# Understanding the Common Type System (CTS)

3


This chapter deals with the object model of the common language runtime, the *CTS*. It is this shared framework that enables different languages to work together at a higher level than the binary compatibility that traditional *COM* uses.

By sharing the same object model it is possible for components written in one language to inherit behavior from components written in another. Rather than just being able to call library functions written in another language, in the *.NET* framework components may pass objects to one another and extend each others capabilities.

## The Virtual Object System

We may classify all types in *.NET* into two categories, as shown in Figure 3.1. There are the **value types**, including the built-in scalar types, and user defined enumerations and structures. The **reference types** include all pointer types and object references.

Reference types have the fundamental property that assignments of such values has alias semantics. If we copy a reference and modify the datum to which the reference refers, then both references will refer to the modified datum. Value types, by contrast have value

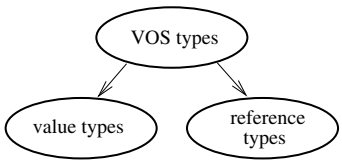


**Figure 3.1:** *CTS* types are either value types or reference types

semantics for assigment. If we make a copy of a value structure and modify one copy, the other copy is unchanged.

There is another useful categorization of types in the *CTS*, which is the separation between *self-describing* types and *non-self-describing* types. The difference is fundamental. If we have an instance of a self-describing type then the value carries with it some denotation of the *exact type* of the value. Values of a non-self-describing type are nothing more than a bunch of bits. For example, there is no sure way of telling whether a particular value in a memory word is a signed or an unsigned integer. In a statically typed language the compiler will be able to ensure that only values of the declared type are placed in the word, but there is no way we can tell the type from the bit-pattern.

In the *CTS*, only the object types are self-describing. These are a subcategory of the reference types, described below.

## Value Types

The subdivision of value types is shown in Figure 3.2. At the level of granularity that we consider, there are four kinds of value types. The two leftmost leaves in the figure are built-in types, while the two on the right are user-defined.

***The built-in types.*** On the left of Figure 3.2 are the scalar types. These have no substructure, and have special encodings in *CIL*. The scalar types include all of the primitive types shown in Figure 2.1—that is, all of the arithmetic types, together with the Boolean and character types.

As we shall see later, some reference types carry type information with their values. These are the *self-describing* types. Other references may be of a known type as a result

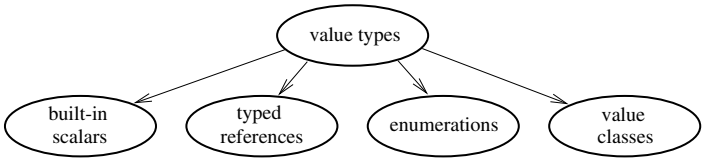


**Figure 3.2:** Hierarchy of value types in the *CTS*

of invariant properties that are enforced by the compiler. When such references need type information to be attached to their values, we use *typed references*. Typed references are value types that aggregate a reference and a type denotation. There are special instructions in *IL* for creating these special values and extracting their components.

***Value classes.*** Value classes are aggregate types, but are not self-describing. Assignment of value classes has value semantics. In *C#*, declaration of **struct**s results in value classes appearing in the *IL*. Value classes may be used when the full capabilities of the object types is not required. The creation and manipulation of values of a value class have much lower resource use than is the case for object types.

Here is a first example in *C#* —

```
public struct ValCls { public int i,j,k; }
```

The resulting textual *CIL* is shown in Figure 3.3. Although in *C#* we use different keywords to define a **struct** or a **class**, in *CIL* they are both classes. The distinction between these two kinds of class is carried by the *value* attribute in the class definition. Notice that the class has been given the *sealed* attribute, and has been given the base class `[mscorlib]System.ValueType`. This system type is the supertype of all value classes.

```
.class value public auto ansi sealed ValCls
       extends [mscorlib]System.ValueType
{
  .field public int32 i
  .field public int32 j
  .field public int32 k
} // end of class ValCls
```

**Figure 3.3:** Disassembly of **struct** definition

In *CIL* value classes must always be declared as inheriting from *System.ValueType* and must always have the **sealed** attribute. That is to say, value types cannot be further extended.

Every value type has an associated *boxed type*. The boxed type is an object type and carries type information with the value. In truth, it is the associated boxed type that really inherits from *System.ValueType*.

It is possible to declare methods that are bound to value types. These may be either static or instance methods, but cannot be virtual methods. Instance methods bound to a value class have a **this** argument, which will always be a reference to the actual receiver value. This reference carries no type information, since value classes are not self-describing. In fact, so far as the *VES* is concerned, the **this** argument may even be **null**. As

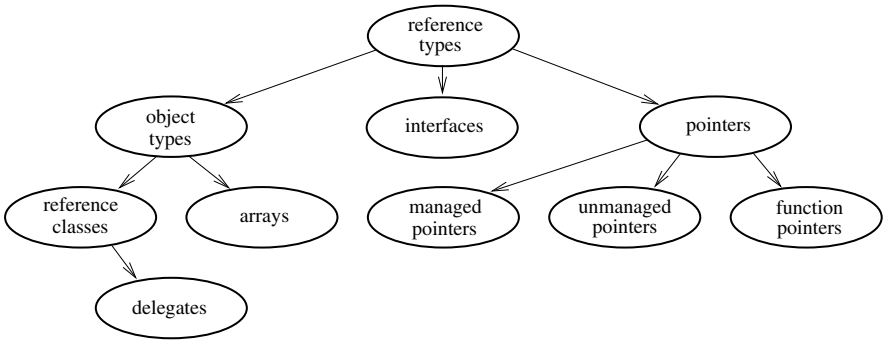


**Figure 3.4:** Hierarchy of reference types in the *CTS*

we shall see, this would be impossible for a virtual method, since the *VES* needs to access the *self-description* of the **this** value[1] in order to dispatch the right method.

Instance methods of value classes do not need to extract type information from their receiver, since an instance method is attached to an exact class. Within an instance method, the type with which the code is dealing is known to the compiler.

## Reference Types

Most of what is popularly thought of as object-oriented behavior results from the characteristics of the reference types, and more specifically the object types.

In the *CTS*, the reference types form a rich hierarchy, as shown in Figure 3.4. There are three subdivisions of the reference types. These are the *object types*, *interface types*, and *pointer types*.

Arrays and reference classes are object types. Object types are self-describing and are complete types. Completeness implies that knowing an object is of some particular exact object type completely specifies the features of the object. As we shall see in the next section, knowing that an object is of some interface type provides only a partial specification of the features of the object.

***Array object types.*** An array's *type* is determined entirely by its *element type*, and at runtime has an attribute that specifies the length. Line 17 of Figure 2.27 was an example of the use of the "`ldlen`" instruction to extract the array length from the array reference.

It is useful to be clear that in the *CTS* two arrays are the same type if they have the same element type. Furthermore, array types do not have names but take their names from

[1] This would read better for *OO* languages in which the receiver is called **self**. "... the *VES* needs to access the self-description of the **self** value ..."

the name of their element type. If we wish to refer to (any) "**array of** *String*" in *IL*, in a method signature for example, we will use the type denotation "**string** `[]`". In *IL*, as in *C#*, an array is specified by naming the element type and following the name by a pair of empty brackets, '[]'.

In *C#*, and in the *CTS*, arrays of the same element type are assignment compatible. This *structural compatibility* of array types is not the semantic model that some programming languages use. For example, in *Pascal*-family languages arrays are named, and two differently named arrays are distinct, even if they have the same element type and length. Thus —

```
type
    Foo = array 8 of CHAR;
    Bar = array 8 of CHAR;
```

are different types and are not assignment compatible. This behavior is called *name compatibility*. In effect, two types are compatible if they share the same (fully qualified) type name. It follows that languages that require name compatibility for arrays must rely on the compiler to enforce the restriction, since neither the *CTS* nor the verifier will do so.

Support for the creation and indexing of one-dimensional arrays is built into the *VES*. Much of the documentation refers to these one-dimensional arrays as *vectors*. The instruction "`newarr`" creates a new one-dimensional array. As we saw on page 27, the "`ldelem.*`" family of instructions loads array elements, and the corresponding "`stelem.*`" family assigns values to array elements.

Multidimensional arrays are not directly supported by the instruction set. There are methods in the *System.Array* class that allow for the creation of arrays of different dimensionality, with element counts that do not necessarily start at zero. The same class also defines methods for accessing and manipulating values of such array types. As at the *Beta* release of *.NET* these methods are not inlined and exact some performance penalty. An alternative is to use arrays of arrays instead. This is the required semantics for *Pascal*-family languages anyway. In *C#* terms this corresponds to using the second line below, rather than the first —

```
int[,]    foo = new int[8,4];
int[][]   bar = new int[8][];
for (int i = 0; i < 8; i++) bar[i] = new int[4];
```

Unfortunately, adopting the second form in *C#* requires providing an explicit initializer loop, as seen in the third line of the code fragment. Of course, in languages in which this is the way in which multidimensional arrays are declared, the compiler would emit constructor code without requiring any user intervention.

Depending on the way in which the multidimensional arrays have been implemented, the code for accessing elements is also different. Suppose we have the following code, accessing the two arrays —

```
foo[2,3]   = 17;      // true two-dim int array type
bar[2][3]  = 19;      // array of array of int type
```

The disassembly of this code, shown in Figure 3.5, demonstrates the special syntax that is used for access to the *System.Array* types. In the first implementation, lines 1–5, all the work is performed by a call of an instance method, *Set*, with a special signature. The receiver is the array reference, and the three arguments are the two array indices and the value to be assigned.

```
// foo[2,3]   = 17;              // true two-dim int array type
ldsfld   int32[0...,0...] Hello::foo                          // line 01
ldc.i4.2                                                      // line 02
ldc.i4.3                                                      // line 03
ldc.i4.s 17                                                   // line 04
call     instance void int32[0...,0...]::Set(int32,
                                             int32,
                                             int32)           // line 05
// bar[2][3] = 19;               // array of array of int type
ldsfld   int32[][] Hello::bar  // push static field bar        line 06
ldc.i4.2                       // push first index            line 07
ldelem.ref                     // get reference               line 08
ldc.i4.3                       // push second index           line 09
ldc.i4.s 19                    // push int32 value            line 10
stelem.i4                      // store the array element
```

**Figure 3.5:** Disassembly of multidimensional array accesses

The second implementation navigates the data structure, using the one-dimension array instructions of *IL*. Figure 3.6 shows the runtime layout. The variable *bar* is a reference to an array of eight references to separate four-long arrays of integer type. We begin by pushing the static field *Hello::bar* at line 6. This field has type "`int32[][]`". We push

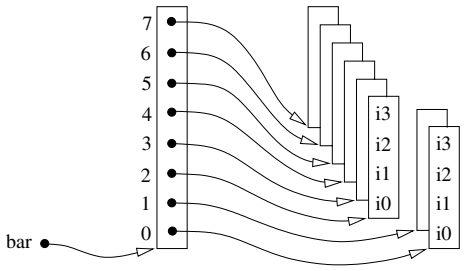


**Figure 3.6:** Runtime layout of array of arrays

the first array index at line 7 and load the element at line 8. It will be recalled that the "`ldelem`" instruction is specialized for the various array element types. In this case the array element is a reference so we use the instruction "`ldelem.ref`". To finish, we push the second index, then the element value, and finally perform the assignment using the "`stelem.i4`" instruction. Note that this instruction has a *stack*-$\Delta$ of $-3$.

If the "call" of *Set* at line 5 of this example, and corresponding "calls" of *Get* are inlined by the *JIT*, the first array access method should be faster than the second.[2] Furthermore, it may be noted that a true two-dimensional array requires a single object allocation, while an array of arrays requires many. In our example, the array of arrays structure requires nine separate objects to be allocated.

