();
MethodInfo writeLineInt = typeof (Console).GetMethod ("WriteLine",
                                        new Type[] { typeof (int) });

// The Ldc* op-codes load numeric literals of various types and sizes.

gen.Emit (OpCodes.Ldc_I4, 123);        // Push a 4-byte integer onto
stack
gen.Emit (OpCodes.Call, writeLineInt);

gen.Emit (OpCodes.Ret);
dynMeth.Invoke (null, null);           // 123
```

To add two numbers together, you first load each number onto the evaluation stack, and then call `Add`. The `Add` opcode pops two values from the evaluation stack and pushes the result back on. The following adds 2 and 2, and then writes the result using the `writeLine` method obtained previously:

```
gen.Emit (OpCodes.Ldc_I4, 2);           // Push a 4-byte integer,
value=2
gen.Emit (OpCodes.Ldc_I4, 2);           // Push a 4-byte integer,
value=2
gen.Emit (OpCodes.Add);                 // Add the result together
gen.Emit (OpCodes.Call, writeLineInt);
```

To calculate `10 / 2 + 1`, you can do either this:

```
gen.Emit (OpCodes.Ldc_I4, 10);
gen.Emit (OpCodes.Ldc_I4, 2);
gen.Emit (OpCodes.Div);
gen.Emit (OpCodes.Ldc_I4, 1);
gen.Emit (OpCodes.Add);
gen.Emit (OpCodes.Call, writeLineInt);
```

or this:

```
gen.Emit (OpCodes.Ldc_I4, 1);
gen.Emit (OpCodes.Ldc_I4, 10);
gen.Emit (OpCodes.Ldc_I4, 2);
gen.Emit (OpCodes.Div);
gen.Emit (OpCodes.Add);
gen.Emit (OpCodes.Call, writeLineInt);
```

## Passing Arguments to a Dynamic Method

The `Ldarg` and `Ldarg_`*XXX* opcodes load an argument passed into a method onto the stack. To return a value, leave exactly one value on the stack upon finishing. For this to work, you must specify the return type and argument types when calling `DynamicMethod`'s constructor. The following creates a dynamic method that returns the sum of two integers:

```
DynamicMethod dynMeth = new DynamicMethod ("Foo",
  typeof (int),                              // Return type
= int
  new[] { typeof (int), typeof (int) },      // Parameter
types = int, int
  typeof (void));
```

```
ILGenerator gen = dynMeth.GetILGenerator();

gen.Emit (OpCodes.Ldarg_0);      // Push first arg onto eval stack
gen.Emit (OpCodes.Ldarg_1);      // Push second arg onto eval stack
gen.Emit (OpCodes.Add);          // Add them together (result on stack)
gen.Emit (OpCodes.Ret);          // Return with stack having 1 value

int result = (int) dynMeth.Invoke (null, new object[] { 3, 4 } );   // 7
```

> **NOTE**
>
> When you exit, the evaluation stack must have exactly 0 or 1 item (depending on whether your method returns a value). If you violate this, the CLR will refuse to execute your method. You can remove an item from the stack without processing it by emitting `OpCodes.Pop`.

Rather than calling `Invoke`, it can be more convenient to work with a dynamic method as a typed delegate. The `CreateDelegate` method achieves just this. In our case, the delegate that we need has two integer parameters and an integer return type. We can use the `Func<int, int, int>` delegate for this purpose. The last line of our preceding example then becomes the following:

```
var func = (Func<int,int,int>) dynMeth.CreateDelegate
                                 (typeof (Func<int,int,int>));
int result = func (3, 4);  // 7
```

**NOTE**

A delegate also eliminates the overhead of dynamic method invocation—saving a few microseconds per call.

We demonstrate how to pass by reference in “Emitting Type Members”.

## Generating Local Variables

You can declare a local variable by calling `DeclareLocal` on an `ILGenerator`. This returns a `LocalBuilder` object, which you can use in conjunction with opcodes such as `Ldloc` (load a local variable) or `Stloc` (store a local variable). `Ldloc` pushes the evaluation stack; `Stloc` pops it. For example, consider the following C# code:

```
int x = 6;
int y = 7;
x *= y;
Console.WriteLine (x);
```

The following generates the preceding code dynamically:

```
var dynMeth = new DynamicMethod ("Test", null, null, typeof (void));
ILGenerator gen = dynMeth.GetILGenerator();

LocalBuilder localX = gen.DeclareLocal (typeof (int));    // Declare x
LocalBuilder localY = gen.DeclareLocal (typeof (int));    // Declare y

gen.Emit (OpCodes.Ldc_I4, 6);        // Push literal 6 onto eval stack
gen.Emit (OpCodes.Stloc, localX);    // Store in localX
```

```
gen.Emit (OpCodes.Ldc_I4, 7);        // Push literal 7 onto eval stack
gen.Emit (OpCodes.Stloc, localY);    // Store in localY

gen.Emit (OpCodes.Ldloc, localX);    // Push localX onto eval stack
gen.Emit (OpCodes.Ldloc, localY);    // Push localY onto eval stack
gen.Emit (OpCodes.Mul);              // Multiply values together
gen.Emit (OpCodes.Stloc, localX);    // Store the result to localX

gen.EmitWriteLine (localX);          // Write the value of localX
gen.Emit (OpCodes.Ret);

dynMeth.Invoke (null, null);         // 42
```

## Branching

In IL, there are no `while`, `do`, and `for` loops; it's all done with labels and the equivalent of `goto` and conditional `goto` statements. These are the branching opcodes, such as `Br` (branch unconditionally), `Brtrue` (branch if the value on the evaluation stack is `true`), and `Blt` (branch if the first value is less than the second value).

To set a branch target, first call `DefineLabel` (this returns a `Label` object), and then call `MarkLabel` at the place where you want to anchor the label. For example, consider the following C# code:

```
int x = 5;
while (x <= 10) Console.WriteLine (x++);
```

We can emit this as follows:

```
ILGenerator gen = ...

