Reference Type Internals

We begin with reference types, whose memory layout is fairly complex and has significant effect on their runtime performance. For the purpose of our discussion, let's consider a textbook example of an `Employee` reference type, which has several fields (instance and static), and a few methods as well:

```
public class Employee
{
  private int _id;
  private string _name;
  private static CompanyPolicy _policy;
  public virtual void Work() {
   Console.WriteLine("Zzzz...");
  }
  public void TakeVacation(int days) {
   Console.WriteLine("Zzzz...");
  }
  public static void SetCompanyPolicy(CompanyPolicy policy) {
   _policy = policy;
  }
}
```

Now consider an instance of the `Employee` reference type on the managed heap. Figure 3-2 describes the layout of the instance in a 32-bit .NET process:

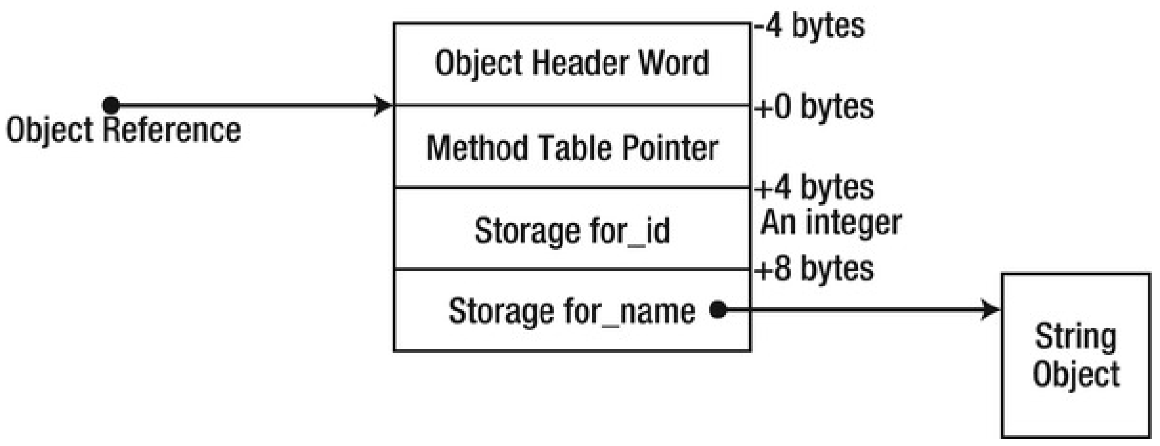

Figure 3-2. The layout of an Employee instance on the managed heap, including the reference type overhead

The order of the `_id` and `_name` fields inside the object is not guaranteed (although it can be controlled, as we will see in the "Value Type Internals" section, using the `StructLayout` attribute). However, the object's memory storage begins with a four byte field called *object header word* (or sync block index), followed by another four byte field called *method table pointer* (or type object pointer). These fields are not directly accessible from any .NET language – they serve the JIT and the CLR itself. The object reference (which, internally, is simply a memory address) points to the beginning of the method table pointer, so that the object header word is situated at a negative offset from the object's address.

Note On 32-bit systems, objects in the heap are aligned to the nearest four byte multiple. This implies that an object with only a single `byte` member would still occupy 12 bytes in the heap, due to alignment (in fact, even a class with no instance fields will occupy 12 bytes when instantiated). There are several differences where 64-bit systems are introduced. First, the method table pointer field (a pointer that it is) occupies 8 bytes of memory, and the object header word takes 8 bytes as well. Second, objects in the heap are aligned to the nearest eight byte multiple. This implies that an object with only a single `byte` member in a 64-bit heap would occupy a whopping 24 bytes of memory. This only serves to demonstrate more strongly the memory density overhead of reference types, in particular where small objects are created in bulk.

The Method Table

The Method Table Pointer points to an internal CLR data structure called a method table (MT), which points in turn to another internal structure called EEClass (EE stands for Execution Engine). Together, the MT and EEClass contain the information required to dispatch virtual method calls, interface method calls, access static variables, determine the type of a runtime object, access the base type methods efficiently, and serve many additional purposes. The method table contains the frequently accessed information, which is required for the runtime operation of critical mechanisms such as virtual method dispatch, whereas the EEClass contains information that is less frequently accessed, but still used by some runtime mechanisms, such as Reflection. We can learn about the contents of both data structures using the `!DumpMT` and `!DumpClass` SOS commands and the Rotor (SSCLI) source code, bearing in mind that we are discussing internal implementation details that might differ significantly between CLR versions.

Note SOS (Son of Strike) is a debugger extension DLL, which facilitates debugging managed applications using Windows debuggers. It is most commonly used with WinDbg, but can be loaded into Visual Studio using the Immediate Window. Its commands provide insight into CLR internals, which is why we are using it often in this chapter. For more information about SOS, consult the inline help (the `!help` command after loading the extension) and the MSDN documentation. An excellent treatment of SOS features in conjunction with debugging managed applications is in Mario Hewardt's book, "Advanced .NET Debugging" (Addison-Wesley, 2009).