***Array covariance.*** Before leaving arrays, a word on array covariance. Suppose that $T_1$ is a subtype of some type $T$. In the *CTS* an array of the type "**array of** $T_1$" is considered to be assignment compatible with a location that has the declared type "**array of** $T$". This property is called *array covariance*.

There are two immediate consequences of this rule. First, it is necessary for the *VES* to type-check every array element assignment. Secondly, it places a small but significant hole in the type system.

Suppose that an "**array of** $T_1$" has been assigned to a location of type "**array of** $T$". Code that has access to the wider array—that is, the "**array of** $T$"—would normally expect to be able to assign any variable of type $T$ to any element position in the array. However, this is illegal, since there may be other, original references of type "**array of** $T_1$" that point to the same array. From the point of view of the original references the sudden appearance of an element of type $T$ in the array would violate their invariants. This danger is removed by the *VES* performing a type check on every array element assignment.

The overhead of a type check is relatively small. The consequences for the type system are arguably more far-reaching. Suppose it is your task as a software engineer to write a library, with a method that takes an argument of array type. For the sake of concreteness, let us assume that it is a method that adds a new element to the array. Under normal circumstances it would be reasonable to assume that it was possible to write such a method, and statically guarantee that it was free from type errors. Unfortunately this is not the case. If a user of the library quite legally passes an array of a subtype to your method, the method will fail with an array element assignment exception.

Of course, array covariance did not appear in the *CTS* by accident. This behavior is specified in some languages, *Java* for example. If the *CTS* did not allow array covariance it would be difficult to implement such languages efficiently. In any case some people believe there are circumstances in which array covariance leads to somewhat more convenient code. At core, these are religious matters, and as compiler writers it is our job to implement the language, not argue the religion.

[2] The converse is also true. If the calls are not inlined, the first method will be significantly slower.

***Class object types.*** Reference classes are the types from which the *CTS* gets its characteristic object-oriented flavor. Reference classes are self-describing and complete. Such classes declare that they inherit from exactly one supertype and may also implement zero or more interfaces. The built-in class *System.Object* is the sole exception to this rule, as it is the root of the class hierarchy and does not inherit from any other type.

Variables that are declared to be of some reference object type may be assigned a value of any assignment compatible type. References to objects of a subtype are assignment compatible with variables of the supertype. This form of assignment compatibility can be guaranteed at compile time in statically typed languages. Consider the following scenario. Two variables have been declared as follows —

```
public A a;                    // A is a class object type
public B b;                    // B is a class object type
  ...
a = b;                         // assign value in b to a
```

The compiler can check the relationship between the types *A* and *B*. If *A* is a supertype of *B*, then the assignment is known to be correct. The value in variable *b* need not be of exact type *B*, but if it is a subtype of *B* then it is necessarily also a subtype of *A*.

If *A* is a subtype of *B*, then the assignment is statically incorrect. It is possible that the assignment might be dynamically correct, if the exact type of *b* happens to be a sufficiently narrowed subtype of *B*. However, the assignment would then require a runtime type check, which would appear as an explicit narrowing cast in the source —

```
a = (A) b;                     // assign value in b to a
```

Finally, the types *A* and *B* might be unrelated. In that case the assignment is known to be incorrect statically, and incorrect dynamically also.

At this point it is worth noting that type-check casts of the kind in the last example are also able to be checked statically. When a value of static type *B* is cast to some other class object type *A*, we have three possibilities. The cast might actually be widening, in which case no check is needed. If the cast is narrowing, then the runtime check needs to be performed and might either succeed or fail. Finally, if the types *A* and *B* are unrelated, we may conclude that the runtime check will always fail, and a compile-time error will be signalled. As we shall see later, it is not possible to reach such strong conclusions in the case of type-check casts to interface types.

Here is the *IL* syntax for class definitions. This syntax applies to both value and reference classes. Value classes are recognised by the fact that they extend *System.ValueType*; all others are reference classes.

```
ClassDecl    →   .class ClassHeader "{" { MemberDecl } "}" .
ClassHeader  →   { ClassAttr } ident [ extends TypeRef ]
                     [ implements TypeRef { ',' TypeRef } ] .
MemberDecl   →   <Member Declaration> .
```

Classes may be defined with a rich repetoire of attributes to control visibility and other characteristics. Here we shall only mention the attributes that apply to top-level classes. The case of nested classes is somewhat more detailed.

The attributes listed in Figure 3.7 may be used, with the indicated semantics. The first group of attributes in the figure control visibility. Top-level classes may only have public or assembly visibility, which are denoted by the keywords **public** and **private** respectively. All members of the class take their *visibility* from this declaration. The modifiers that the members include in their definitions can only control the member's *accessibility*.

The next group of attributes control the semantics of inheritance, and instantiation. A class that is declared to be **interface** is not an object class at all but is an interface class. These are described in a later section.

A class that is declared to be **abstract** cannot have an instance created. If a class has one or more abstract methods, then it is necessarily an abstract class. However, abstract classes need not have any abstract methods, or indeed any methods at all. A **sealed** class cannot be extended further. All virtual methods of a sealed class will be final, although it is not clear that they need to be declared that way. It is possible for a class to be both **abstract** and **sealed**. Logically, such a class can neither be instantiated nor extended. Such a class

| *Attribute* | *Effect* |
|---|---|
| **private** | (Default) This class and its members will not be visible outside of this assembly. Cannot be used with **public**. |
| **public** | This class and its members will be visible outside of this assembly. Cannot be used with **private**. |
| **abstract** | This class cannot be instantiated. |
| **interface** | This class is an interface definition. |
| **sealed** | This class cannot be further extended. |
| **ansi** | This class uses *ANSI* marshalling for character strings. This is the default behavior. |
| **autochar** | This class uses platform-specific marshalling of character strings. |
| **unicode** | This class uses unicode marshalling for character strings. |
| **auto** | The layout of this class is determined automatically. |
| **explicit** | An explicit layout will be given for this class. |
| **sequential** | The class is laid out sequentially. |

**Figure 3.7:** Class declaration attributes

can only contain static features, and might be used to implement *static classes* for those languages that have such things.

The third group of attributes in the figure control the way in which character strings are marshalled across a managed-to-unmanaged call boundary. The **ansi** attribute is the default.

The final group of attributes control the layout of features in the memory image of instances of the class. The default layout is **auto**, which leaves the layout up to the runtime. There are circumstances in which objects are passed to and from unmanaged code. In such cases the layout expected by the *VES* must correspond to the native layout used by the unmanaged code. The other two layout attribute values allow control of this characteristic. Explicit layout is considered further in Chapter 4.

***Class members.*** Classes have *members*, sometimes also called *features*. In the *CTS* there are four kinds. These are *fields*, *methods*, *properties*, and *events*. Method declarations are dealt with in some detail in "Defining Methods" (page 93).

Fields may be either instance fields or static fields. Every object of a particular class type has its own copy of the instance fields. Static fields are shared, and may be thought of as residing inside a runtime class descriptor that is unique for each self-describing class. In the *IL* there are different instructions for accessing the values of static and instance fields. Static fields are declared with the **static** attribute. If **static** is not specified, the field will be an instance field.

As well as the **static** attribute that determines the basic kind of field, fields have other attributes that control accessibility, mutability, and some special name markers. These additional predefined attributes for fields are shown[3] in Figure 3.8. Accessibility was discussed on page 43. In the figure the first group of attributes control the accessibility of the field, as described. The **privatescope** accessibility was not discussed previously. This mode allows the compiler complete control over access, since the datum can only be accessed by using the field definition token. This feature supports static local variables of functions in language *C* or **own** variables in *ALGOL*. It may be thought of as a structured form of name mangling.

Fields that are marked **literal** do not occupy space in the object but appear in the metadata. Only static fields may be literal, and they must have an explicit value in their definition. Fields that are marked **initonly** are used for fields that are given a value at load time, and are never changed thereafter. **Initonly** fields may only be given a value by the code of an initializer. However, an attempt to change an **initonly** field is a verification error rather than a runtime error. This means that marking a field as **initonly** is only effective in verified contexts.

The final group of attributes mark names as having special significance to various tools, or to the runtime system. For example, the names of the access methods for properties are marked with the **specialname** attribute.

[3] Attributes that control serialization and marshalling have been omitted from this discussion.

| *Attribute* | *Effect* |
|---|---|
| **assembly** | This field is only accessible within this assembly. |
| **family** | This field is only accessible in this class and in subclasses of this class. |
| **famandassem** | This field is accessible in this class and its subclasses but only within this assembly. |
| **famorassem** | This field is accessible in this class, its subclasses, and throughout this assembly. |
| **private** | This field is only accessible within this class. |
| **privatescope** | This field is only accessible within the assembly, and then only by using the field definition token. |
| **public** | This field is publicly accessible. |
| **literal** | This static field is a literal value. |
| **initonly** | This field may only be assigned in constructors. |
| **specialname** | The name has special significance to tools. |
| **rtspecialname** | The name has significance to the runtime. |

**Figure 3.8:** Field declaration attributes

Properties provide an additional mechanism, other than fields, for associating named data values with classes or with object instances. In languages such as *C#* properties are declared as part of classes. Properties have names, and have associated *getter* and *setter* methods. In *C#* properties are accessed by name, using the usual field-access syntax. However, since properties may be defined to have only a *setter* method or only a *getter* method, properties may be write-only or read-only. If both methods are defined, the property allows both reading and writing. Every used occurrence of a property name in a source program translates into a call to the associated accessor method in the *IL*. From a theoretical point of view, properties provide a mechanism for enforcing invariants on values. Since all accesses to the property value must pass through the access methods, the bodies of the methods are the place to enforce assertions. For example, if some checking code is put inside a *setter* method, it is guaranteed that no code may change the value without passing the check. The same argument applies to code-instrumentation. If instrumentation is placed in the property access methods, it is guaranteed that *all* accesses to the value will be intercepted.

The access methods may be static, instance, or virtual methods. They may work by simply getting or setting the value of an associated, private *backing field* or may compute their values on the fly. Of course, the backing field must be **private**; otherwise we could not be sure that the value was not accessed explicitly, thereby evading the discipline of the accessor methods. The case of virtual methods for property access is particularly important, since method overriding may then be used to add additional checks to existing property access code.

At runtime the *VES* neither knows nor cares about properties. All accesses are encoded as conventional calls to the *getter* and *setter* methods. Thus languages that do not

have special syntax for properties can still access the values. However, properties have a special status in the metadata and thus have special syntax in *CIL*. Figure 3.9 is typical code of a property. In this case the property is read-only, but has an additional method for incrementing the property value. In practice the private *count* field would be initialized by the constructor, which is not shown in the figure. In a cross-language environment, programs that do not understand properties will ignore the metadata and call the two methods directly.

```
.class public Counter
  .method virtual instance specialname public int32 get_Count()
    {                              // the getter method
       ldarg.0                     // push this value
       ldfld int32 'count'         // fetch the private field
       ret                         // return the result
    }

  .method virtual instance public void Increment() {
       ...                         // increment the count field
    }

  .property int32 Count() {        // the property declaration
       .backing int32 count()
       .get instance int32 get_Count()
       .other instance void Increment()
    }
} // end of public class Counter
```

**Figure 3.9:** *IL* example of property definition

Events are the last of the four kinds of member that a class may contain. Events are a special kind of object reference that are distinguished in the metatdata because of their role in the *event-handling model* of the virtual object system. Events are references to a particular specialization of delegate types, which are the subject of the next section.