Label startLoop = gen.DefineLabel();                  // Declare labels
```

```
Label endLoop = gen.DefineLabel();

LocalBuilder x = gen.DeclareLocal (typeof (int));     // int x
gen.Emit (OpCodes.Ldc_I4, 5);                         //
gen.Emit (OpCodes.Stloc, x);                          // x = 5
gen.MarkLabel (startLoop);
  gen.Emit (OpCodes.Ldc_I4, 10);              // Load 10 onto eval stack
  gen.Emit (OpCodes.Ldloc, x);                // Load x onto eval stack

  gen.Emit (OpCodes.Blt, endLoop);              // if (x > 10)
goto endLoop

  gen.EmitWriteLine (x);                      // Console.WriteLine (x)

  gen.Emit (OpCodes.Ldloc, x);                // Load x onto eval stack
  gen.Emit (OpCodes.Ldc_I4, 1);               // Load 1 onto the stack
  gen.Emit (OpCodes.Add);                     // Add them together
  gen.Emit (OpCodes.Stloc, x);                // Save result back to x

  gen.Emit (OpCodes.Br, startLoop);             // return to
start of loop
gen.MarkLabel (endLoop);

gen.Emit (OpCodes.Ret);
```

## Instantiating Objects and Calling Instance Methods

The IL equivalent of `new` is the `Newobj` opcode. This takes a constructor and loads the constructed object onto the evaluation stack. For instance, the following constructs a `StringBuilder`:

```
var dynMeth = new DynamicMethod ("Test", null, null, typeof (void));
ILGenerator gen = dynMeth.GetILGenerator();

ConstructorInfo ci = typeof (StringBuilder).GetConstructor (new
```

```
Type[0]);
gen.Emit (OpCodes.Newobj, ci);
```

After loading an object onto the evaluation stack, you can use the `Call` or `Callvirt` opcode to invoke the object's instance methods. Extending this example, we'll query the `StringBuilder`'s `MaxCapacity` property by calling the property's get accessor and then write out the result:

```
gen.Emit (OpCodes.Callvirt, typeof (StringBuilder)
                            .GetProperty
("MaxCapacity").GetGetMethod());

gen.Emit (OpCodes.Call, typeof (Console).GetMethod ("WriteLine",
                                          new[] { typeof (int) } ));
gen.Emit (OpCodes.Ret);
dynMeth.Invoke (null, null);              // 2147483647
```

To emulate C# calling semantics:

- Use `Call` to invoke static methods and value type instance methods.
- Use `Callvirt` to invoke reference type instance methods (whether or not they're declared virtual).

In our example, we used `Callvirt` on the `StringBuilder` instance—even though `MaxProperty` is not virtual. This doesn't cause an error: it simply performs a nonvirtual call instead. Always invoking reference type instance methods with `Callvirt` avoids risking the opposite condition: invoking a virtual method with `Call`. (The risk is real. The author of the target method may later *change* its declaration.) `Callvirt` also has the benefit of checking that the receiver is non-null.

> **NOTE**
>
> Invoking a virtual method with `Call` bypasses virtual calling semantics, and calls that method directly. This is rarely desirable and, in effect, violates type safety.

In the following example, we construct a `StringBuilder` passing in two arguments, append `", world!"` to the `StringBuilder`, and then call `ToString` on it:

```
// We will call:   new StringBuilder ("Hello", 1000)

ConstructorInfo ci = typeof (StringBuilder).GetConstructor (
                     new[] { typeof (string), typeof (int) } );

gen.Emit (OpCodes.Ldstr, "Hello");   // Load a string onto the eval
stack
gen.Emit (OpCodes.Ldc_I4, 1000);     // Load an int onto the eval stack
gen.Emit (OpCodes.Newobj, ci);       // Construct the StringBuilder

Type[] strT = { typeof (string) };
gen.Emit (OpCodes.Ldstr, ", world!");
gen.Emit (OpCodes.Call, typeof (StringBuilder).GetMethod ("Append",
strT));
gen.Emit (OpCodes.Callvirt, typeof (object).GetMethod ("ToString"));
gen.Emit (OpCodes.Call, typeof (Console).GetMethod ("WriteLine", strT));
gen.Emit (OpCodes.Ret);
dynMeth.Invoke (null, null);        // Hello, world!
```

For fun we called `GetMethod` on `typeof(object)`, and then used `Callvirt` to perform a virtual method call on `ToString`. We could have gotten the same result by calling `ToString` on the `StringBuilder` type itself:

```
gen.Emit (OpCodes.Callvirt, typeof (StringBuilder).GetMethod
("ToString",
                                                          new Type[0]
));
```

(The empty type array is required in calling `GetMethod` because `StringBuilder` overloads `ToString` with another signature.)

> **NOTE**
>
> Had we called `object`'s `ToString` method nonvirtually:
>
> ```
> gen.Emit (OpCodes.Call,
>           typeof (object).GetMethod ("ToString"));
> ```
>
> the result would have been `System.Text.StringBuilder`. In other words, we would have circumvented `StringBuilder`'s `ToString` override and called `object`'s version directly.

## Exception Handling

`ILGenerator` provides dedicated methods for exception handling. Thus, the translation for this C# code:

```
try                               { throw new NotSupportedException(); }
catch (NotSupportedException ex)  { Console.WriteLine (ex.Message);    }
finally                           { Console.WriteLine ("Finally");     }
```

is this:

```
MethodInfo getMessageProp = typeof (NotSupportedException)
                            .GetProperty ("Message").GetGetMethod();

MethodInfo writeLineString = typeof (Console).GetMethod ("WriteLine",
                                             new[] { typeof (object) }
);
gen.BeginExceptionBlock();
  ConstructorInfo ci = typeof (NotSupportedException).GetConstructor (
                                                        new Type[0] );
  gen.Emit (OpCodes.Newobj, ci);
  gen.Emit (OpCodes.Throw);
gen.BeginCatchBlock (typeof (NotSupportedException));
  gen.Emit (OpCodes.Callvirt, getMessageProp);
  gen.Emit (OpCodes.Call, writeLineString);
gen.BeginFinallyBlock();
  gen.EmitWriteLine ("Finally");
gen.EndExceptionBlock();
```

Just as in C#, you can include multiple `catch` blocks. To rethrow the same exception, emit the `Rethrow` opcode.

> **NOTE**
>
> `ILGenerator` provides a helper method called `ThrowException`. This contains a bug, however, preventing it from being used with a `DynamicMethod`. It works only with a `MethodBuilder` (see the next section).

## Emitting Assemblies and Types

Although `DynamicMethod` is convenient, it can generate only methods. If you need to emit any other construct—or a complete type—you need to use the full “heavyweight” API. This means dynamically building an assembly and module. The assembly need not have a disk presence (in fact it cannot, because .NET Core 3 does not let you save generated assemblies to disk).

Let’s assume that we want to dynamically build a type. Because a type must reside in a module within an assembly, we first must create the assembly and module before we can create the type. This is the job of the `AssemblyBuilder` and `ModuleBuilder` types:

```
AssemblyName aname = new AssemblyName ("MyDynamicAssembly");

AssemblyBuilder assemBuilder =
  AssemblyBuilder.DefineDynamicAssembly (aname,
AssemblyBuilderAccess.Run);