The location of static fields is determined by the EEClass. Primitive fields (such as integers) are stored in dynamically allocated locations on the loader heap, while custom value types and reference types are stored as indirect references to a heap location (through an AppDomain-wide object array). To access a static field, it is not necessary to consult the method table or EEClass – the JIT compiler can hard-code the addresses of the static fields into the generated machine code. The array of references to static fields is pinned so that its address can't change during a garbage collection (discussed in more detail in Chapter 4), and the primitive static fields reside in the method table which is not touched by the garbage collector either. This ensures that hard-coded addresses can be used to access these fields:

```
public static void SetCompanyPolicy(CompanyPolicy policy)
{
   _policy = policy;
}
mov ecx, dword ptr [ebp+8] ;copy parameter to ECX
mov dword ptr [0x3543320], ecx ;copy ECX to the static field location in the global pinned array
```

The most obvious thing the method table contains is an array of code addresses, one for every method of the type, including any virtual methods inherited from its base types. For example, Figure 3-3 shows a possible method table layout for the `Employee` class above, assuming that it derives only from `System.Object`:

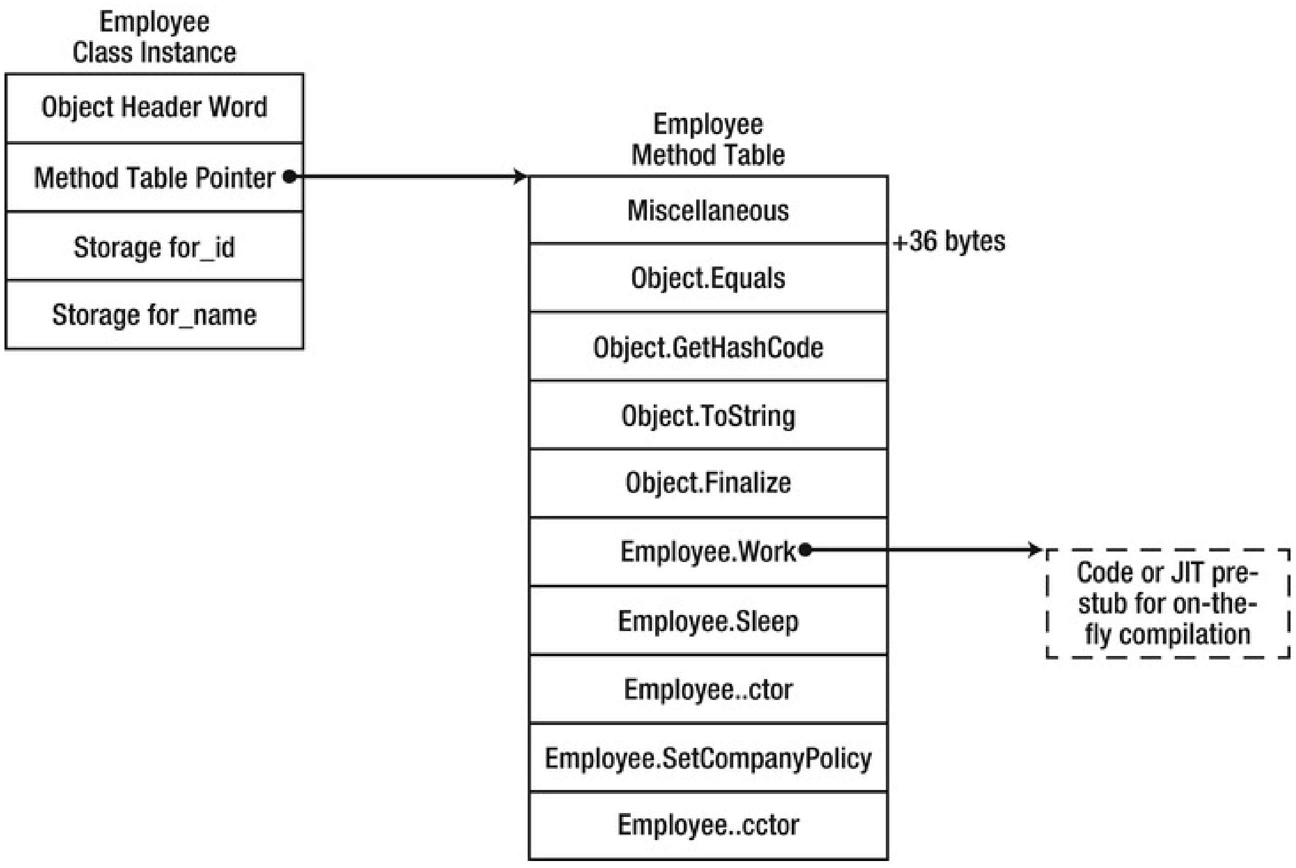

Figure 3-3. The Employee class' method table (partial view)

We can inspect method tables using the `!DumpMT` SOS command, given a method table pointer (which can be obtained from a live object reference by inspecting its first field, or by named lookup with the `!Name2EE` command). The -md switch will output the method descriptor table, containing the code addresses and method descriptors for each of the type's methods. (The `JIT` column can have one of three values: `PreJIT`, meaning that the method was compiled using NGEN; `JIT`, meaning that the method was JIT-compiled at runtime; or `NONE`, meaning that the method was not yet compiled.)

```
0:000> r esi
esi=02774ec8
0:000> !do esi
Name: CompanyPolicy
MethodTable: 002a3828
EEClass: 002a1350
Size: 12(0xc) bytes
File: D:\Development\...\App.exe
Fields:
None
0:000> dd esi L1
02774ec8 002a3828
0:000> !dumpmt -md 002a3828
EEClass: 002a1350
```

```
Module: 002a2e7c
Name: CompanyPolicy
mdToken: 02000002
File: D:\Development\...\App.exe
BaseSize: 0xc
ComponentSize: 0x0
Slots in VTable: 5
Number of IFaces in IFaceMap: 0
--------------------------------------
MethodDesc Table
Entry    MethodDe   JIT Name
5b625450 5b3c3524 PreJIT System.Object.ToString()
5b6106b0 5b3c352c PreJIT System.Object.Equals(System.Object)
5b610270 5b3c354c PreJIT System.Object.GetHashCode()
5b610230 5b3c3560 PreJIT System.Object.Finalize()
002ac058 002a3820 NONE CompanyPolicy..ctor()
```