***Delegate types.*** Delegates may be thought of as a type-safe mechanism for implementing the "*function pointers*" of language *C*. In *Pascal*-family languages, the equivalent construct is the *procedure variable*. In contrast to the direct implementation of function pointers described earlier, delegates are type-safe and verifiable. Unlike traditional function pointers in most languages, delegates may be used for both static and dispatched methods.

Delegate types, as may be seen from their position in the hierarchy of Figure 3.4, are implemented as a kind of reference class. Each instance of a delegate encapsulates a datum that denotes the method that has been assigned to the instance. In the case of instance methods it also encapsulates an object reference that will be the **this** of any invocation. If

the delegate instance has been assigned a static method, the encapsulated object reference will be **null**.

The key to the verifiability of delegates derives from the fact that delegates are entirely opaque structures. There are no operations on delegates except for a constructor and methods to invoke the encapsulated function. Furthermore, the code of even these few methods is supplied by the runtime, rather than by the compiler. The runtime is able to guarantee that the encapsulated function has the correct signature, because it checks this itself within the hidden constructor code.

As a first example, the declaration of a delegate type to encapsulate a function that takes two parameters of type "`double`" and returns "`double`" may be achieved by the *IL* in Figure 3.10. The signature of the constructor is always the same. The first argument is of some object type, while the second is of type **unsigned native int**. On current 32-bit machines the function pointer value will be an **int32**, but future compatability and the verifier require the native type here. The *Invoke* method is the method that is called to activate the encapsulated function. The signature of this method must match the signatures of the functions that will be assigned to the delegate.

```
.class public auto sealed DoubleToDouble
                    extends [mscorlib]System.MulticastDelegate {
    .method public specialname rtspecialname instance
            void .ctor(object, unsigned native int) runtime managed {}
    .method public virtual instance
                    float64 Invoke(float64, float64) runtime managed {}
} // end of class DoubleToDouble
```

**Figure 3.10:** A simple delegate declaration

Notice that all delegate types must be declared to be **sealed**, and they signal their special semantics by extending the class *System.MulticastDelegate*.[4] The fact that the implementations of the two methods are supplied by the runtime, rather than by explicit *IL*, is shown by the empty bodies of the methods and the *runtime-managed* denotation. The only additional methods that a delegate may possess are used for asynchronous calls. In such cases the two additonal methods are called *BeginInvoke* and *EndInvoke*.

When the constructor is called it is passed two data. First on the stack is the receiver object for the delegated calls, and a function pointer is on the top. For a static procedure, the first datum is **null**. Suppose, for example, that we wish to create an instance of the delegate type declared in Figure 3.10. We wish to encapsulate the function *Math.Power*.

[4] The name of the class that is extended is a historical relic. Originally the *Delegate* class was used for delegates, and an extension, the *MulticastDelegate* class, was used for events. In the final version the extended class must be used as the base class for both (ordinary) delegates and the *event* types that we discuss next.

The *C#* source code might have gone —

```
DoubleToDouble x = new DoubleToDouble(Math.Power);
```

Figure 3.11 shows how this would be translated into *IL*, in the case that the variable *x* is a static field.

```
...
ldnull                          // no encapsulated receiver for static method
ldftn   float64 Math::'Power'(float64, float64)
newobj  instance void ThisMod.DoubleToDouble::.ctor
                              (object, unsigned native int)
stsfld  class ThisMod.DoubleToDouble ThisMod::'x'
...
```

**Figure 3.11:** Constructing a static delegate value

If the delegate is to be bound to a particular object, then that object is specified at the time of delegate construction, and the encapsulated method will be an instance or virtual method of the object's type. In *C#* the syntax for such an assignment is similar to the static case. The compiler recognizes the instance method and generates the alternative code.

As an example, suppose we wish to use a delegate to attach a no-args method *Count* to an object *TargetObj*. Presumably whenever the delegate is invoked, we wish for some field of the target object to be incremented. The source syntax might be —

```
NoArgDelegate x = new NoArgDelegate(targetObj.Count);
```

where it is assumed that the appropriate delegate type is named *NoArgDelegate*. Figure 3.12 shows typical resulting *IL*. Once again we have assumed that the delegate reference is held in a static field of the class *ThisMod*.

```
...
<push reference to targetObj>                    // receiver for invocation
ldftn   instance void Target::'Count'()
newobj  instance void ThisMod.NoArgDelegate::.ctor
                              (object, unsigned native int)
stsfld  class ThisMod.NoArgDelegate ThisMod::'x'
...
```

**Figure 3.12:** Constructing a delegate value with an instance method

In this example, if the method *Count* had been a virtual method of the type of the target object, then the produced code would have needed to be slightly different. In that

case, the *v-table* of the object is accessed at the time that the delegate is constructed. Figure 3.13 shows the variant code. Notice the "`dup`" instruction. We need to duplicate the object reference, since one copy is passed to the delegate constructor and one copy is used up by the "`ldvirtftn`" instruction.

```
...
<push reference to targetObj>                    // receiver for invocation
dup
ldvirtftn   instance void Target::'Count'()
newobj  instance void ThisMod.NoArgDelegate::.ctor
                          (object, /unsigned native int/)
stsfld  class ThisMod.NoArgDelegate ThisMod::'x'
...
```

**Figure 3.13:** Constructing a delegate value with a virtual method

We have now seen how delegates are declared and instantiated. It remains only to see how they are invoked. In most languages a call to a function pointer or procedure variable has the same syntax as a function call, but with the variable designator expression replacing the name of the function. In *IL*, in order to invoke the encapsulated function, we need to make a virtual call to the *Invoke* function of the delegate. The pattern is —

```
<push reference to delegate object>
<push arguments to invocation>
callvirt instance retType DelegateClass::Invoke( <args>)
```

where *retType* is the return type of the encapsulated functions. For our second example, the instance of a *NoArgDelegate* held in a static field *x*, the *IL* would just be —

```
ldsfld  class ThisMod.NoArgDelegate ThisMod::'x'
callvirt instance void ThisMod.NoArgDelegate::Invoke()
```

Notice that the reference that we push is a reference to the delegate. There is no mention of the object receiving the encapsulated *Count* method. The receiver object is frozen inside the delegate at the time of instantiation.

***Event types.*** Event types are delegates with some additional semantics. The declaration of an event type in *IL* is identical to the declaration of an ordinary delegate type. Both extend *System.MulticastDelegate*. In the case of ordinary delegates some of the inherited functionality is not used. As before the internal structure is opaque.

The essential semantic difference between delegates and multicast delegates is that multicast delegates use the built-in support for *lists* of delegates. When the *Invoke* method of an event is called, *all* of the encapsulated methods on the list are invoked. The base class of delegates has methods for combining multicast delegates together, and for removing delegates from the list. When the *Invoke* procedure of a multicast delegate is activated, the delegates on the list are called in the order that they were linked. If the encapsulated methods are value-returning functions, the call of *Invoke* returns the return value of the last delegate to be called. The return values of all but the final delegate are discarded.

Under normal circumstances events are implemented by a *backing field* of the appropriate type. This field will be declared **private** to preserve its integrity, and will only be acessible via method calls in the *IL*. The reasoning is the same as for the private backing fields used for property implementation.

Compilers for languages that support this event-handling model define methods to register and deregister delegates on their multicast variable. These methods wrap the underlying methods of *System.MulticastDelegate* so as to expose simple *add* and *remove* procedures specific to each event member of a class. The wrappers take a single argument that is the new delegate to add or remove, and are hard-coded to link to their specific event backing field. Suppose that a class has a public member named "`reaction`" of some event type *BlahHandler*. The add and remove methods will be generated by the compiler, in addition to any methods of the class that are declared by the user. The skeleton code is shown in Figure 3.14. The *add_reaction* method adds a new delegate to the multicast delegate currently held in the private field *reaction*. It does so by calling the static *Combine* method of the *System.Delegate* class. The *remove_reaction* method is almost precisely the same, except that it calls the static *Remove* method of *System.Delegate*. It should be noted that these two methods are declared in *System.Delegate*, even though they may only be called on objects of classes that extend *System.MulticastDelegate*. It is possible to call *Combine* directly; indeed it is necessary to do so if linking to an event datum that is a local variable.[5] The public nature of the event in this example is embodied in the **public** declaration for the add and remove methods, while the backing field remains private. The two methods are declared to be **synchronized** since the linking and unlinking must be performed atomically. Remember that the common context in which event handling is required is one of multithreading.

The final few lines in Figure 3.14 show how information about the event is passed into the metainformation through `ilasm`. These lines declare the semantics of the add and remove methods for the metadata.

***Interface types.*** Interfaces are *fully abstract types*. Interfaces declare abstract methods and may possibly define static methods and fields. However, they cannot have instance fields or nonabstract instance methods. Classes may declare that they implement particular interfaces. If they do so, they enter into a contract to supply methods to implement

[5] Why? Well for languages without nested procedures there is no way that an "`add_*`" wrapper method could manipulate the local variable of another procedure.

```
.field private class BlahHandler 'reaction'  // the backing field
...                                          // any user methods
.method public specialname instance void add_reaction(
            class BlahHandler) il managed synchronized {
    ldarg.0                                  // destination ref
    ldarg.0
    ldfld    class BlahHandler ThisClass::'reaction'
    ldarg.1                                  // delegate to add
    call     class [mscorlib]System.Delegate
             [mscorlib]System.Delegate::Combine(
                         [mscorlib]System.Delegate,
                         [mscorlib]System.Delegate)
    castclass BlahHandler                    // cast to dest. type
    stfld    class BlahHandler ThisClass::'reaction'
    ret
}
...                                  // remove_reaction is similar
.event BlahHandler reaction {
    .addon instance void ThisClass::
                          add_reaction(class BlahHandler)
    .removeon instance void ThisClass::
                        remove_reaction(class BlahHandler)
}
```

**Figure 3.14:** Wrapping the linking and unlinking methods

all of the nonstatic methods declared in the interface. Abstract classes that implement interfaces may leave methods **abstract**, but concrete classes must define or inherit concrete implementations for all of the methods in the interface.

Variables that are declared to be of some interface type may be assigned values of any type that implements the interface. Unlike the case with class object types, it is not always possible to be able to guarantee correctness of such assigments at compile time. Consider the following scenario. Two variables have been declared as follows —

```
public A a;                    // A is an interface type
public B b;                    // B is a class object type
  ...
a = (A) b;                     // assign value in b to a
```

In the corresponding example with class types, on page 62, the compiler was able to statically check the relationship between the types *A* and *B*. In this case things are not so simple. If class *B* is statically known to implement interface *A*, then the assignment is known to be correct. This follows since, even if the exact type of *b* is a subtype of *B*, it will inherit the

obligation to implement *A*. The problem occurs if *A* and *B* are apparently unrelated. In the class object case we could reject the code as certain to fail. However, in the case of interfaces, if class *B* is not sealed, then the exact type of *b* might be some subtype of *B*, validly implementing the interface *A*. Casts to interface types are thus seldom able to be rejected at compile time and mostly remain as runtime type checks in the *IL*.

Interfaces are not complete types in the sense defined earlier. If we know the exact type of an object, then we know about all of the accessible features, including fields and methods. On the other hand if we only know that an object implements a particular interface, then we have only partial knowledge. We may be sure that the object has a set of methods that may be invoked, but we do not have any static guarantee about any other features. In the *.NET* system we may find out about the other features using *runtime introspection* with the reflection *API*, but this does not help at compile time.

Interfaces are defined in *IL* with the same syntax as classes. However, the class definition must contain the **interface** attribute. Interfaces cannot declare that they inherit from any class but may declare that they implement any number of other interfaces. Of course, interfaces do not actually implement anything, since they cannot define instance methods. The effect of an interface *I* declaring that it implements a particular interface *J* is to ensure that any class implementing *I* transitively inherits the obligation to implement *J* as well.