ModuleBuilder modBuilder = assemBuilder.DefineDynamicModule
("DynModule");
```

> **NOTE**
>
> You can't add a type to an existing assembly, because an assembly is immutable after it's created.
>
> Dynamic assemblies are not garbage-collected and remain in memory until the process ends, unless you specify `AssemblyBuilderAccess.RunAndCollect` when defining the assembly. Various restrictions apply to collectible assemblies (see *http://albahari.com/dynamiccollect*).

After we have a module in which the type can reside, we can use `TypeBuilder` to create the type. The following defines a class called `Widget`:

```
TypeBuilder tb = modBuilder.DefineType ("Widget",
TypeAttributes.Public);
```

The `TypeAttributes` flags enum supports the CLR type modifiers you see when disassembling a type with *ildasm*. As well as member visibility flags, this includes type modifiers such as `Abstract` and `Sealed`—and `Interface` for defining a .NET interface. It also includes `Serializable`, which is equivalent to applying the `[Serializable]` attribute in C#, and `Explicit`, which is equivalent to applying `[StructLayout(LayoutKind.Explicit)]`. We describe how to apply other kinds of attributes later in this chapter, in "Attaching Attributes".

**NOTE**

The `DefineType` method also accepts an optional base type:

- To define a struct, specify a base type of `System.ValueType`.
- To define a delegate, specify a base type of `System.MulticastDelegate`.
- To implement an interface, use the constructor that accepts an array of interface types.
- To define an interface, specify `TypeAttributes.Interface | TypeAttributes.Abstract`.

Defining a delegate type requires a number of extra steps. In his weblog, Joel Pobar demonstrates how this is done in his article titled "Creating delegate types via Reflection.Emit."

We can now create members within the type:

```
MethodBuilder methBuilder = tb.DefineMethod ("SayHello",
                                             MethodAttributes.Public,
                                             null, null);
ILGenerator gen = methBuilder.GetILGenerator();
gen.EmitWriteLine ("Hello world");
gen.Emit (OpCodes.Ret);
```

We're now ready to create the type, which finalizes its definition:

```
Type t = tb.CreateType();
```

After the type is created, we can use ordinary reflection to inspect and perform late binding:

```
object o = Activator.CreateInstance (t);
t.GetMethod ("SayHello").Invoke (o, null);        // Hello world
```

## The Reflection.Emit Object Model

Figure 19-2 illustrates the essential types in `System.Reflection.Emit`. Each type describes a CLR construct and is based on a counterpart in the `System.Reflection` namespace. This allows you to use emitted constructs in place of normal constructs when building a type. For example, we previously called `Console.WriteLine` as follows:

```
MethodInfo writeLine = typeof(Console).GetMethod ("WriteLine",
                                       new Type[] { typeof (string) });
gen.Emit (OpCodes.Call, writeLine);
```

We could just as easily call a dynamically generated method by calling `gen.Emit` with a `MethodBuilder` instead of a `MethodInfo`. This is essential—otherwise, you couldn’t write one dynamic method that called another in the same type.

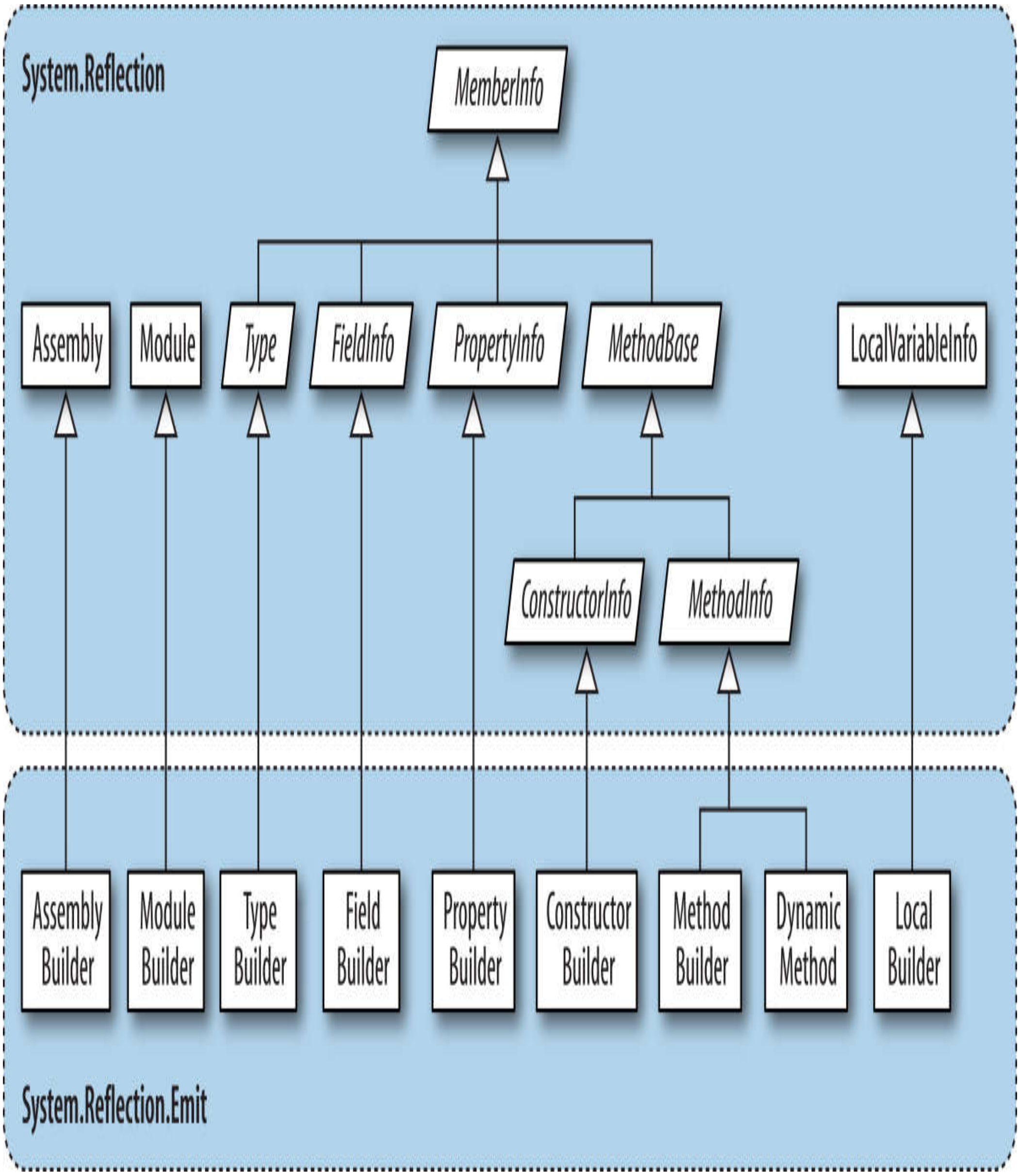


*Figure 19-2.* `System.Reflection.Emit`

Recall that you must call `CreateType` on a `TypeBuilder` when you've finished populating it. Calling `CreateType` seals the `TypeBuilder` and all its members—so nothing more can be added or changed—and gives you back a real `Type` that you can instantiate.

Before you call `CreateType`, the `TypeBuilder` and its members are in an *uncreated* state. There are significant restrictions on what you can do with uncreated constructs. In particular, you cannot call any of the members that return `MemberInfo` objects, such as `GetMembers`, `GetMethod`, or `GetProperty`—these all throw an exception. If you want to refer to members of an uncreated type, you must use the original emissions:

```
TypeBuilder tb = ...