Note Unlike C++ virtual function pointer tables, CLR method tables contain code addresses for *all* methods, including non-virtual ones. The order in which methods are laid out by the method table creator is unspecified. Currently, they are arranged in the following order: inherited virtual methods (including any possible overrides – discussed later), newly introduced virtual methods, non-virtual instance methods, and static methods.

The code addresses stored in the method table are generated on the fly – the JIT compiler compiles methods when they are first called, unless NGEN was used (discussed in Chapter 10). However, users of the method table need not be aware of this step thanks to a fairly common compiler trick. When the method table is first created, it is populated with pointers to special pre-JIT stubs, which contain a single `CALL` instruction dispatching the caller to a JIT routine that compiles the relevant method on the fly. After compilation completes, the stub is overwritten with a `JMP` instruction that transfers control to the newly compiled method. The entire data structure which stores the pre-JIT stub and some additional information about the method is called a method descriptor (MD) and can be examined by the `!DumpMD` SOS command.

Before a method has been JIT-compiled, its method descriptor contains the following information:

```
0:000> !dumpmd 003737a8
Method Name:   Employee.Sleep()
Class:   003712fc
MethodTable:   003737c8
mdToken:   06000003
Module:   00372e7c
IsJitted:   no
CodeAddr:       ffffffff
Transparency:   Critical
```

Here is an example of a pre-JIT stub that is responsible for updating the method descriptor:

```
0:000> !u 002ac035
Unmanaged code
002ac035 b002 mov al,2
002ac037 eb08 jmp 002ac041
002ac039 b005 mov al,5
002ac03b eb04 jmp 002ac041
002ac03d b008 mov al,8
002ac03f eb00 jmp 002ac041
002ac041 0fb6c0 movzx eax,al
002ac044 c1e002 shl eax,2
002ac047 05a0372a00 add eax,2A37A0h
002ac04c e98270ca66 jmp clr!ThePreStub (66f530d3)
```

After the method was JIT-compiled, its method descriptor changes to the following:

```
0:007> !dumpmd 003737a8
Method Name: Employee.Sleep()
Class: 003712fc
MethodTable: 003737c8
mdToken: 06000003
Module: 00372e7c
IsJitted: yes
CodeAddr: 00490140
Transparency: Critical
```

A real method table contains more information that we have previously disclosed. Understanding some of the additional fields is critical to the details of method dispatch discussed next; this is why we must take a longer look at the method table structure for an `Employee` instance. We assume additionally that the `Employee` class implements three interfaces: `IComparable`, `IDisposable`, and `ICloneable`.

In Figure 3-4, there are several additions to our previous understanding of method table layout. First, the method table header contains several interesting flags that allow dynamic discovery of its layout, such as the number of virtual methods and the number of interfaces the type implements. Second, the method table contains a pointer to its base class's method table, a pointer to its module, and a pointer to its EEClass (which contains a back-reference to the method table). Third, the actual methods are preceded by a list of interface method tables that the type implements. This is why there is a pointer to the method list within the method table, at a constant offset of 40 bytes from the method table start.