An interface may only define fields that are static. Methods in an interface definition must be either **static**[6] or be **abstract public**. Figure 3.15 is a very simple example of an interface, expressed in *CIL*. In this case there is a single method. Presumably classes that implement this method will pass on the strings to some kind of voice synthesizer, so that the program can chatter on to the user. In practice such an interface would probably need some kind of class constructor to initialize the synthesizer engine, and so on.

```
.class public interface abstract Talkative {
    .method public virtual abstract void speak(string)
     {}
} // end of interface Talkative
```

**Figure 3.15:** A very simple interface example

Classes that contract to implement an interface have a number of ways to supply the required methods. Suppose that a class *A* declares that it implements an interface *I*, which contains a method *Foo*. Class *A* may —

- provide a so-called "*MethodImpl*" that declares that *Foo* is implemented by some named public method with a matching signature

[6] In fact if the class is to be *CLS* compliant, there can be no static methods other than the class constructor "`.cctor`".

- provide the definition of a public method named *Foo* with the required signature
- have a parent that implements *I* and defines a public method *Foo* with the right signature
- have a parent that that does not implement *I* but defines a public method *Foo* with the right signature anyway
- leave the slot empty if *A* is declared **abstract**

These rules are applied in order, in case several apply. If no rules apply, then it is a load-time exception.

It may be noted that the first rule gives a way of avoiding ambiguity if a class implements two interfaces and each defines an equally named method. In this case two methods may be defined with different names to implement the separate semantics of the two contracts. Separate *MethodImpls* are then used in the *IL* to associate each method with its contractual obligation.

***Managed and unmanaged pointers.*** So far we have discussed only references to objects. Such references are totally *opaque*. The only operations on such values are assignment to type-compatible locations, and their use to gain access as a handle to the referenced object. As well as object references, there are other some kinds of references. The other kinds of references are the *pointers*.

We have already discussed function pointers briefly in the previous chapter, on page 40. The three other kinds of pointers are —

- **managed pointers**. These are created by taking the address of an object field or a managed array element. They may point to the address one beyond the end of a managed array.
- **transient pointers**. These are created by taking the address of a datum, including local variables and parameters. Transient pointers can only exist on the evaluation stack. A location cannot be declared to be of this type.
- **unmanaged pointers**. These are the traditional pointers of languages such as *C*. They may be used to hold arbitrary addresses, but such use usually results in unverifiable code and may threaten memory safety. Pointer arithmetic is permitted on such values.

Local variables and parameters may be declared to be of a managed pointer type. In *IL* such a declaration is of the form *TypeName &*. Fields of objects, array elements, and static fields can never be declared to be of managed pointer type.

Unmanaged pointers may be declared anywhere that an integer may be declared. In fact, so far as the *CLR* is concerned, unmanaged pointers *are* just unsigned integers of an

appropriate size. Nevertheless, it is good practice to declare such data as being bound to a particular type by using the *IL* declaration form *TypeName* `*`. Unmanaged pointers should never be allowed to point into the garbage-collected heap, as such use may compromise memory safety.

Perhaps the key pointer kind is the transient pointer, since all pointers that have values derived by use of the "`ld*a`" (load address) instructions start their lives as transient pointers. When a transient pointer is assigned to a managed pointer location, the value becomes a managed pointer. If it is assigned to an unmanaged pointer location, then the value becomes an unmanaged pointer. In particular, when a transient pointer on the stack is passed as an actual parameter to a method that expects a reference parameter, the value becomes a managed pointer.

Managed pointers may point either to the garbage-collected heap or elsewhere in memory. For example, the address of a class object field will clearly be in the heap, while the address of a local variable will be in the activation record of the current method. Managed pointers are reported to the garbage collector, so that the collector may find all references to the heap. The collector will be able to discover which pointer values are in the heap and which point elsewhere in memory.

Managed and unmanaged pointers may be combined with integers in certain ways. Integers may be either added or subtracted from pointers, returning a pointer of the same kind but almost certainly scoring a "fail" from the verifier. Pointers may be subtracted from each other, resulting in an integer. Here is an experiment to try. Compile the following legal but boring *Component Pascal* program with the "`/nocode`" flag. This will result in an *IL* file, but no executable.

```
MODULE Hack;
  IMPORT CPmain, Console;
  VAR arr = ARRAY 4 OF INTEGER;
BEGIN
  Console.WriteInt(arr[0], 1); Console.WriteLn;
END Hack.
```

Inside the *IL* file, the code to push the arguments of the call of *Console.WriteInt* will look like this —

```
ldsfld int32[] Hack.Hack::'arr'
ldc.i4.0
ldelem.i4                  // push arr[0]
ldc.i4.1                   // push 1, then call...
```

Now hack on the "`Hack.il`" file with your favorite editor, replacing the loading of the array element in the first three lines of the fragment with the following *IL* —

```
ldsfld int32[] Hack.Hack::'arr'
ldc.i4.1
ldelema int32           // address of arr[1]
ldsfld int32[] Hack.Hack::'arr'
ldc.i4.0
ldelema int32           // address of arr[0]
sub                     // subtract the pointers
```

Now assemble this file with `ilasm`, and run it. The program writes out 4, showing that the default integer type takes up four bytes. Maybe on future 64-bit machines this program will say 8 instead. If the stand-alone verifier is now run over the executable using "`peverify /il Hack.exe`", then the verifier will tell you exactly what it mistrusts about this program, and why.[7]

# The Object Instruction Set

In the last chapter, we dealt with the base instructions of *IL*. In this section we shall look at the remainder of the instructions. These instructions implement the object model of the *CTS*. Since the base instruction set is complete, in principle all of the instructions of the object instruction set could be synthesized from sequences of instructions in the base set. However, keeping the instructions separate provides two benefits. First of all, the user does not need to know the details of object layout, which are abstracted away in the *CTS* model. Secondly, because the object instructions carry symbolic information with them, the task of verification of correct usage is made feasible.

## Loading and Storing Data

We have already seen most of the instructions for manipulating fields and array elements. Figures 2.6 to 2.9 include all of these instructions. However, we collect them together here in Figure 3.16 for convenience. We have instructions to load and store instance fields, static fields, and array elements. In the case of instance fields a reference to the object is on the top of the stack, and the class that defines the field is specified in the instruction argument. The exact type[8] of the object must always be the specified class or an extension of the class. In the code generation chapters we refer to the reference to the object as the *object handle*.

It is an important detail that the class that is specified must be the class that *defines* the field. If the field is inherited from some supertype, then the class name of the supertype

[7] This process of writing a type-safe program and then modifying the *IL* is a useful experimental technique. The compiler produces all of the "boilerplate" code required, allowing short experimental code sequences to be substituted with minimal effort.

[8] The term *exact type* is used in much of the *CTS* documentation to mean the type of the precise class of which the object is an instance. This is to distinguish the cases in which a reference to the *type* of an object would mean "that type or any subtype of that type."

| *Data kind* | *Opcode* | *Comment* |
|---|---|---|
| instance field | `ldfld` | Op-arg: *Type Class::field-name* |
| static field | `ldsfld` | Op-arg: *Type Class::field-name* |
| array element | `ldelem.*` | Suffix: specialized for element type |
| instance field | `stfld` | Op-arg: *Type Class::field-name* |
| static field | `stsfld` | Op-arg: *Type Class::field-name* |
| array element | `stelem.*` | Suffix: specialized for element type |

**Figure 3.16:** Load and store instructions in object set

must qualify the reference. This is different to the *JVM*, which is happy to accept any qualifying class in which the particular field is visible.

In the case of static fields there is no reference on the stack, and the class that defines the field is specified in the instruction argument. Thus "`ldfld`" has a *stack*-$\Delta$ of 0, while "`ldsfld`" has a *stack*-$\Delta$ of 1. Static fields have an empty *object handle*, in our code generation jargon.

Array element loads and stores expect a reference to the array and an array index to be on the stack. In the code generation chapters we refer to this pair of values as the *array object handle*. In the case of element store instructions, the value to be stored is on top of the handle. Figure 3.17 represents the stack transitions for an array store.

Corresponding to the load instructions in Figure 3.16 we have instructions that load addresses of fields and elements. The value pushed on the stack is a transient pointer. These pointers may be dereferenced either for loading or storing with the load indirect, "`ldind.*`", and store indirect, "`stind.*`", instructions. Of course these instructions are also used to dereference pointers passed as reference parameters. All of these instructions are shown in Figure 3.18.

The load and store instructions that have a type suffix are "`ldelem.*`", "`stelem.*`", "`ldind.*`", and "`stind.*`". Usually these suffixes are the two-character type tags such as "`i4`". However, all of these instructions also have a form "`*.ref`", used for accessing references.

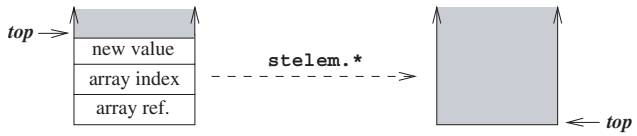


**Figure 3.17:** Storing an array element

| *Data kind* | *Opcode* | *Comment* |
|---|---|---|
| static field | `ldsflda` | Op-arg: *Type Class::field-name* |
| instance field | `ldflda` | Op-arg: *Type Class::field-name* |
| array element | `ldelema` | Op-arg: *Array element typename* |
| pointer target | `ldind.*` | Suffix: specialized for target type |
| pointer target | `stind.*` | Suffix: specialized for target type |

**Figure 3.18:** Load address, load and store indirect instructions in object set

## Loading Type Descriptors

At runtime, types are described by instances of the *System.Type* class. Obtaining access to these objects is the mechanism on which all *program introspection* depends. Once we have the *System.Type* object, we are able to call the methods of the *System.Reflection* classes to ask about the type. The loading of *System.Type* objects onto the stack is therefore the key primitive operation.

Two contexts arise. We may know the name of the class and wish to push the corresponding *System.Type* object on the stack. This corresponds to the expression evaluation on the right-hand side of the *C#* assignment —

```
System.Type t = typeof(<type name>);
```

Alternatively, we may have an object reference, and wish to perform introspection on the dynamic type of the object. This case corresponds to the expression evaluation on the right-hand side of the *C#* assignment —

```
System.Type t = obj.GetType();
```

If we know the name of the type, we use the "`ldtoken`" instruction to fetch the corresponding *runtime type handle*. From this handle we may get the *System.Type* object by a call to the *GetTypeFromHandle* method of the *System.Type* class. The *IL* is shown at the top of Figure 3.19. If we have an object reference, then we must call the *GetType* method inherited from *System.Object*. The *IL* is shown at the bottom of Figure 3.19.

## Using Types to Direct Control

For many people, the essence of object-oriented programming is that behavior may be specialized according to the *class* of an object. In principle this may be done two ways. We may have denotations for method calls for which the actual method that is invoked depends on the exact type of the *receiver object*. Alternatively, we may have type-test predicates that select the flow of control. In *.NET*, both mechanisms are supported.

```
// Get Type object for statically known type
    ldtoken <type-name>                          // get runtime type handle
    call class System.Type
        System.Type::GetTypeFromHandle(   // get System.Type object
               value class System.RuntimeTypeHandle)
    ...

// Get Type object for object on stack
    <push object reference>
    call instance class System.Type
        System.Object::GetType()             // get System.Type object
    ...
```

**Figure 3.19:** Pushing a *System.Type* object on the stack

***Type tests.*** There are two type-test instructions in *CIL*. The first of these is "`castclass`". This instruction is used for *type assertions*. A typical usage of this instruction looks like this —

```
<push object reference on the stack >
castclass    class [asm]MyClass       // assert type of object
...                                   // top of stack is MyClass
```

The type definition that appears in the instruction may be either the name of a class or the name of an interface type. If the cast fails, an *InvalidCastException* exception is thrown. If the cast succeeds, the value left on the stack is guaranteed either to be of the designated class (or one of its subclasses) or to implement the designated interface, as the case may be. The verifier understands the semantics of this instruction and will treat the top of stack value as the designated type downstream of the assertion. The *stack*-$\Delta$ for this instruction is 0.