MethodBuilder method1 = tb.DefineMethod ("Method1", ...);
MethodBuilder method2 = tb.DefineMethod ("Method2", ...);

ILGenerator gen1 = method1.GetILGenerator();

// Suppose we want method1 to call method2:

gen1.Emit (OpCodes.Call, method2);                    // Right
gen1.Emit (OpCodes.Call, tb.GetMethod ("Method2"));   // Wrong
```

After calling `CreateType`, you can reflect on and activate not only the `Type` returned, but also the original `TypeBuilder` object. The `TypeBuilder`, in fact, morphs into a proxy for the real `Type`. You'll see why this feature is important in "Awkward Emission Targets".

## Emitting Type Members

All the examples in this section assume a `TypeBuilder`, `tb`, has been instantiated as follows:

```
AssemblyName aname = new AssemblyName ("MyEmissions");
```

```
AssemblyBuilder assemBuilder = AssemblyBuilder.DefineDynamicAssembly (
  aname, AssemblyBuilderAccess.Run);

ModuleBuilder modBuilder = assemBuilder.DefineDynamicModule
("MainModule");

TypeBuilder tb = modBuilder.DefineType ("Widget",
TypeAttributes.Public);
```

## Emitting Methods

You can specify a return type and parameter types when calling `DefineMethod`, in the same manner as when instantiating a `DynamicMethod`. For instance, the following method:

```
public static double SquareRoot (double value) => Math.Sqrt (value);
```

can be generated like this:

```
MethodBuilder mb = tb.DefineMethod ("SquareRoot",
  MethodAttributes.Static | MethodAttributes.Public,
  CallingConventions.Standard,
  typeof (double),                          // Return type
  new[]  { typeof (double) } );             // Parameter types

mb.DefineParameter (1, ParameterAttributes.None, "value");  // Assign
name

ILGenerator gen = mb.GetILGenerator();
gen.Emit (OpCodes.Ldarg_0);                                // Load 1st
arg
gen.Emit (OpCodes.Call, typeof(Math).GetMethod ("Sqrt"));
gen.Emit (OpCodes.Ret);
```

```
Type realType = tb.CreateType();
double x = (double) tb.GetMethod ("SquareRoot").Invoke (null,
                                                new object[] { 10.0 });
Console.WriteLine (x);   // 3.16227766016838
```

Calling `DefineParameter` is optional and is typically done to assign the parameter a name. The number 1 refers to the first parameter (0 refers to the return value). If you call `DefineParameter`, the parameter is implicitly named `__p1`, `__p2`, and so on. Assigning names makes sense if you will write the assembly to disk; it makes your methods friendly to consumers.

> **NOTE**
>
> `DefineParameter` returns a `ParameterBuilder` object upon which you can call `SetCustomAttribute` to attach attributes (see "Attaching Attributes").

To emit pass-by-reference parameters, such as in the following C# method:

```
public static void SquareRoot (ref double value)
  => value = Math.Sqrt (value);
```

call `MakeByRefType` on the parameter type(s):

```
MethodBuilder mb = tb.DefineMethod ("SquareRoot",
  MethodAttributes.Static | MethodAttributes.Public,
  CallingConventions.Standard,
  null,
  new Type[] { typeof (double).MakeByRefType() } );
```

```
mb.DefineParameter (1, ParameterAttributes.None, "value");

ILGenerator gen = mb.GetILGenerator();
gen.Emit (OpCodes.Ldarg_0);
gen.Emit (OpCodes.Ldarg_0);
gen.Emit (OpCodes.Ldind_R8);
gen.Emit (OpCodes.Call, typeof (Math).GetMethod ("Sqrt"));
gen.Emit (OpCodes.Stind_R8);
gen.Emit (OpCodes.Ret);

Type realType = tb.CreateType();
object[] args = { 10.0 };
tb.GetMethod ("SquareRoot").Invoke (null, args);
Console.WriteLine (args[0]);                    // 3.16227766016838
```

The opcodes here were copied from a disassembled C# method. Notice the difference in semantics for accessing parameters passed by reference: `Ldind` and `Stind` mean "load indirectly" and "store indirectly," respectively. The R8 suffix means an eight-byte floating-point number.

The process for emitting `out` parameters is identical, except that you call `DefineParameter` as follows:

```
mb.DefineParameter (1, ParameterAttributes.Out, "value");
```

## GENERATING INSTANCE METHODS

To generate an instance method, specify `MethodAttributes.Instance` when calling `DefineMethod`:

```
MethodBuilder mb = tb.DefineMethod ("SquareRoot",
```

```
MethodAttributes.Instance | MethodAttributes.Public
...
```

With instance methods, argument zero is implicitly `this`; the remaining arguments start at 1. So, `Ldarg_0` loads `this` onto the evaluation stack; `Ldarg_1` loads the first real method argument.

### OVERRIDING METHODS

Overriding a virtual method in a base class is easy: simply define a method with an identical name, signature, and return type, specifying `MethodAttributes.Virtual` when calling `DefineMethod`. The same applies when implementing interface methods.

`TypeBuilder` also exposes a method called `DefineMethodOverride`, which overrides a method with a different name. This makes sense only with explicit interface implementation; in other scenarios, use `DefineMethod`.

### HIDEBYSIG

If you're subclassing another type, it's nearly always worth specifying `MethodAttributes.HideBySig` when defining methods. `HideBySig` ensures that C#-style method hiding semantics are applied, which is that a base method is hidden only if a subtype defines a method with an identical *signature*. Without `HideBySig`, method hiding considers only the *name*, so `Foo(string)` in the subtype will hide `Foo()` in the base type, which is generally undesirable.

## Emitting Fields and Properties

To create a field, you call `DefineField` on a `TypeBuilder`, specifying the desired field name, type, and visibility. The following creates a private integer field called *length*:

```
FieldBuilder field = tb.DefineField ("length", typeof (int),
                                     FieldAttributes.Private);
```

Creating a property or indexer requires a few more steps. First, call `DefineProperty` on a `TypeBuilder`, providing it with the name and type of the property:

```
PropertyBuilder prop = tb.DefineProperty (
                         "Text",                      // Name of
property
                         PropertyAttributes.None,
                         typeof (string),             // Property type
                         new Type[0]                  // Indexer types
                       );
```

(If you're writing an indexer, the final argument is an array of indexer types.) Note that we haven't specified the property visibility: this is done individually on the accessor methods.

The next step is to write the `get` and `set` methods. By convention, their names are prefixed with "get_" or "set_". You then attach them to the property by calling `SetGetMethod` and `SetSetMethod` on the `PropertyBuilder`.

To give a complete example, let's take the following field and property declaration:

```
string _text;
public string Text
{
  get          => _text;
  internal set => _text = value;
}
```

and generate it dynamically:

```
FieldBuilder field = tb.DefineField ("_text", typeof (string),
                                      FieldAttributes.Private);
PropertyBuilder prop = tb.DefineProperty (
                         "Text",                      // Name of
property
                         PropertyAttributes.None,
                         typeof (string),             // Property type
                         new Type[0]);                // Indexer types

MethodBuilder getter = tb.DefineMethod (
  "get_Text",                                         // Method name
  MethodAttributes.Public | MethodAttributes.SpecialName,
  typeof (string),                                    // Return type
  new Type[0]);                                       // Parameter types

ILGenerator getGen = getter.GetILGenerator();
getGen.Emit (OpCodes.Ldarg_0);        // Load "this" onto eval stack
getGen.Emit (OpCodes.Ldfld, field);   // Load field value onto eval
stack
getGen.Emit (OpCodes.Ret);            // Return

MethodBuilder setter = tb.DefineMethod (
  "set_Text",
  MethodAttributes.Assembly | MethodAttributes.SpecialName,
  null,                                                 // Return type
  new Type[] { typeof (string) } );                     // Parameter
types
```

```
ILGenerator setGen = setter.GetILGenerator();
setGen.Emit (OpCodes.Ldarg_0);        // Load "this" onto eval stack
setGen.Emit (OpCodes.Ldarg_1);        // Load 2nd arg, i.e., value
setGen.Emit (OpCodes.Stfld, field);   // Store value into field
setGen.Emit (OpCodes.Ret);            // return

prop.SetGetMethod (getter);           // Link the get method and
property
prop.SetSetMethod (setter);           // Link the set method and
property
```

We can test the property as follows:

```
Type t = tb.CreateType();
object o = Activator.CreateInstance (t);
t.GetProperty ("Text").SetValue (o, "Good emissions!", new object[0]);
string text = (string) t.GetProperty ("Text").GetValue (o, null);

Console.WriteLine (text);             // Good emissions!
```

Notice that in defining the accessor `MethodAttributes`, we included `SpecialName`. This instructs compilers to disallow direct binding to these methods when statically referencing the assembly. It also ensures that the accessors are handled appropriately by reflection tools and Visual Studio’s IntelliSense.

> **NOTE**
>
> You can emit events in a similar manner, by calling `DefineEvent` on a `TypeBuilder`. You then write explicit event accessor methods and attach them to the `EventBuilder` by calling `SetAddOnMethod` and `SetRemoveOnMethod`.

## Emitting Constructors

You can define your own constructors by calling `DefineConstructor` on a type builder. You’re not obliged to do so—a default parameterless constructor is automatically provided if you don’t. The default constructor calls the base class constructor if subtyping, just like in C#. Defining one or more constructors displaces this default constructor.

If you need to initialize fields, the constructor’s a good spot. In fact, it’s the only spot: C#’s field initializers don’t have special CLR support—they are simply a syntactic shortcut for assigning values to fields in the constructor.

So, to reproduce this:

```
class Widget
{
  int _capacity = 4000;
}
```

you would define a constructor as follows:

```
FieldBuilder field = tb.DefineField ("_capacity", typeof (int),
                                      FieldAttributes.Private);
ConstructorBuilder c = tb.DefineConstructor (
  MethodAttributes.Public,
  CallingConventions.Standard,
  new Type[0]);                  // Constructor parameters

ILGenerator gen = c.GetILGenerator();
```

```
gen.Emit (OpCodes.Ldarg_0);            // Load "this" onto eval stack
gen.Emit (OpCodes.Ldc_I4, 4000);       // Load 4000 onto eval stack
gen.Emit (OpCodes.Stfld, field);       // Store it to our field
gen.Emit (OpCodes.Ret);
```

### CALLING BASE CONSTRUCTORS

If subclassing another type, the constructor we just wrote would *circumvent the base class constructor*. This is unlike C#, in which the base class constructor is always called, whether directly or indirectly. For instance, given the following code:

```
class A     { public A() { Console.Write ("A"); } }
class B : A { public B() {} }
```

the compiler, in effect, will translate the second line into this:

```
class B : A { public B() : base() {} }
```

This is not the case when generating IL: you must explicitly call the base constructor if you want it to execute (which nearly always, you do). Assuming the base class is called `A`, here's how to do it:

```
gen.Emit (OpCodes.Ldarg_0);
ConstructorInfo baseConstr = typeof (A).GetConstructor (new Type[0]);
gen.Emit (OpCodes.Call, baseConstr);
```

Calling constructors with arguments is just the same as with methods.

## Attaching Attributes

You can attach custom attributes to a dynamic construct by calling `SetCustomAttribute` with a `CustomAttributeBuilder`. For example, suppose that we want to attach the following attribute declaration to a field or property:

```
[XmlElement ("FirstName", Namespace="http://test/", Order=3)]
```

This relies on the `XmlElementAttribute` constructor that accepts a single string. To use `CustomAttributeBuilder`, we must retrieve this constructor as well as the two additional properties that we want to set (`Namespace` and `Order`):

```
Type attType = typeof (XmlElementAttribute);

ConstructorInfo attConstructor = attType.GetConstructor (
  new Type[] { typeof (string) } );

var att = new CustomAttributeBuilder (
  attConstructor,                         // Constructor
  new object[] { "FirstName" },           // Constructor arguments
  new PropertyInfo[]
  {
    attType.GetProperty ("Namespace"),    // Properties
    attType.GetProperty ("Order")
  },
  new object[] { "http://test/", 3 }      // Property values
);

myFieldBuilder.SetCustomAttribute (att);
// or propBuilder.SetCustomAttribute (att);
// or typeBuilder.SetCustomAttribute (att);  etc
```

# Emitting Generic Methods and Types

All the examples in this section assume that `modBuilder` has been instantiated as follows:

```
AssemblyName aname = new AssemblyName ("MyEmissions");

AssemblyBuilder assemBuilder = AssemblyBuilder.DefineDynamicAssembly (
  aname, AssemblyBuilderAccess.Run);

ModuleBuilder modBuilder = assemBuilder.DefineDynamicModule
("MainModule");
```

## Defining Generic Methods

Follow these steps to emit a generic method:

1. Call `DefineGenericParameters` on a `MethodBuilder` to obtain an array of `GenericTypeParameterBuilder` objects.
2. Call `SetSignature` on a `MethodBuilder` using these generic type parameters.
3. Optionally, name the parameters as you would otherwise.

For example, the following generic method:

```
public static T Echo<T> (T value)
{
  return value;
}
```

can be emitted like this:

```
TypeBuilder tb = modBuilder.DefineType ("Widget",
TypeAttributes.Public);

MethodBuilder mb = tb.DefineMethod ("Echo", MethodAttributes.Public |
                                            MethodAttributes.Static);
GenericTypeParameterBuilder[] genericParams
  = mb.DefineGenericParameters ("T");

mb.SetSignature (genericParams[0],     // Return type
                 null, null,
                 genericParams,        // Parameter types
                 null, null);

mb.DefineParameter (1, ParameterAttributes.None, "value");   // Optional

ILGenerator gen = mb.GetILGenerator();
gen.Emit (OpCodes.Ldarg_0);
gen.Emit (OpCodes.Ret);
```

The `DefineGenericParameters` method accepts any number of string arguments—these correspond to the desired generic type names. In this example, we needed just one generic type called `T`. `GenericTypeParameterBuilder` is based on `System.Type`, so you can use it in place of a `TypeBuilder` when emitting opcodes.

`GenericTypeParameterBuilder` also lets you specify a base type constraint:

```
genericParams[0].SetBaseTypeConstraint (typeof (Foo));
```

and interface constraints:

```
genericParams[0].SetInterfaceConstraints (typeof (IComparable));
```

To replicate this:

```
public static T Echo<T> (T value) where T : IComparable<T>
```

you would write:

```
genericParams[0].SetInterfaceConstraints (
  typeof (IComparable<>).MakeGenericType (genericParams[0]) );
```

For other kinds of constraints, call `SetGenericParameterAttributes`. This accepts a member of the `GenericParameterAttributes` enum, which includes the following values:

```
DefaultConstructorConstraint
NotNullableValueTypeConstraint
ReferenceTypeConstraint
Covariant
Contravariant
```

The last two are equivalent to applying the `out` and `in` modifiers to the type parameters.

## Defining Generic Types

You can define generic types in a similar fashion. The difference is that you call `DefineGenericParameters` on the `TypeBuilder` rather than the `MethodBuilder`. So, to reproduce this:

```
public class Widget<T>
{
  public T Value;
}
```

you would do the following:

```
TypeBuilder tb = modBuilder.DefineType ("Widget",
TypeAttributes.Public);

GenericTypeParameterBuilder[] genericParams
  = tb.DefineGenericParameters ("T");

tb.DefineField ("Value", genericParams[0], FieldAttributes.Public);
```

Generic constraints can be added, just as with a method.

# Awkward Emission Targets

All of the examples in this section assume that a `modBuilder` has been instantiated as in previous sections.

## Uncreated Closed Generics

Suppose that you want to emit a method that uses a closed generic type:

```
public class Widget
{
```

```
  public static void Test() { var list = new List<int>(); }
}
```

The process is fairly straightforward:

```
TypeBuilder tb = modBuilder.DefineType ("Widget",
TypeAttributes.Public);

MethodBuilder mb = tb.DefineMethod ("Test", MethodAttributes.Public |
                                            MethodAttributes.Static);
ILGenerator gen = mb.GetILGenerator();

Type variableType = typeof (List<int>);

ConstructorInfo ci = variableType.GetConstructor (new Type[0]);

LocalBuilder listVar = gen.DeclareLocal (variableType);
gen.Emit (OpCodes.Newobj, ci);
gen.Emit (OpCodes.Stloc, listVar);
gen.Emit (OpCodes.Ret);
```

Now suppose that instead of a list of integers, we want a list of widgets:

```
public class Widget
{
  public static void Test() { var list = new List<Widget>(); }
}
```

In theory, this is a simple modification; all we do is replace this line:

```
Type variableType = typeof (List<int>);
```

with this one:

```
Type variableType = typeof (List<>).MakeGenericType (tb);
```

Unfortunately, this causes a `NotSupportedException` to be thrown when we then call `GetConstructor`. The problem is that you cannot call `GetConstructor` on a generic type closed with an uncreated type builder. The same goes for `GetField` and `GetMethod`.

The solution is unintuitive. `TypeBuilder` provides three static methods:

```
public static ConstructorInfo GetConstructor (Type, ConstructorInfo);
public static FieldInfo       GetField       (Type, FieldInfo);
public static MethodInfo      GetMethod      (Type, MethodInfo);
```

Although it doesn’t appear so, these methods exist specifically to obtain members of generic types closed with uncreated type builders! The first parameter is the closed generic type; the second parameter is the member that you want on the *unbound* generic type. Here’s the corrected version of our example:

```
MethodBuilder mb = tb.DefineMethod ("Test", MethodAttributes.Public |
                                             MethodAttributes.Static);
ILGenerator gen = mb.GetILGenerator();

Type variableType = typeof (List<>).MakeGenericType (tb);

ConstructorInfo unbound = typeof (List<>).GetConstructor
(new Type[0]);
```

```
ConstructorInfo ci = TypeBuilder.GetConstructor
(variableType, unbound);

LocalBuilder listVar = gen.DeclareLocal (variableType);
gen.Emit (OpCodes.Newobj, ci);
gen.Emit (OpCodes.Stloc, listVar);
gen.Emit (OpCodes.Ret);
```

## Circular Dependencies

Suppose that you want to build two types that reference each other, such as these:

```
class A { public B Bee; }
class B { public A Aye; }
```

You can generate this dynamically:

```
var publicAtt = FieldAttributes.Public;

TypeBuilder aBuilder = modBuilder.DefineType ("A");
TypeBuilder bBuilder = modBuilder.DefineType ("B");

FieldBuilder bee = aBuilder.DefineField ("Bee", bBuilder, publicAtt);
FieldBuilder aye = bBuilder.DefineField ("Aye", aBuilder, publicAtt);