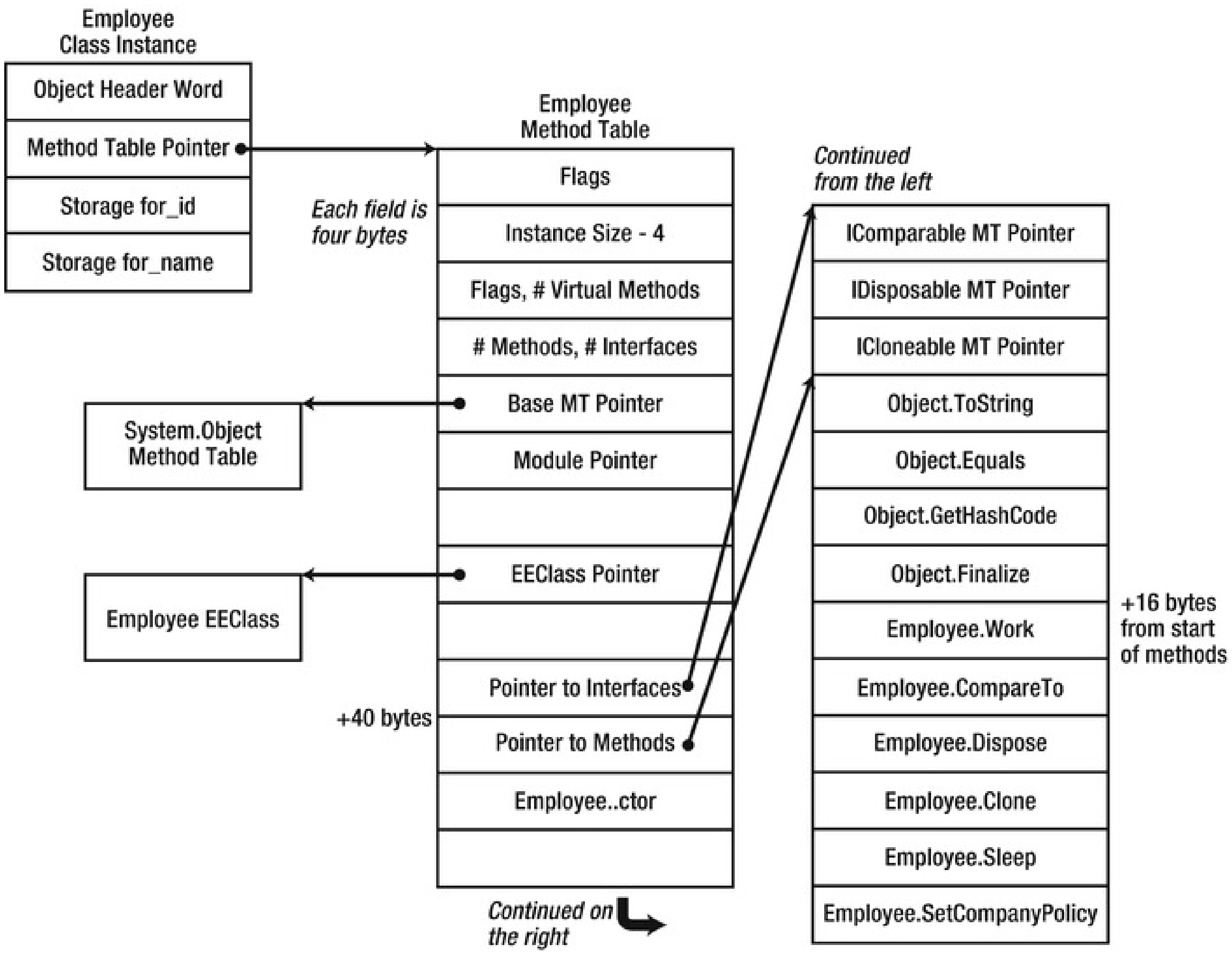

Figure 3-4. Detailed view of the Employee method table, including internal pointers to interface list and method list used for virtual method invocation

Note The additional dereference step required to reach the table of code addresses for the type's methods allows this table to be stored separately from the method table object, in a different memory location. For example, if you inspect the method table for `System.Object`, you may find that its method code addresses are stored in a separate location. Furthermore, classes with many virtual methods will have several first-level table pointers, allowing partial reuse of method tables in derived classes.

Invoking Methods on Reference Type Instances

Clearly, the method table can be used to invoke methods on arbitrary object instances. Suppose that the stack location `EBP-64` contains the address of an `Employee` object with a method table layout as in the previous diagram. Then we can call the `Work` virtual method by using the following instruction sequence:

```
mov ecx, dword ptr [ebp-64]
mov eax, dword ptr [ecx] ; the method table pointer
mov eax, dword ptr [eax+40] ; the pointer to the actual methods inside the method table
call dword ptr [eax+16] ; Work is the fifth slot (fourth if zero-based)
```

The first instruction copies the reference from the stack to the `ECX` register, the second instruction dereferences the `ECX` register to obtain the object's method table pointer, the third instruction fetches the internal pointer to the list of methods inside the method table (which is located at the constant offset of 40 bytes), and the fourth instruction dereferences the internal method table at offset 16 to obtain the code address of the `Work` method and calls it. To understand why it is necessary to use the method table for virtual method dispatching, we need to consider how runtime binding works – i.e., how polymorphism can be implemented through virtual methods.

Suppose that an additional class, `Manager`, were to derive from `Employee` and override its `Work` virtual method, as well as implement yet another interface:

```
public class Manager : Employee, ISerializable
{
  private List<Employee> _reports;
  public override void Work() ...
  //...implementation of ISerializable omitted for brevity
}
```

The compiler might be required to dispatch a call to the `Manager.Work` method through an object reference that has only the `Employee` static type, as in the following code listing:

```
Employee employee = new Manager(...);
employee.Work();
```

In this particular case, the compiler might be able to deduce – using static flow analysis – that the `Manager.Work` method should be invoked (this doesn't happen in the current C# and CLR implementations). In the general case, however, when presented with a statically typed `Employee` reference, the compiler needs to defer binding to runtime. In fact, the only way to bind to the right method is to determine at runtime the actual type of the object referenced by the `employee` variable, and dispatch the virtual method based on that type information. This is exactly what the method table enables the JIT compiler to do.