Of course, it is often known statically that the cast will succeed. Typically an explicit cast is placed in the source code so that the compiler will allow selection of members of the asserted type or assignment to a location of a narrower type. The corresponding "`castclass`" instruction in the *IL* will keep the verifier happy, and also check that the programmer was telling the truth!

In should be noted that it is valid to perform a type assertion on a **null** value. In such cases the assertion always succeeds, leaving a **null** value on the top of the stack.

The other type-test instruction is the instance test "`isinst`". This instruction is similar to "`castclass`", to the extent that it takes an object reference on the top of stack and returns an object reference of the designated type. However, in this case, if the value cannot be cast to the designated type the instruction pushes **null** on the stack, rather than throwing

an exception. Once again the verifier understands the semantics, so the returned value is treated as being of the designated type downstream of the test.

It may be noted that this test is rather different from all of the other predicates that *IL* supplies. The others all return a Boolean value—that is, a 0 or 1 of type *int32*. This instruction returns either a **null**, for **false**, or a non-**null** value of known type, for **true**. This does not cause a problem, since the "`brfalse`" and "`brtrue`" instructions branch on **null** and non-**null** as well as on 0 and 1.

This instruction may be used to implement type tests in a fairly obvious way. A more interesting use is for the *regional type guard* of *Component Pascal*. In this language, the form —

```
with ident = Type1 do . . .          (* ident is known to be of Type1 here *)
| ident = Type2 do . . .             (* ident is known to be of Type2 here *)
else . . .
end
```

uses the type of the variable to select the branch of the statement to execute. It performs selection in the same way that the corresponding ordinary **if** statement would do —

```
if ident is Type1 then . . .
elsif ident is Type2 then . . .
else . . .
end
```

However, there is one important difference. Within each branch of the **with**, the *guarded region* as it is called, the selected identifier is known to be of the designated type. Thus no further casting of the identifier is required within each guarded region. This is why the statement is called a *regional type guard*.

The "`isinst`" instruction allows an elegant implementation of this construct. Figure 3.20 gives the complete *IL* for the code fragment shown above. In this figure, the source code has been interspersed as comments, to mark the source position in the *IL*. Additional local variables have been declared, one for each of the type guards.

The implementation of the statement begins by pushing the selected identifier. The "`isinst`" test is applied to this value, and the result duplicated. One copy of the result is saved into the corresponding temporary, and the other copy is tested to see if it is **null**. If it is, control branches to the next test; otherwise the code of the guarded region is executed. Within each guarded region applied occurrences of the selected identifier are replaced by accesses to the appropriate temporary variable.

In theory, if the selected identifier was a local variable of a procedure, it would not be necessary to define the typed temporary variables. This is because the verifier is able to track the types of local variables within control flow such as this. The verifier does not or

```
.locals(class Type1 t1, class Type2 t2)   // declare some temporaries
      ...
//   with ident = Type1 do
       <push ident on stack>
       isinst class [asm]Type1              // do first type test
       dup                                  // duplicate the result
       stloc.0                              // save one copy in t1
       brfalse  lb01                        // branch on null
       ...                  // first guarded region, use t1 for ident here
       br       lb03                        // branch to end
//   | ident = Type2 do
lb01: <push ident on stack>
       isinst class [asm]Type2              // do second type test
       dup                                  // duplicate the result
       stloc.1                              // save one copy in t2
       brfalse  lb02                        // branch on null
       ...                  // second guarded region, use t2 for ident here
       br       lb03                        // branch to end
//   else
lb02  ...                   // else part code
//   end
lb03: ...
```

**Figure 3.20:** *IL* code for **with** statement fragment

cannot track the types of other possible identifier kinds, such as arguments and static fields, so for generality the construct must be implemented with a temporary variable as shown.

***Virtual methods.*** Virtual methods provide a way of attaching different behavior to each type in a type hierarchy. As noted earlier, instances of object types are self-describing. You may think of the self-description as being implemented by a hidden field of every object that points to a shared *runtime type descriptor*. Every object of the same type will reference the same descriptor and will contain all of the information that is the same for all instances. Among other things the runtime type descriptor must have the hook that allows runtime introspection on the type. It must also contain any information that the garbage collector needs to safely collect objects of the type.

However, from the point of view of mediating behavior, the most important information in the runtime type descriptor is the *virtual method dispatch table*, or *v-table* for short. The *v-table* consists of an indexed array of *slots*, each of which contains a method pointer. Each slot of the *v-table* is associated with a particular method name and signature. In order to invoke a virtual method, the method name and signature are resolved (possibly at compile time) to a *v-table* slot index. At runtime the *VES* takes the reference to the **this** and follows the hidden reference to the *v-table*. The *VES* pulls out the method pointer in the chosen slot of the *v-table* and invokes that method.

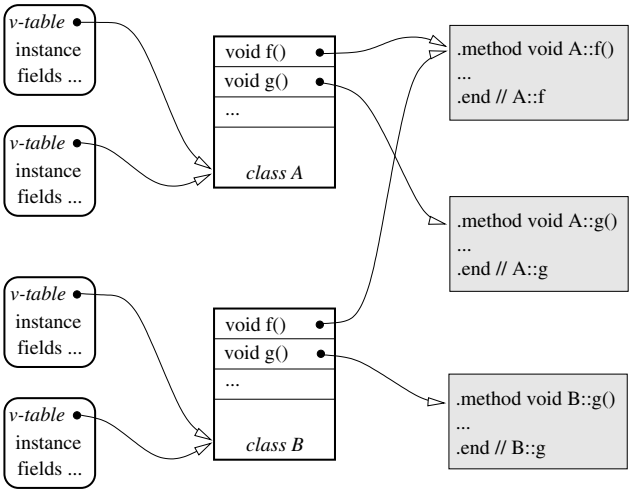


**Figure 3.21:** Objects with methods `f()` and `g()`

Consider Figure 3.21, which shows four objects, two each of two related types, *A* and *B*. Each object contains the instance data for that particular object and a reference, called *v-table* in the figure, that references the runtime type descriptor for that particular object. Each runtime descriptor has a *v-table* with just two methods, called *f* and *g*, shown in the figure. The *v-table* for class *A* selects `A::f()` and `A::g()` to implement the two behaviors. For class *B*, for behavior *f*, the *v-table* selects the same method `A::f()`, as did class *A*. For behavior *g*, the *v-table* of class *B* selects its own method, `B::g()`. Presumably *B* is a subtype of *A* and has inherited the *f* behavior unchanged, but has overidden the *g* behavior.

***Calling virtual methods.*** Virtual methods are declared *virtual* in the *IL* and must obey a number of other rules that are discussed later. They are invoked by first pushing the receiver onto the evaluation stack. The actual parameters, if there are any, are pushed next in order. The method is then invoked, using the "`callvirt`" instruction.

In the case of a virtual call the *stack*-Δ is "*number of return values – number of ingoing values*". The ingoing number is the number of arguments plus one extra to account for the receiver "argument". As noted earlier, inside the method the **this** will always be arg-0.

The operation argument to "`callvirt`" has the same general structure as a call to a static or (nonvirtual) instance method. Typical examples of the three cases are —

```
call void [RTS]Console::'WriteInt'(int32,int32)                  // static call, stack-Δ = –2
call instance float64 [LitValue]LitValue.value::'real'()         // instance call, stack-Δ = 0
callvirt instance int32 [Symbols]Symbols.Idnt::'parMode'()       // virtual call, stack-Δ = 0
```

In the case of an instance call, the name of the method in the instruction argument is exactly the name of the method that will be invoked. In the case of a virtual call the name in the instruction argument is the name of the method qualified by the static type of the reference. Of course, the exact type of the reference that will become the receiver may be a subtype of the static type. In this case, the actual method that is invoked will be whatever method is in the corresponding slot of the *v-table* of the exact type.

The "`callvirt`" instruction is also used for invoking interface methods. In such cases the name of the method in the instruction argument will be qualified by the name of the interface "class."

Virtual methods in *.NET* do not have to be invoked with "`callvirt`". It is legal to call such a method using the static call instruction, "`call`". If this is done, there is no dispatch via the *v-table*. Instead, the exact named method will be invoked. This particular ability is useful under at least two circumstances. First, if the compiler is able to statically know the exact type of a reference, then the use of the static call instruction will save time, since it is more efficient in most implementations. Secondly, if a method needs to call the virtual method that it overrides—that is, a method wants to make a **super** call, then it must use the static call instruction.

## Creating Objects

There are two instructions that are used for creating new objects. These are "`newarr`", which creates a new one-dimensional array, and "`newobj`", which does everything else.

One-dimensional arrays are created by the "`newarr`" instruction. This instruction takes a type as instruction argument and expects a length value, of unsigned type, on the top of the stack. The instruction returns a reference to the new array on the top of the stack. The array will index from zero and will be initialized to zeros of the appropriate type.

The type appearing as the argument to the instruction may be a built-in type, such as *int32* or *wchar*, or it may be any other type, including value types. Figure 3.22 is the *IL* to create the array of arrays structure of Figure 3.6. The code begins by allocating the reference array of eight elements. The instruction argument to the call of "`newarr`" is **array of int**, so the array that is created has type **array of array of int**. All of the elements of this one-dimensional array will be initially **null**. This is the object on the left of Figure 3.6.

The code now executes the **for** loop that allocates the eight one-dimensional arrays with element type **int**. The body of the loop pushes the array reference and the element index, then allocates the sub-array with another "`newarr`" instruction. In this case the array is an **array of int**. The new element is assigned to the array by the "`stelem.ref`" instruction.

The code ends by incrementing the loop counter and testing the continuation condition for the loop. If the continuation condition is met, control jumps back to the loop label "`lb01`".

```
// int[][]  bar = new int[8][];
// for (int i = 0; i < 8; i++) bar[i] = new int[4];

      ldc.i4.8                  // push array length
      newarr int[]              // an array of int[]
      stsfld int[][] A.'bar'    // store the reference array
      ldc.i4.0                  // start the for loop
      stloc 'i'                 // initialize i to begin
lb01:                           // the loop header label
      ldsfld int[][] A.'bar'    // load the reference array
      ldloc 'i'                 // load the array index
      ldc.i4.4                  // push array length
      newarr int                // an array of 4 ints
      stelem.ref                // store int[] elem in int[][] array

      ldloc 'i'                 // fetch loop counter
      ldc.i4.1                  // push a literal one value
      add                       // add to the counter value
      dup                       // duplicate the new value
      stloc 'i'                 // save the updated counter
      ldc.i4.8                  // push a literal eight value
      blt lb01                  // test for end, or goto loop
```

**Figure 3.22:** *IL* to create array of arrays structure

The other object creation instruction, "`newobj`", performs two functions. First, it allocates space for the new object, and then it invokes the object constructor that is designated by the intstruction argument. A typical use of this instruction is —

```
ldstr "Assertion failure at Symbols.cp:315"
newobj instance void [mscorlib]System.Exception::.ctor(string)
```

In this example the argument to the constructor is pushed on the stack before the "`newobj`" instruction. The constructor is an instance method, but the receiver for the call is created by the first phase of execution of the instruction. This example creates an exception object, with an optional information string. The next instruction is almost certainly going to be "`throw`", as we shall see in the "Exception Handling" section (see page 95).