Type realA = aBuilder.CreateType();
Type realB = bBuilder.CreateType();
```

Notice that we didn’t call `CreateType` on `aBuilder` or `bBuilder` until we populated both objects. The principle is this: first hook everything up, and then call `CreateType` on each type builder.

Interestingly, the `realA` type is valid but *dysfunctional* until you call `CreateType` on `bBuilder`. (If you started using `aBuilder` prior to this, an exception would be thrown when you tried to access field `Bee`.)

You might wonder how `bBuilder` knows to *fix up* `realA` after creating `realB`. The answer is that it doesn't: `realA` can fix *itself* the next time it's used. This is possible because after calling `CreateType`, a `TypeBuilder` morphs into a proxy for the real runtime type. So, `realA`, with its references to `bBuilder`, can easily obtain the metadata it needs for the upgrade.

This system works when the type builder demands simple information of the unconstructed type—information that can be *predetermined*—such as type, member, and object references. In creating `realA`, the type builder doesn't need to know, for instance, how many bytes `realB` will eventually occupy in memory. This is just as well because `realB` has not yet been created! But now imagine that `realB` was a struct. The final size of `realB` is now critical information in creating `realA`.

If the relationship is noncyclical; for instance:

```
struct A { public B Bee; }
struct B {               }
```

you can solve this by first creating struct `B` and then struct `A`. But consider this:

```
struct A { public B Bee; }
struct B { public A Aye; }
```

We won't try to emit this because it's nonsensical to have two structs contain each other (C# generates a compile-time error if you try). But the following variation is both legal and useful:

```
public struct S<T> { ... }    // S can be empty and this demo will work.

class A { S<B> Bee; }
class B { S<A> Aye; }
```

In creating `A`, a `TypeBuilder` now needs to know the memory footprint of `B`, and vice versa. To illustrate, let's assume that struct `S` is defined statically. Here's the code to emit classes `A` and `B`:

```
var pub = FieldAttributes.Public;

TypeBuilder aBuilder = modBuilder.DefineType ("A");
TypeBuilder bBuilder = modBuilder.DefineType ("B");

aBuilder.DefineField ("Bee", typeof(S<>).MakeGenericType (bBuilder),
pub);
bBuilder.DefineField ("Aye", typeof(S<>).MakeGenericType (aBuilder),
pub);

Type realA = aBuilder.CreateType();    // Error: cannot load type B
Type realB = bBuilder.CreateType();
```

`CreateType` now throws a `TypeLoadException` no matter in which order you go:

- Call `aBuilder.CreateType` first and it says "cannot load type `B`".
- Call `bBuilder.CreateType` first and it says "cannot load type `A`"!

To solve this, you must allow the type builder to create `realB` partway through creating `realA`. You do this by handling the `TypeResolve` event on the `AppDomain` class just before calling `CreateType`. So, in our example, we replace the last two lines with this:

```
TypeBuilder[] uncreatedTypes = { aBuilder, bBuilder };

ResolveEventHandler handler = delegate (object o, ResolveEventArgs args)
{
  var type = uncreatedTypes.FirstOrDefault (t => t.FullName ==
args.Name);
  return type == null ? null : type.CreateType().Assembly;
};

AppDomain.CurrentDomain.TypeResolve += handler;

Type realA = aBuilder.CreateType();
Type realB = bBuilder.CreateType();

AppDomain.CurrentDomain.TypeResolve -= handler;
```

The `TypeResolve` event fires during the call to `aBuilder.CreateType`, at the point when it needs you to call `CreateType` on `bBuilder`.

> **NOTE**
>
> Handling the `TypeResolve` event as in this example is also necessary when defining a nested type, when the nested and parent types refer to each other.

# Parsing IL

You can obtain information about the content of an existing method by calling `GetMethodBody` on a `MethodBase` object. This returns a `MethodBody` object that has properties for inspecting a method's local variables, exception handling clauses, stack size, as well as the raw IL. Rather like the reverse of `Reflection.Emit`!

Inspecting a method's raw IL can be useful in profiling code. A simple use would be to determine which methods in an assembly have changed when an assembly is updated.

To illustrate parsing IL, we'll write an application that disassembles IL in the style of *ildasm*. This could be used as the starting point for a code analysis tool or a higher-level language disassembler.

> **NOTE**
>
> Remember that in the reflection API, all of C#'s functional constructs are either represented by a `MethodBase` subtype, or (in the case of properties, events, and indexers) have `MethodBase` objects attached to them.

## Writing a Disassembler

Here is a sample of the output that our disassembler will produce:

```
IL_00EB:  ldfld        Disassembler._pos
IL_00F0:  ldloc.2
IL_00F1:  add
IL_00F2:  ldelema      System.Byte
IL_00F7:  ldstr        "Hello world"
IL_00FC:  call         System.Byte.ToString
IL_0101:  ldstr        " "
IL_0106:  call         System.String.Concat
```

To obtain this output, we must parse the binary tokens that make up the IL. The first step is to call the `GetILAsByteArray` method on `MethodBody` to obtain the IL as a byte array. To make the rest of the job easier, we will write this into a class as follows:

```
public class Disassembler
{
  public static string Disassemble (MethodBase method)
    => new Disassembler (method).Dis();

  StringBuilder _output;    // The result to which we'll keep appending
  Module _module;           // This will come in handy later
  byte[] _il;               // The raw byte code
  int _pos;                 // The position we're up to in the byte code

  Disassembler (MethodBase method)
  {
    _module = method.DeclaringType.Module;
    _il = method.GetMethodBody().GetILAsByteArray();
  }

  string Dis()
  {
    _output = new StringBuilder();
```

```
    while (_pos < _il.Length) DisassembleNextInstruction();
    return _output.ToString();
  }
}
```

The static `Disassemble` method will be the only public member of this class. All other members will be private to the disassembly process. The `Dis` method contains the *main* loop where we process each instruction.

With this skeleton in place, all that remains is to write `DisassembleNextInstruction`. But before doing so, it will help to load all the opcodes into a static dictionary so that we can access them by their 8- or 16-bit value. The easiest way to accomplish this is to use reflection to retrieve all the static fields whose type is `OpCode` in the `OpCodes` class:

```
static Dictionary<short,OpCode> _opcodes = new Dictionary<short,OpCode>
();

static Disassembler()
{
  Dictionary<short, OpCode> opcodes = new Dictionary<short, OpCode>();
    foreach (FieldInfo fi in typeof (OpCodes).GetFields
                             (BindingFlags.Public |
BindingFlags.Static))
      if (typeof (OpCode).IsAssignableFrom (fi.FieldType))
      {
        OpCode code = (OpCode) fi.GetValue (null);   // Get field's
value
        if (code.OpCodeType != OpCodeType.Nternal)
          _opcodes.Add (code.Value, code);
      }
}
```

We’ve written it in a static constructor so that it executes just once.

Now we can write `DisassembleNextInstruction`. Each IL instruction consists of a one- or two-byte opcode, followed by an operand of zero, one, two, four, or eight bytes. (An exception is inline switch opcodes, which are followed by a variable number of operands.) So, we read the opcode, then the operand, and then write out the result:

```
void DisassembleNextInstruction()
{
  int opStart = _pos;

  OpCode code = ReadOpCode();
  string operand = ReadOperand (code);

  _output.AppendFormat ("IL_{0:X4}:  {1,-12} {2}",
                        opStart, code.Name, operand);
  _output.AppendLine();
}
```

To read an opcode, we advance one byte and see whether we have a valid instruction. If not, we advance another byte and look for a two-byte instruction:

```
OpCode ReadOpCode()
{
  byte byteCode = _il [_pos++];
  if (_opcodes.ContainsKey (byteCode)) return _opcodes [byteCode];

  if (_pos == _il.Length)  throw new Exception ("Unexpected end of IL");

  short shortCode = (short) (byteCode * 256 + _il [_pos++]);
```

```
  if (!_opcodes.ContainsKey (shortCode))
    throw new Exception ("Cannot find opcode " + shortCode);

  return _opcodes [shortCode];
}
```

To read an operand, we first must establish its length. We can do this based on the operand type. Because most are four bytes long, we can filter out the exceptions fairly easily in a conditional clause.

The next step is to call `FormatOperand`, which attempts to format the operand:

```
string ReadOperand (OpCode c)
{
  int operandLength =
    c.OperandType == OperandType.InlineNone
      ? 0 :
    c.OperandType == OperandType.ShortInlineBrTarget ||
    c.OperandType == OperandType.ShortInlineI ||
    c.OperandType == OperandType.ShortInlineVar
      ? 1 :
    c.OperandType == OperandType.InlineVar
      ? 2 :
    c.OperandType == OperandType.InlineI8 ||
    c.OperandType == OperandType.InlineR
      ? 8 :
    c.OperandType == OperandType.InlineSwitch
      ? 4 * (BitConverter.ToInt32 (_il, _pos) + 1) :
      4;  // All others are 4 bytes

  if (_pos + operandLength > _il.Length)
    throw new Exception ("Unexpected end of IL");

  string result = FormatOperand (c, operandLength);
```

```
  if (result == null)
  {                        // Write out operand bytes in hex
    result = "";
    for (int i = 0; i < operandLength; i++)
      result += _il [_pos + i].ToString ("X2") + " ";
  }
  _pos += operandLength;
  return result;
}
```

If the `result` of calling `FormatOperand` is `null`, it means the operand needs no special formatting, so we simply write it out in hexadecimal. We could test the disassembler at this point by writing a `FormatOperand` method that always returns `null`. Here's what the output would look like:

```
IL_00A8:  ldfld        98 00 00 04
IL_00AD:  ldloc.2
IL_00AE:  add
IL_00AF:  ldelema      64 00 00 01
IL_00B4:  ldstr        26 04 00 70
IL_00B9:  call         B6 00 00 0A
IL_00BE:  ldstr        11 01 00 70
IL_00C3:  call         91 00 00 0A
...
```

Although the opcodes are correct, the operands are not much use. Instead of hexadecimal numbers, we want member names and strings. The `FormatOperand` method, when it's written, will address this—identifying the special cases that benefit from such formatting. These comprise most four-byte operands and the short branch instructions:

```
string FormatOperand (OpCode c, int operandLength)
```

```
{
  if (operandLength == 0) return "";

  if (operandLength == 4)
    return Get4ByteOperand (c);
  else if (c.OperandType == OperandType.ShortInlineBrTarget)
    return GetShortRelativeTarget();
  else if (c.OperandType == OperandType.InlineSwitch)
    return GetSwitchTarget (operandLength);
  else
    return null;
}
```

There are three kinds of four-byte operands that we treat specially. The first is references to members or types—with these, we extract the member or type name by calling the defining module's `ResolveMember` method. The second case is strings—these are stored in the assembly module's metadata and can be retrieved by calling `ResolveString`. The final case is branch targets, where the operand refers to a byte offset in the IL. We format these by working out the absolute address *after* the current instruction (+ four bytes):

```
string Get4ByteOperand (OpCode c)
{
  int intOp = BitConverter.ToInt32 (_il, _pos);

  switch (c.OperandType)
  {
    case OperandType.InlineTok:
    case OperandType.InlineMethod:
    case OperandType.InlineField:
    case OperandType.InlineType:
      MemberInfo mi;
      try   { mi = _module.ResolveMember (intOp); }
```

```
      catch { return null; }
      if (mi == null) return null;

      if (mi.ReflectedType != null)
        return mi.ReflectedType.FullName + "." + mi.Name;
      else if (mi is Type)
        return ((Type)mi).FullName;
      else
        return mi.Name;

    case OperandType.InlineString:
      string s = _module.ResolveString (intOp);
      if (s != null) s = "'" + s + "'";
      return s;

    case OperandType.InlineBrTarget:
      return "IL_" + (_pos + intOp + 4).ToString ("X4");

    default:
      return null;
  }
}
```

> **NOTE**
>
> The point where we call `ResolveMember` is a good window for a code analysis tool that reports on method dependencies.

For any other four-byte opcode, we return `null` (this will cause `ReadOperand` to format the operand as hex digits).

The final kinds of operand that need special attention are short branch targets and inline switches. A short branch target describes the destination offset as a single signed byte, as at the end of the current instruction (i.e., + one byte). A switch target is followed by a variable number of four-byte branch destinations:

```
string GetShortRelativeTarget()
{
  int absoluteTarget = _pos + (sbyte) _il [_pos] + 1;
  return "IL_" + absoluteTarget.ToString ("X4");
}

string GetSwitchTarget (int operandLength)
{
  int targetCount = BitConverter.ToInt32 (_il, _pos);
  string [] targets = new string [targetCount];
  for (int i = 0; i < targetCount; i++)
  {
    int ilTarget = BitConverter.ToInt32 (_il, _pos + (i + 1) * 4);
    targets [i] = "IL_" + (_pos + ilTarget + operandLength).ToString
("X4");
  }
  return "(" + string.Join (", ", targets) + ")";
}
```

This completes the disassembler. We can test it by disassembling one of its own methods:

```
MethodInfo mi = typeof (Disassembler).GetMethod (
  "ReadOperand", BindingFlags.Instance | BindingFlags.NonPublic);

Console.WriteLine (Disassembler.Disassemble (mi));
```