As depicted in Figure 3-5, the method table layout for the `Manager` class has the `Work` slot overridden with a different code address, and the method dispatch sequence would remain the same. Note that the offset of the overridden slot from the beginning of the method table is different because the `Manager` class implements an additional interface; however, the "Pointer to Methods" field is still at the same offset, and accommodates for this difference:

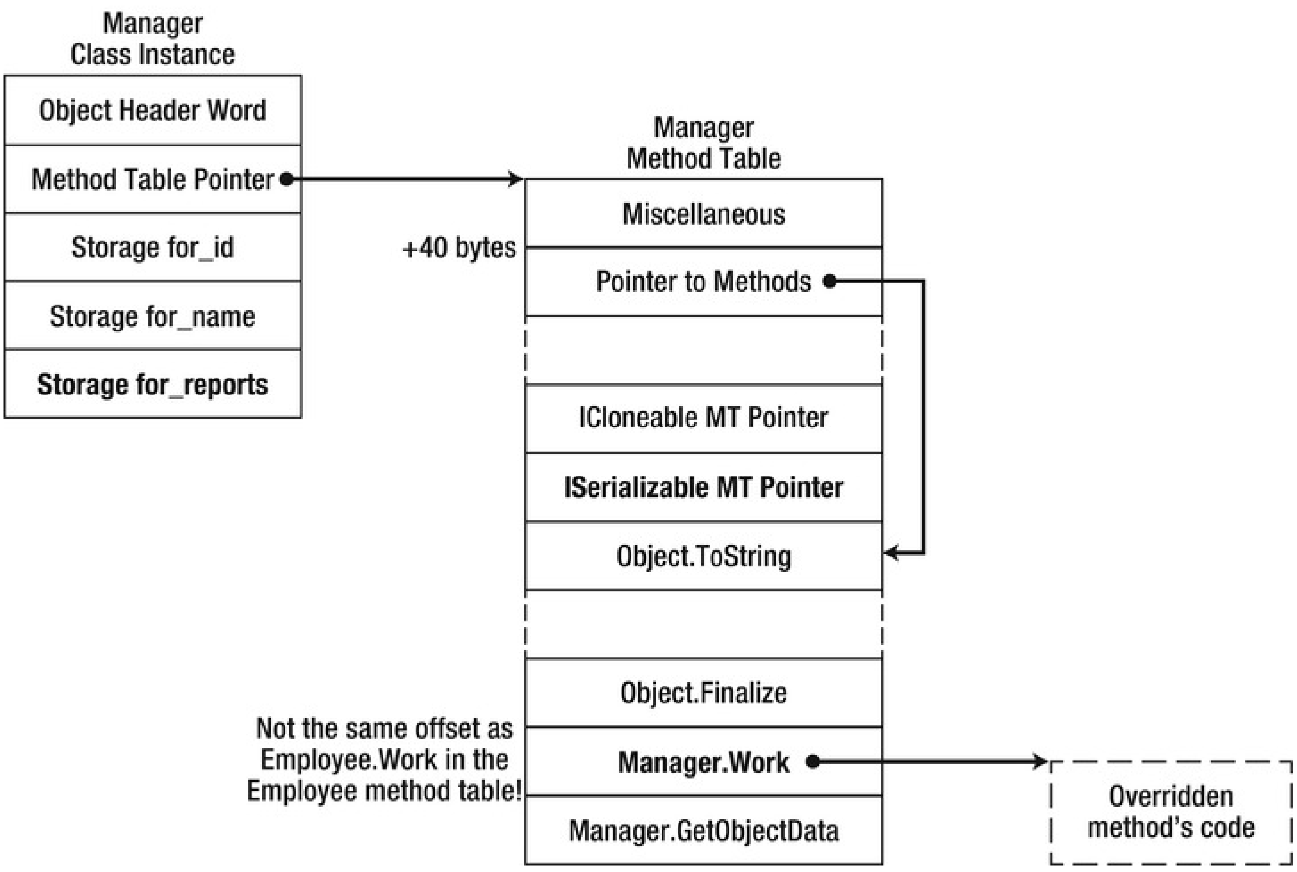

Figure 3-5. Method table layout for the Manager method table. This method table contains an additional interface MT slot, which makes the "Pointer to Methods" offset larger

```
mov ecx, dword ptr [ebp-64]
mov eax, dword ptr [ecx]
mov eax, dword ptr [ecx+40] ;this accommodates for the Work method having a different
call dword ptr [eax+16] ;absolute offset from the beginning of the MT
```

Note The object layout in which an overridden method's offset from the beginning of the method table is not guaranteed to be the same in derived classes is new in CLR 4.0. Prior to CLR 4.0, the list of interfaces implemented by the type was stored at the end of the method table, after the code addresses; this meant that the offset of the `Object.Equals` address (and the rest of the code addresses) was constant in all derived classes. In turn, this meant that the virtual method dispatch sequence consisted of only three instructions instead of four (the third instruction in the sequence above was not necessary). Older articles and books may still reference the previous invocation sequence and object layout, serving as an additional demonstration for how internal CLR details can change between versions without any notice.

Dispatching Non-Virtual Methods

We can use a similar dispatch sequence to call non-virtual methods as well. However, for non-virtual methods, there is no need to use the method table for method dispatch: the code address of the invoked method (or at least its pre-JIT stub) is known when the JIT compiles the method dispatch. For example, if the stack location `EBP-64` contains the address of an `Employee` object, as before, then the following instruction sequence will call the `TakeVacation` method with the parameter 5:

```
mov edx, 5 ;parameter passing through register – custom calling convention
mov ecx, dword ptr [ebp-64] ;still required because ECX contains 'this' by convention
call dword ptr [0x004a1260]
```

It is still required to load the object's address into the `ECX` register – all instance methods expect to receive in `ECX` the implicit `this` parameter. However, there's no longer any need to dereference the method table pointer and obtain the address from the method table. The JIT compiler still needs to be able to update the call site after performing the call; this is obtained by performing an indirect call through a memory location (`0x004a1260` in this example) that initially points to the pre-JIT stub and is updated by the JIT compiler as soon as the method is compiled.