## Instructions for Value Types

There are a number of instructions that handle value types in managed code. These instructions initialize objects, load and store value objects on the evaluation stack, and copy the fields of a value object. There are also instructions to box and unbox value objects. In order

to see how value classes are copied and passed as various kinds of parameter, we shall look at some typical uses.

***Initializing and copying value objects.*** Data aggregates, such as arrays and classes, may declare that they have fields of value class types, and local variables of such types may be declared. In each case the space for the object is allocated when the encapsulating object or activation record is allocated. However, we may need to initialize or reinitialize the value of such value data. The "`initobj`" instruction does this. It expects the address of the value object on the stack, and initializes the bound object. A typical idiom for initializing a local variable of value class type would be —

```
.locals (value class [asm]Nsp.T 'vct')
    ...
    ldloca 'vct'
    initobj value class [asm]Nsp.T
```

The "`ldloca`" instruction pushes the address of the local variable onto the stack. The value is a transient pointer. There is also an instruction for copying the contents of value objects, "`cpobj`". This instruction may be used for assigment of entire values, although as we shall see in the following example, most compilers use a "`ldobj stobj`" sequence instead.

***Loading and storing.*** The "`ldobj`" and "`stobj`" instructions load and store value objects from the evaluation stack. It should be remembered that in *.NET* an abstract stack value may be a complete value object. The "`ldobj`" instruction expects an address on the stack, while the "`stobj`" instruction expects the value to be stored on the top of the stack with the destination address immediately below that.

An example shows most of the variations of loading and storing value classes. One such example is the short *C#* program in Figure 3.23. This program declares a **struct** named *ValCls*, which will be implemented as a value class. The class has two static methods, each with two formal parameters of the value class type. One method, *ValRef*, has a first parameter that is passed by value, and a second parameter passed by reference. The other method, *RefOut*, has a first parameter that is passed by reference, and a second that is also passed by reference, but marked with the **out**[9] attribute. Both methods simply copy the value of their first parameter to the second.

The program defines a second class with an entry point. The *Main* method calls the two instance methods, so that we may see how value class objects are passed as value and reference parameters.

The *Params* class declares two objects of value class type. These two objects, unlike the case with reference class objects, do not require any explicit initialization. If these two

[9] In *C#* an **out** formal parameter is a reference parameter with special behavior. The marker indicates that the parameter only carries information out of the method. This allows optimization of the call in situations that require marshalling.

```
public struct ValCls {
    public int i;
    public int j;
    public static void ValRef(ValCls inV, ref ValCls outV) {
        outV = inV;
    }
    public static void RefOut(ref ValCls inV, out ValCls outV) {
        outV = inV;
    }
}

public class Params {
    ValCls a;                                   // no new needed for value class
    ValCls b;                                   // no new needed for value class
    public static void Main() {                          // program entry point
        ValCls.ValRef(a, ref b);
        ValCls.RefOut(ref a, out b);
    }
}
```

**Figure 3.23:** Manipulating value objects in *C#*

variables had been declared as local variables of the *Main* method, then we would have had to initialize them explicitly with a call of **new**. Any such initialization calls will appear in the *IL* as inline occurrences of the "`initobj`" instruction.

The disassembly of the class with the entry point is shown in Figure 3.24. Here it may be seen that the first call, to the static method *ValRef*, passes its first parameter by value and its second by reference. The first actual parameter, *a*, is pushed onto the evaluation stack by a "`ldsfld`" (load static field) instruction. This is a value rather than a reference copy. The second actual parameter, *b*, is passed by reference, by having its address pushed by the "`ldsflda`" (load static field address) instruction.

The second call, to the static method *RefOut*, passes both of its parameters by reference. In both cases the addresses are pushed by the "`ldsflda`" instruction. Note that the same instruction is used to push the address of both ordinary reference (**inout**) and **out**-mode parameters.

The implementation of the value class is shown in Figure 3.25. In this figure we may see that for the method *ValRef* the reference parameter, marked **ref** in *C#*, is marked with an ampersand & in the *IL*. In the method *RefOut* both of the formal parameters are marked as reference, but the second parameter is also marked with the **out** attribute.

In order to assign a value of value class type, we always begin by pushing the address of the destination object. In the first method we do this by pushing the second parameter,

```
.class public auto ansi Params extends object
{
  .field private static value class ValCls a
  .field private static value class ValCls b
  .method public static void Main() il managed
  {
    .entrypoint
     ldsfld     value class ValCls Params::a
     ldsflda    value class ValCls Params::b
     call       void ValCls::ValRef(value class ValCls,
                                    value class ValCls&)
     ldsflda    value class ValCls Params::a
     ldsflda    value class ValCls Params::b
     call       void ValCls::RefOut(value class ValCls&,
                                    value class ValCls&)
     ret
  } // end of method Params::Main
} // end of class Params
```

**Figure 3.24:** Passing value objects as parameters

which is a transient pointer to the destination actual parameter. Since the first formal parameter is a copy of the actual value object, the "`ldarg.0`" instruction pushes the value. The assignment is then performed by a "`stobj`" instruction.

In the second method, as before, we begin by pushing the destination address, which is held in the second parameter. In this case however the first formal parameter is not the source value, but is a pointer to the source. We therefore push the address, using a "`ldarg.0`" instruction, but we must then use a "`ldobj`" instruction to push the bound object value. As in the first case, the value is assigned by a "`stobj`" instruction.

It might be noted that, throughout this example, the unqualified name of the class may be used, as it is local to the same assembly.

***Boxing and unboxing.*** Although we are able to create pointers to data of value types, these are not object references and are not self-describing. In order to be able to call virtual methods on value types, we must *box* the type. When we want to retrieve the value, we must then *unbox* the object. These are built-in primitives of the execution engine.

The instruction "`box`" takes a pointer to a value type and returns a reference to an object with a copy of the value instance embedded in it. The instruction takes an operator argument of the name of a value type. The object is treated by the verifier as being of type *System.Object*. The original pointer may point into the garbage-collected heap, if the value instance is a field of an object, or may be a local variable or argument, or even a static

```
.class value public auto ansi sealed ValCls
       extends [mscorlib]System.ValueType
{
  .field public int32 i
  .field public int32 j
  .method public static void
           ValRef(value class ValCls inV,
                  value class ValCls& outV) il managed {
    ldarg.1                      // push destination address
    ldarg.0                      // push incoming value
    stobj      ValCls            // store at destination address
    ret
  } // end of method ValCls::ValRef

  .method public static void
           RefOut(value class ValCls& inV,
            [out] value class ValCls& outV) il managed {
    ldarg.1                      // push destination address
    ldarg.0                      // push source value address
    ldobj      ValCls            // fetch source value
    stobj      ValCls            // store at destination address
    ret
  } // end of method ValCls::RefOut
} // end of class ValCls
```

**Figure 3.25:** Making copies of value class parameters

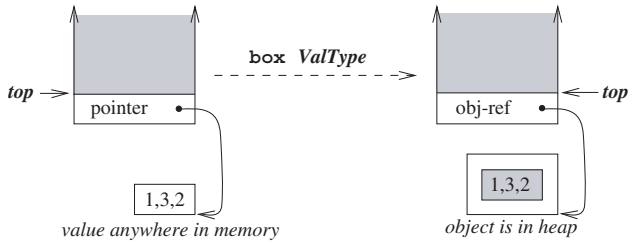


**Figure 3.26:** Evaluation stack before and after executing *box*

field of a class. The boxed object will certainly be in the heap. Figure 3.26 shows the relationship between the stack before and after execution of the instruction. In the figure it has been assumed that the value type is a value class that encapsulates three integers.

The "unbox" instruction takes a reference to a boxed value object and returns a managed pointer to the embedded value. The value is not copied and is still in the heap. Since

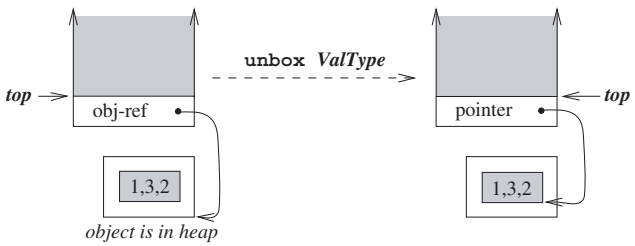


**Figure 3.27:** Evaluation stack before and after executing *unbox*

the garbage collector is aware of all managed pointers, this poses no problem. Figure 3.27 shows the relationship between the stack before and after execution of the instruction. In the next chapter, we shall examine an alternative strategy to this built-in facility, in which compiler-defined, named box classes are created for value classes.

***Typed references.*** There are a small number of instructions that are used for manipulating typed references. You will recall from the discussion of Figure 3.2 that typed references are opaque aggregates for attaching type information to references. These facilities are used for dealing with dynamically typed data.

Typed references are created by pushing a pointer to the data and executing the "`mkrefany`" instruction. The pointer may be either managed or unmanaged. The instruction has the type name as a operation argument. The result is a typed reference on the stack. This instruction is not verifiable if the verifier cannot prove that the pointer actually points to data of the nominated type.

The two components of a typed reference are a type token and the address of the data. These may be retrieved by the "`refanytype`" instruction, which returns the type token, and "`refanyval`", which returns the pointer contents as a managed pointer. Both of these require the typed reference on the stack. In the case of "`refanytype`" the instruction has no operator argument; "`refanyval`" must nominate the type expected.

# Defining Methods

Methods contain the executable code of a program and are the callable units of *CIL*. All such callable units are called "methods" in *IL*, whether they are static or dispatched procedures, and whether they are value returning functions or proper procedures. There are four different kinds of method declarations that are used in *IL*. These are *method declarations*, *method definitions*, *method references*, and *method implementations*.

The syntax for method declarations is —

MethodDecl → **.method** MethodHead "{" { MethodBodyItem } "}"
MethodHead → { MethodAttr } [ CallConv ] TypeRef *DottedName*
'('[Parameter { ',' Parameter }] ')' { ImplAttr } .

The method body items are the instructions and any directives associated with the implementation of the method body. The return type of the method together with the list of types of the parameters form the *signature* of the method.

Method declarations have a method head, but have no method body items. These are used for declaring the name and signature of a method so it may be used at a call-site. Method declarations are also used to declare abstract methods.

Method definitions are the main interest of this section. These define the name and signature of a method, and also give the body items that define the operational behavior of the method at runtime.

Method references are references to methods whose definition lies in another module or assembly. These are resolved to some concrete method at runtime.

A method implementation or *MethodImpl* associates a particular method body with some method declaration. In particular, a *MethodImpl* may be used to specify that a particular method should be used to implement a specified abstract method in the implementation of some interface type.

The method heading, as shown in the syntax, consists of method attributes, optional call convention markers, a return type, the method name, a parenthesized parameter list, and implementation attributes.

## Method Attributes

There are many different method attributes that are predefined. These include all of the accessibility attributes that are used for fields, repeated here in Figure 3.28. The various accessibility attributes have the same meaning and effect as when used on field declarations.

| *Attribute* | *Effect* |
|---|---|
| **assembly** | This method is only accessible within this assembly. |
| **family** | This method is only accessible in this class and in subclasses of this class. |
| **famandassem** | This method is accessible in this class and its subclasses but only within this assembly. |
| **famorassem** | This method is accessible in this class, its subclasses, and throughout this assembly. |
| **private** | This method is only accessible within this class. |
| **privatescope** | This method is only accessible within the assembly, and then only by using the method definition token. |
| **public** | This method is publicly accessible. |

**Figure 3.28:** Method declaration accessibility attributes

| *Attribute* | *Effect* |
|---|---|
| **abstract** | This method is abstract and must be declared abstract. |
| **virtual** | This method is virtual. |
| **static** | This method is static, and does not have a receiver |
| **final** | This method cannot be overridden in subclasses. |
| **hidebysig** | Hide by signature. |
| **newslot** | This method does not override any other method. |
| **specialname** | The name has special significance to tools. |
| **rtspecialname** | The name has significance to the runtime. |

**Figure 3.29:** Further method attributes

Other attributes apply just to methods. These specify the form of the method, whether static or virtual, and so on. They also give some additional information that may be used by the compiler for checking for correctness. These additional attributes are shown in Figure 3.29. Methods may be exactly one of *static*, *instance*, or *virtual*. Static methods do not have a **this** pointer. They are associated with a particular class, rather than an **instance** of a class. Plain (that is nonvirtual) instance methods are associated with an object instance that becomes the **this** pointer of the call. However, the particular method that is invoked is determined statically from the name and signature of the call. Virtual methods are associated with an object instance that becomes the **this** pointer of the call. However, unlike instance methods, the particular method that is invoked by the call is determined from the *v-table* attached to the **this** pointer. As noted, methods that are declared to be virtual may be called either through the *v-table* or statically.

If a method is declared to be abstract, then it must also be declared virtual. Such a method heading can only be part of a method *declaration*, rather than a complete method definition. We may declare that a virtual method is *final*. In this case the method cannot be overridden in any subclasses of the present class. This is a useful annotation, since it often allows the compiler to optimize calls to such methods. This marker is also used in cases where the overriding of a particular method might lead to the breaking of invariants in the present class. The *hidebysig* attribute specifies that methods only obscure inherited methods of the same name if the signatures match. This attribute is the default and is ignored by the runtime. Finally, the *newslot* attribute declares that this method does not override an equally named inherited method.

## Call Conventions

The call convention of the method header specifies how the method is to be invoked. The most common of these are shown in Figure 3.30. It is not strictly necessary to specify the *instance* marker in method definitions, since it may be deduced from the other attributes. However, it is always necessary in method references at call sites, since the method attributes are absent in that location.

| *Attribute* | *Effect* |
|---|---|
| **instance** | This method has a **this** pointer. |
| **explicit** | Only used directly following **instance**, the **this** appears explicitly in the parameter list. |
| **varargs** | This method uses the varargs convention. |
| *others* | Various native call convention markers. |

**Figure 3.30:** Various call convention attributes

## Implementation Attributes

Implementation attributes follow the parameter list of the method declaration. Most commonly the attributes are just *il managed*, but there are several other possibilities. The most likely attributes are shown in Figure 3.31. The *il managed synchronized* combination has been seen in connection with the "`add_*`" and "`remove_*`" methods of events, which necessarily contain critical sections. The constructor and *Invoke* methods of delegates provide examples of the *runtime-managed* combination.

| *Attribute* | *Effect* |
|---|---|
| **il** | This method body is specified by *IL*. |
| **runtime** | The method body is supplied by the runtime. |
| **native** | The method body is native code. |
| **managed** | This method is managed. |
| **unmanaged** | This method is unmanaged. |
| **synchronized** | This method must be executed in a single thread, and should block other callers. |

**Figure 3.31:** Various method implementation attributes

## Signatures

The signature of a method consists of the return type (or **void** in the case of a proper procedure) and the ordered list of parameter types and modes. Each element of the parameter list has optional modes and an optional identifier.

Parameter → [ParamAttributes] Type [*identifier*] .
ParamAttributes → '**[in]**' | '**[out]**' | '**[opt]**' .

The **in** and **out** attributes may only be used with parameters of reference type and do not form part of the signature so far as matching is concerned. They are hints to the compiler so

that efficient marshalling may be attempted in cases of remote calling. The optional name is simply a convenience in textual *IL*; `ilasm` will map such occurrences of names to the ordinal numbers used in the *PEM*.

The **opt** attribute indicates that the parameter is optional from the point of view of the source language. The *CLR* will still expect to receive a value, but that value may be supplied by a tool rather than by code in the source program.

The types that appear in parameter lists indicate whether the corresponding parameter is passed by value or by reference. Reference parameters that are managed pointers have the type followed by the ampersand character "`&`". The type itself consists of a type reference followed by zero or more array markers "`[]`".

## Method Pointers

Method pointers may be created by the "`ldftn`" and"`ldvirtftn`" instructions. Some uses of the "`ldftn`" instruction were discussed on page 40 in the previous chapter.

Locations intended to hold function pointers may be declared bound to functions with a specified signature. Uses of such data are seldom verifiable. Function pointer values are scalar types, holding just an address, possibly the entry point of the method. For languages that allow nested procedure declarations, such pointers are inadequate, since they are not associated with the activation of an enclosing method.

The "`ldvirtftn`" instruction takes a reference to an object on the top of the stack and returns a pointer to the virtual function bound to the exact type of the object. The operator argument is the same as for a virtual call of the same method. Thus the code sequence —

```
<push receiver>
<push args>
<push receiver>
ldvirtftn   returnType MethClass::methName(<arg type list>)
calli   returnType (<arg type list>)
```

calls exactly the same method as —

```
<push receiver>
<push args>
callvirt   returnType MethClass::methName(<arg type list>)
```

Of course, the point of creating a function pointer is to store it in some location or pass it as a parameter to some call. If the function pointer in the above example was to be stored in a local variable, the location could be declared with the location signature —

```
.locals(... ,
        method returnType*(<arg type list>) 'mpnm',
        ...  )
```

Note the use of the abstract declarator format. The signature of the concrete method is copied, but with the qualified name of the method replaced by the star character '*'.

## Overriding and Overloading

Whenever a new method is introduced it is necessary to consider the effect of the new declaration on the accessibility of any other equally named method. There are two separate mechanisms at work: *overriding* and *overloading*.

***Overriding.*** In the object model of the *CTS* virtual methods are inherited from the supertype of each class. Suppose that we wish to dispatch a virtual method with a particular name and signature on some particular receiver object. Let us further suppose that several of the ancestor types of the exact type of the receiver object define a matching procedure. The dispatch mechanisms of the *CLR* will ensure that the matching method from the exact type will be invoked, or if there is no such method, the matching method from the closest ancestor will be invoked. This does not mean to say that some kind of search is performed at runtime. At class load time the values in the *v-table*s are organized in such a way that the effect of the lookup algorithm is obtained by a simple lookup in the *v-table* of the receiver object.

The summary effect of the resolution algorithm is that whenever a new virtual method is defined it hides any equally named method with the same signature. This *overriding* behavior is one of the pillars of object-oriented programming.

Note that this particular behavior is specific to the dispatch of virtual methods. If methods are simply (nonvirtual) instance methods, or are invoked by the "`call`" rather than the "`callvirt`" instruction, then the hiding does not occur. To be specific, consider a class *SubC* that extends a class *SuperC*, where both classes declare a virtual no-arg void method *Foo*. Let us suppose that that we have a reference that is statically of type *SuperC*. The *IL* instruction —

```
callvirt instance void ThisNameSpace.SuperC::'Foo'()
```

will invoke either *SuperC::Foo* or *SubC::Foo* depending on whether the exact type of the reference is *SuperC* or *SubC*, respectively. However the two instructions —

```
call instance void ThisNameSpace.SuperC::'Foo'()
call instance void ThisNameSpace.SubC::'Foo'()
```

will call precisely the named methods without regard to the exact type of the reference.

So far as the *CTS* is concerned, any definition of a matching method in a subclass replaces the inherited method in the corresponding slot of the virtual method table of the subclass. Such a definition does not make the inherited method inaccessible for statically bound invocations. Whether or not any particular source language has some syntax for exploiting this capability of the *CTS* is another matter entirely.

It is important to recognize that overriding occurs based on the entire name and signature of the method. If two methods are defined that have the same name and parameter list, but differ in return value, then they will occupy separate slots in the *v-table*. Both *C#* and *Java* would reject programs which try to do this, although it is a common feature of languages that permit *return type covariance*. In a later chapter we shall explore ways of programming around this restriction in the *CTS*.

***Overloading.*** If two methods of a class have the same name but different signatures, then the method name is said to be *overloaded*. Some people believe that overloading is inherently evil and prefer languages that outlaw the practice. Others argue that overloading is convenient and unharmful. Still others take the middle ground and believe that overloading is admissible for constructor methods but nowhere else. This is another deeply religious issue. However, the plain truth is that many widely used languages require overloading, and hence overloading must be efficiently supported by the *CLR*.

In principle, it is the job of the compiler to resolve any applied occurrence of a method name to one exact name-and-signature combination. This may be a nontrival exercise in the presence of implicit type coercions, and the rules vary from language to language.[10]

Both *C#* and *Java* allow overloading but only on the number and type of the parameters. Overloading on the basis of the return type of the method is not allowed in either of these languages, even though the *CTS* is able to support this. *Component Pascal* does not permit overloading but curiously does permit covariance of return type for overridden methods.

So far as the *CLR* is concerned, methods are matched on the basis of the complete name-and-signature. The possibility of having different methods for which the simple name is overloaded follows directly.

A more interesting question is whether standard libraries ought to define methods that require users to resolve overloaded simple names. Certainly the *.NET* answer to this question is "yes," with the standard class libraries having many sets of overloaded methods. This particular choice poses difficult questions for users of the libraries that code in languages that do not support overloading. Some attempts to answer these questions are discussed in Chapter 10.

[10] And, in the case of *Java*, between preliminary and final definition of the same language.

## Method Bodies

The method body is a sequence of method body items. These include instructions, labels, and various directives. Most of these will be dealt with in other sections of this book. Figure 3.32 is a summary of the most common method body items. The *.entrypoint* directive states that this method is the entry point of the application. Only one such method may be so marked in the whole assembly. In a *C#* program the method would be named *Main*.

A *.locals* declaration introduces one or more local variables. The syntax is —

LocalDecl → **.locals** '(' LocalSignature { ',' LocalSignature } ')' .

A complete discussion of the syntax and options for local variable declaration is given on page 219. In the same section there is a discussion of how the maximum stack height may be computed. It is fortunate that both the maximum stack declaration and the locals declaration may be placed anywhere within the method body. *gpcp* computes the maximum stack height during emission of instructions to the *IL* and declares additional temporary local variables during the code emission process. For compilers that write out their code this way, the values are not known until emission is complete for a particular method.

| *Body Item* | *Description* |
|---|---|
| **.entrypoint** | This method is the application entry point. |
| **.locals** | Declares local variables for this method. |
| **.maxstack** | Specifies the maximum stack height. |
| **.line** | Specifies a source line number. |
| *instruction* | An *IL* instruction. |
| *label* | A code label in the *IL*. |

**Figure 3.32:** Various method body items

# Exception Handling

The *CLR* provides facilities for many different structured exception-handling models. With the present exception of what have been called *resumption models*, the *CLR* supports the implementation of exception handling in most languages. This support comprises two parts. First, there are instructions and directives in *CIL* for the primitive operations of throwing and catching exceptions. Secondly, there is a base type in the *CTS* from which all exceptions classes should directly or indirectly derive.

In this context we use the word *exception* to mean any abnormal execution state. Entry into the abnormal state may have been requested by program code or by a check performed by the *VES*. Some languages allow source programs to explicitly raise exceptions using some dedicated syntactic construct. Other languages require that the compiler

test certain assertions and explicitly raise an exception in the event that the test fails. For example, in most *Pascal*-family languages it is an error if a **case** statement does not select any case, and the statement does not have an **else** clause. In both of these cases the *IL* of the program will have an explicit use of the "`throw`" instruction. The *VES* responds to a very large number of program errors by raising an exception of some type or another. Failure of class casts, indexing out of bounds on arrays, attempting to divide by zero, or attempting to load an assembly that cannot be found are just some of these.