Unfortunately, the method dispatch sequence above suffers from a significant problem. It allows method calls on null object references to be dispatched successfully and possibly remain undetected until the instance method attempts to access an instance field or a virtual method, which would cause an access violation. In fact, this is the behavior for C++ instance method calls – the following code would execute unharmed in most C++ environments, but would certainly make C# developers shift uneasily in their chairs:

```
class Employee {
public: void Work() { } //empty non-virtual method
};
Employee* pEmployee = NULL;
pEmployee->Work(); //runs to completion
```

If you inspect the actual sequence used by the JIT compiler to invoke non-virtual instance methods, it would contain an additional instruction:

```
mov edx, 5 ;parameter passing through register – custom calling convention
mov ecx, dword ptr [ebp-64] ;still required because ECX contains 'this' by convention
cmp ecx, dword ptr [ecx]
call dword ptr [0x004a1260]
```

Recall that the `CMP` instruction subtracts its second operand from the first and sets CPU flags according to the result of the operation. The code above does not use the comparison result stored in the CPU flags, so how would the `CMP` instruction help prevent calling a method using a null object reference? Well, the `CMP` instruction attempts to access the memory address in the `ECX` register, which contains the object reference. If the object reference is null, this memory access would fail with an access violation, because accessing the address 0 is always illegal in Windows processes. This access violation is converted by the CLR to a `NullReferenceException` which is thrown at the invocation point; a much better choice than emitting a null check inside the method after it has already been called. Furthermore, the `CMP` instruction occupies only two bytes in memory, and has the advantage of being able to check for invalid addresses other than null.

Note There is no need for a similar `CMP` instruction when invoking virtual methods; the null check is implicit because the standard virtual method invocation flow accesses the method table pointer, which makes sure the object pointer is valid. Even for virtual method calls you might not always see the `CMP` instruction being emitted; in recent CLR versions, the JIT compiler is smart enough to avoid redundant checks. For example, if the program flow has just returned from a virtual method invocation on an object — which contains the null check implicitly — then the JIT compiler might not emit the `CMP` instruction.

The reason we are so concerned with the precise implementation details of invoking virtual methods versus non-virtual methods is not the additional memory access or extra instruction that might or might not be required. The primary optimization precluded by virtual methods is *method inlining*, which is critical for modern high-performance applications. Inlining is a fairly simple compiler trick that trades code size for speed, whereby method calls to small or simple methods are replaced by the method's body. For example, in the code below, it makes perfect sense to replace the call to the `Add` method by the single operation performed inside it:

```
int Add(int a, int b)
{
  return a + b;
}
int c = Add(10, 12);
//assume that c is used later in the code
```

The non-optimized call sequence will have almost 10 instructions: three to set up parameters and dispatch the method, two to set up the method frame, one to add the numbers together, two to tear down the method frame, and one to return from the method. The optimized call sequence will have only *one* instruction – can you guess which one? One option is the `ADD` instruction, but in fact another optimization called constant-folding can be used to calculate at compile-time the result of the addition operation, and assign to the variable `c` the constant value 22.

The performance difference between inlined and non-inlined method calls can be vast, especially when the methods are as simple as the one above. Properties, for instance, make a great candidate for inlining, and compiler-generated automatic properties even more so because they don't contain any logic other than directly accessing a field. However, virtual methods prevent inlining because inlining can occur only when the compiler knows at compile-time (in the case of the JIT compiler, at JIT-time) which method is about to be invoked. When the method to be invoked is determined at runtime by the type information embedded into the object, there is no way to generate correctly inlined code for a virtual method dispatch. If all methods were virtual by default, properties would have been virtual as well, and the accumulated cost from indirect method dispatches where inlining was otherwise possible would have been overwhelming.

You might be wondering about the effects of the `sealed` keyword on method dispatch, in light of how important inlining may be. For example, if the `Manager` class declares the `Work` method as sealed, invocations of `Work` on object references that have the `Manager` static type can proceed as a non-virtual instance method invocation:

```
public class Manager : Employee
{
  public override sealed void Work() ...
}
Manager manager = ...; //could be an instance of Manager, could be a derived type
manager.Work(); //direct dispatch should be possible!
```

Nonetheless, at the time of writing, the `sealed` keyword has no effect on method dispatch on all the CLR versions we tested, even though knowledge that a class or method is sealed can be used to effectively remove the need for virtual method dispatch.

Dispatching Static and Interface Methods

For completeness, there are two additional types of methods we need to consider: static methods and interface methods. Dispatching static methods is fairly easy: there is no need to load an object reference, and simply calling the method (or its pre-JIT stub) would suffice. Because the invocation does not proceed through a method table, the JIT compiler uses the same trick as for non-virtual instance method: the method dispatch is indirect through a special memory location which is updated after the method is JIT compiled.