Whenever a thread of control is in an exceptional state, there is an object called *the exception object* that plays a special role. In the case of explicit executions of "`throw`", the exception object is explicitly constructed by a call of "`newobj`" with an appropriate constructor method. A typical idiom would be —

```
    brtrue lb01                                    // skip if assertion is true
    ldstr "Assertion failure at Symbols.cp:315"
    newobj instance void [mscorlib]System.Exception::.ctor(string)
    throw
lb01:
```

## The Basics

The underlying basis of the exception-handling model of *CIL* is a **try ... catch ... finally** structure, with semantics similar to those of *C#* or *Java*. Instructions that appear within a **try** block are said to be in a *protected block*. If any exception is raised in this code, or an unhandled exception is raised in code called by the code of the protected block, then execution of the protected block is stopped, and the *VES* searches for a *handler*. Before any handling action is taken, any **finally** code associated with the protected block is executed.

The *VES* checks the scope of the declared handlers within the current method, to see if the protected block to which they apply covers the instruction that raised the exception. If such a handler is found, then that handler is selected; otherwise the search continues in the calling method.

Handlers may be filtered or unfiltered. An unfiltered handler accepts any exception that is raised, and control enters the code of the handler with the exception object being the sole object on the evaluation stack. The code of the handler must pop this reference before it calls "`leave`".

There are two kinds of filtered handler. The most used kind is declared to accept some particular class of exception object. When such a handler is selected the *VES* performs a type check on the current exception to see if it is the nominated type, or a subtype of that type. If that is the case, the code of the handler is entered, with the exception object on the evaluation stack so that the code may do further tests on the object.

It is also possible for a program to declare explicit tests on the exception object with code in a **filter** block. The code of this block finishes by pushing a Boolean on the stack

to indicate whether the handler block should be entered. In the event that a filtered handler declines to handle an exception, the search for a handler resumes as before.

Programs may also define **finally** blocks. These blocks are guaranteed to always be executed after completion of the **try** block, even if the execution terminated abnormally. These are typically used to ensure that resources are reclaimed before execution is abandoned.

There are three different formats that may be used to define the various regions of a structured exception handler. At runtime the exception handling regions are defined by tables associated with each method. These tables denote the limits of the various regions in terms of their offsets from the method entry point. This is the way in which exception handling appears in the disassembler output. It is also possible to define the regions in terms of labels, rather than offsets. Finally, it is possible to write *CIL* in a structured way, with keywords and braces delimiting the scope of the various blocks. This is the only approach that we will use in the examples here.

## Defining the Blocks

A grammar of structured exception-handling block (*SehBlock*) in *IL*, for the structured format that we consider, is as follows —

| | | |
|---|---|---|
| SehBlock | → | TryBlock SehClause { SehClause } . |
| TryBlock | → | **.try** "{" *instructionSequence* "}" . |
| SehClause | → | **catch** typeRef "{" *instructionSequence* "}" |
| | \| | **filter** *label* "{" *instructionSequence* "}" |
| | \| | **finally** "{" *instructionSequence* "}" |
| | \| | **fault** "{" *instructionSequence* "}" . |

Such a structured exception handler block may occur within any instruction sequence, leading to nested exception-handling regions. We shall deal with each of these types of handler clauses in order.

***The .try block.*** A try block is a sequence of instructions enclosed within braces, following the keyword **.try**. Control may only exit from a try block by an exception being thrown or by the special instruction "`leave`". In particular, it is not possible to branch either into or out of a try block, and it is not possible to return out of a try block by a "`ret`" instruction.[11]

The "`leave`" instruction takes a label as instruction argument. The label is the destination to which control transfers, possibly after a **finally** block is executed. The evaluation

[11] Of course, it is quite possible to write a *C#* program that "returns" out of a try block. If you check the resulting *IL* however, you will find that the return has been translated into a "`leave`" instruction that jumps to a "`ret`" outside of the block.

stack must be empty when the "`leave`" is executed. This means that value-returning functions may need to store their return values in a local variable. Figure 3.33 shows a typical idiom, for a function returning a Boolean value.

```
.locals(bool 'retVal')          // local to hold return value
      ...
.try {
      ...
      <compute return value on stack>
      stloc 'retVal'            // save result value in local
      leave lb01                // jump to label lb01
} catch [mscorlib]System.Exception {
      ...
}
lb01:
      ldloc 'retVal'            // fetch the function result
      ret                       // now return the function result
```

**Figure 3.33:** Returning a value from inside a **try** block

***The catch block.*** In most languages, the **catch** block is the mechanism for handling exceptions. The definition of a catch block nominates the type that the handler is able to handle.

SehClause → ...
| **catch** *typeRef* "{" *instructionSequence* "}".

The *VES* will perform filtering on the exception objects to ensure that the block is entered only with objects of the nominated type.

Control enters the catch block with the exception object on an otherwise empty evaluation stack. The code of the block is responsible for ensuring that this object is popped from the stack before control exits the block. As with a **try** block, control may only leave a catch block by raising another exception or by executing "`leave`".

Here are two examples of using the catch block to achieve similar semantics, but originating from differing styles of encoding in the source language. In *C#*, a handler that performs different recovery actions based on the type of the exception object would be encoded with multiple **catch** clauses, as shown in Figure 3.34.

In *IL*, the multiple catch clauses map directly into separate type-filtered catch clauses. Figure 3.35 shows the structure. Notice that each catch block must finish with a "`leave`" instruction. The exception object on the stack at the entry to each catch block must be popped within the block.

```
try {
    ...                     // body of try block
} catch (TypeA x) {
    ...                     // x is known to have type TypeA
} catch (TypeB x) {
    ...                     // x is known to have type TypeB
} catch (TypeC x) {
    ...                     // x is known to have type TypeC
}
```

**Figure 3.34:** **try** with multiple **catch**es in *C#*

```
.try {
    ...                     // body of try block
    leave exitLb
} catch TypeA {
    ...                     // x is known to have type TypeA
    leave exitLb
} catch TypeB {
    ...                     // x is known to have type TypeB
    leave exitLb
} catch TypeC {
    ...                     // x is known to have type TypeC
    leave exitLb
}
exitLb:
```

**Figure 3.35:** **try** with multiple **catch**es in *IL*

The *Component Pascal* dialect accepted by *gpcp* has a single **rescue** handler per procedure. Thus, any selection on exception object type must be explicit in the source language handler. Semantics similar to Figure 3.34 may be achieved in *gpcp* with the source fragment in Figure 3.36.

In this case the *IL* contains a single catch clause, filtered in this case on the exception base type *System.Exception*. Inside this, **catch**, a sequential type-selection structure, tests the type of the exception object. Figure 3.37 sketches the structure in the *IL*.

There is a slight semantic difference in the *Component Pascal* case. In the *C#* example if no **catch** is selected the search for a handler continues in the caller. In the *Component Pascal* case, it is an error for no case to be selected. If the exact *C#* semantics are required, the **with** statement in Figure 3.36 must end with **else** *RTS.throw*(x) to rethrow the same exception in the context of the caller.

```
begin
    ...                     (* body of "try" block *)
rescue (x)                  (* local x has type System.Exception *)
    with x : TypeA do
      ...                   (* x is known to have type TypeA *)
    | x : TypeB do
      ...                   (* x is known to have type TypeB *)
    | x : TypeC do
      ...                   (* x is known to have type TypeC *)
    end (* with *)
end (* proc *)
```

**Figure 3.36:** **rescue** with type selection in *gpcp*

```
.try {
    ...                     // body of try block
    leave exitLb
} catch [mscorlib]System.Exception {
    stloc 'x'               // store in local variable x
    ldloc 'x'               // push exception object
    isinst TypeA            // test for object of TypeA
    brfalse lb01
    ...                     // x is known to have type TypeA
    leave exitLb
lb01:
    ldloc 'x'               // push exception object
    isinst TypeB            // test for object of TypeB
    brfalse lb02
    ...                     // x is known to have type TypeB
    leave exitLb
lb02:
    ldloc 'x'               // push exception object
    isinst TypeC            // test for object of TypeC
    brfalse lb03
    ...                     // x is known to have type TypeC
    leave exitLb
lb03:
    ...                     // with statement trap here
}
exitLb:
```

**Figure 3.37:** **rescue** with explicit type selection

***The filter block.*** In languages in which it is necessary to select handlers on criteria other than the type of the exception object, the **filter** clause may be used. In this structure an *IL* code label is nominated in the declaration —

SehClause → ...
| **filter** *label* "{" *instructionSequence* "}".

The code of the filter consists of two logical parts. These are the filtering logic and the handler proper. When an exception is raised in a block protected by a **filter** clause, control is passed to the designated label, with the exception object on the evaluation stack. The filtering logic that follows the designated label is responsible for popping the exception object from the stack. It must also decide whether the handler should be entered, and push a Boolean result onto the stack. The filtering code returns using the special "`endfilter`" instruction, with just the Boolean result on the stack. The handler code itself has the same constraints as a **catch** block and has the exception object on the stack at entry, as usual.

In the defining syntax, the instruction sequence following the **filter** keyword is the *IL* of the handler. The code of the filter may be anywhere within the same method and cannot overlap with any **try** block. The instructions of the filter would typically be parked at the end of the method, following the return instruction that terminates normal control flow.

Filtered handler blocks would be used when information other than type determines the applicability of a particular handler. A typical idiom would define some extension of *System.Exception* with some additional fields set by the class constructors. The filtering logic would retrieve the exception object, perform whatever computation was required on the new fields, and push an appropriate Boolean.

***The finally block.*** A **finally** clause contains code that is executed at the end of a **try** block, whether the block is terminated normally or exceptionally. The block must end with the special "`endfinally`" instruction, with an empty stack.

SehClause → ...
| **finally** "{" *instructionSequence* "}".

A **finally** clause is thus entered under two circumstances. If the exit from the **try** block was normal—that is, the "`leave`" instruction was executed—then the instructions of the handler are executed and control continues to the label designated in the "`leave`". If the exit was exceptional, then the instructions of the handler are executed, and the search for a **catch** handler either begins, or continues.

Notice that the execution of the **finally** clause does not affect the state of the exception handling. The clause *observes* the state of the exception system but does not *handle* any exception.

***The fault block.*** A **fault** clause is used if some special action must be taken when an exception is detected, but not otherwise. If the associated **try** block was completed by the "`leave`" instruction, the **fault** clause is skipped.

SehClause → ...
| **fault** "{" *instructionSequence* "}".

As is the case for the **finally** clause, this clause *observes* the exception state but does not modify it. The clause must end with the "`endfault`" instruction, which is a synonym for "`endfinally`".

# Notes on Chapter 3

There is a huge amount of material on the *CTS* that simply has to be left out of a chapter of this kind. Hopefully what has been included is enough to get started, with the documentation filling in the finer detail on a demand-driven basis.

Languages that do not directly have the notions of properties and events can still access properties and events in the framework, through the method call pathway. Chapter 10 gives some further discussion of this issue. There is also another, *indexed* form of property that has not been mentioned in the current chapter. These *indexed properties* are included in the discussion in Chapter 10.

An interesting taxonomy of exception-handling semantics was given by Drew and Gough in "Exception Handling: Expecting the Unexpected" in the journal *Computer Languages* (Vol. 38, No. 8), 1994. The paper reviews the world immediately before the *try, catch* model achieved its current dominance.

A number of example programs relating to the material in this chapter are available from the Software Automata web site referenced in the notes from Chapter 1.