Interface methods, however, are a wholly different matter. It might appear that dispatching an interface method is not different from dispatching a virtual instance method. Indeed, interfaces enable a form of polymorphism reminiscent of classical virtual methods. Unfortunately, there is no guarantee that the interface implementations of a particular interface across several classes end up in the same slots in the method table. Consider the following code, where two classes implement the `IComparable` interface:

```
class Manager : Employee, IComparable {
  public override void Work() ...
  public void TakeVacation(int days) ...
  public static void SetCompanyPolicy(...) ...
  public int CompareTo(object other) ...
}
class BigNumber : IComparable {
  public long Part1, Part2;
  public int CompareTo(object other) ...
}
```

Clearly, the method table layout for these classes will be very different, and the slot number where the `CompareTo` method ends up will be different as well. Complex object hierarchies and multiple interface implementations make it evident that an additional dispatch step is required to identify where in the method table the interface methods were placed.

In prior CLR versions, this information was stored in a global (AppDomain-level) table indexed by an interface ID, generated when the interface is first loaded. The method table had a special entry (at offset 12) pointing into the proper location in the global interface table, and the global interface table entries pointed back into the method table, to the sub-table within it where the interface method pointers were stored. This allowed multi-step method dispatch, along the following lines:

```
mov ecx, dword ptr [ebp-64] ; object reference
mov eax, dword ptr [ecx] ; method table pointer
mov eax, dword ptr [eax+12] ; interface map pointer
mov eax, dword ptr [eax+48] ; compile time offset for this interface in the map
call dword ptr [eax] ; first method at EAX, second method at EAX+4, etc.
```

This looks complicated – and expensive! There are four memory accesses required to fetch the code address of the interface implementation and dispatch it, and for some interfaces this may be too high a cost. This is why you will never see the sequence above used by the production JIT compiler, even without optimizations enabled. The JIT uses several tricks to effectively inline interface methods, at least for the common case.

Hot-path analysis — when the JIT detects that the same interface implementation is often used, it replaces the specific call site with optimized code that may even inline the commonly used interface implementation:

```
mov ecx, dword ptr [ebp-64]
cmp dword ptr [ecx], 00385670 ; expected method table pointer
jne 00a188c0 ; cold path, shown below in pseudo-code
jmp 00a19548 ; hot path, could be inlined body here
cold path:
if (--wrongGuessesRemaining < 0) { ;starts at 100
  back patch the call site to the code discussed below
} else {
  standard interface dispatch as discussed above
}
```

Frequency analysis — when the JIT detects that its choice of hot path is no longer accurate for a particular call site (across a series of several dispatches), it replaces the former hot path guess with the new hot path, and continues to alternate between them every time it gets the guess wrong:

```
start: if (obj->MTP == expectedMTP) {
  direct jump to expected implementation
} else {
  expectedMTP = obj->MTP;
  goto start;
}
```

For more details on interface method dispatch, consider reading Sasha Goldshtein's article "JIT Optimizations" (`http://www.codeproject.com/Articles/25801/JIT-Optimizations`) and Vance Morrison's blog post (`http://blogs.msdn.com/b/vancem/archive/2006/03/13/550529.aspx`). Interface method dispatch is a moving target and a ripe ground for optimization; future CLR versions might introduce further optimizations not discussed here.

Sync Blocks And The `lock` Keyword

The second header field embedded in each reference type instance is the object header word (or sync block index). Unlike the method table pointer, this field is used for a variety of purposes, including synchronization, GC book-keeping, finalization, and hash code storage. Several bits of this field determine exactly which information is stored in it at any instant in time.

The most complex purpose for which the object header word is used is synchronization using the CLR monitor mechanism, exposed to C# through the `lock` keyword. The gist is as follows: several threads may attempt to enter a region of code protected by the `lock` statement, but only one thread at a time may enter the region, achieving mutual exclusion:

```
class Counter
{
  private int _i;
  private object _syncObject = new object();
  public int Increment()
  {
   lock (_syncObject)
   {
     return ++_i; //only one thread at a time can execute this statement
```

```
    }
  }
}
```

The `lock` keyword, however, is merely syntactic sugar that wraps the following construct, using the `Monitor.Enter` and `Monitor.Exit` methods:

```
class Counter
{
  private int _i;
  private object _syncObject = new object();
  public int Increment()
  {
   bool acquired = false;
   try
   {
     Monitor.Enter(_syncObject, ref acquired);
     return ++_i;
   }
   finally
   {
     if (acquired) Monitor.Exit(_syncObject);
   }
  }
}
```

To ensure this mutual exclusion, a synchronization mechanism can be associated with every object. Because it is expensive to create a synchronization mechanism for every object on the outset, this association occurs lazily, when the object is used for synchronization for the first time. When required, the CLR allocates a structure called a *sync block* from a global array called the *sync block table.* The sync block contains a backwards reference to its owning object (although this is a weak reference that does not prevent the object from being collected), and, among other things, a synchronization mechanism called monitor, which is implemented internally using a Win32 event. The numeric index of the allocated sync block is stored in the object's header word. Subsequent attempts to use the object for synchronization recognize the existing sync block index and use the associated monitor object for synchronization.

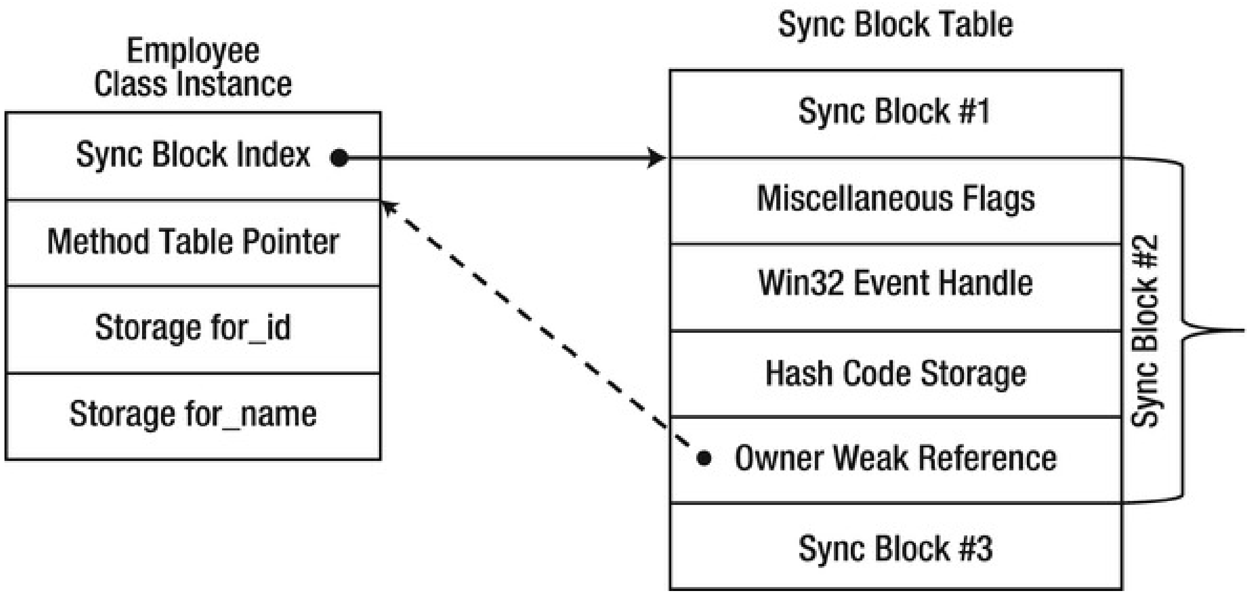

Figure 3-6 *. A sync block associated with an object instance. The sync block index fields stores only the index into the sync block table, allowing the CLR to resize and move the table in memory without modifying the sync block index*

After a sync block has been unused for a long period of time, the garbage collector reclaims it and detaches its owning object from it, setting the sync block index to an invalid index. Following this reclamation, the sync block can be associated with another object, which conserves expensive operating system resources required for synchronization mechanisms.

The `!SyncBlk` SOS command can be used to inspect sync blocks that are currently contended, i.e., sync blocks which are owned by a thread and waited for by another thread (possibly more than one waiter). As of CLR 2.0, there is an optimization that may lazily create a sync block only when there is contention for it. While there is no sync block, the CLR may use a *thin lock* to manage synchronization state. Below we explore some examples of this.

First, let's take a look at the object header word of an object that hasn't been used for synchronization yet, but whose hash code has already been accessed (later in this chapter we will discuss hash code storage in reference types). In the following example, `EAX` points to an `Employee` object whose hash code is 46104728:

```
0:000> dd eax-4 L2
023d438c 0ebf8098 002a3860
0:000> ? 0n46104728
Evaluate expression: 46104728 = 02bf8098
0:000> .formats 0ebf8098
```

```
Evaluate expression:
  Hex: 0ebf8098
  Binary: 00001110 10111111 10000000 10011000
0:000> .formats 02bf8098
Evaluate expression:
  Hex: 02bf8098
  Binary: 00000010 10111111 10000000 10011000
```

There is no sync block index here; only the hash code and two bits set to 1, one of them probably indicating that the object header word now stores the hash code. Next, we issue a `Monitor.Enter` call for the object from one thread to lock it, and inspect the object header word:

```
0:004> dd 02444390-4 L2
0244438c 08000001 00173868
0:000> .formats 08000001
Evaluate expression:
  Hex: 08000001
  Binary: 00001000 00000000 00000000 00000001
0:004> !syncblk
Index   SyncBlock       MonitorHeld     Recursion       Owning  Thread  Info    SyncBlock       Owner
   1    0097db4c 3 1    0092c698 1790 0 02444390        Employee
```

The object was assigned the sync block #1, which is evident from the `!SyncBlk` command output (for more information about the columns in the command's output, consult the SOS documentation). When another thread attempts to enter the `lock` statement with the same object, it enters a standard Win32 wait (albeit with message pumping if it's a GUI thread). Below is the bottom of the stack of a thread waiting for a monitor:

```
0:004> kb
ChildEBP RetAddr Args to Child
04c0f404 75120bdd 00000001 04c0f454 00000001 ntdll!NtWaitForMultipleObjects+0x15
04c0f4a0 76c61a2c 04c0f454 04c0f4c8 00000000 KERNELBASE!WaitForMultipleObjectsEx+0x100
04c0f4e8 670f5579 00000001 7efde000 00000000 KERNEL32!WaitForMultipleObjectsExImplementation+0xe0
04c0f538 670f52b3 00000000 ffffffff 00000001 clr!WaitForMultipleObjectsEx_SO_TOLERANT+0x3c
04c0f5cc 670f53a5 00000001 0097db60 00000000 clr!Thread::DoAppropriateWaitWorker+0x22f
04c0f638 670f544b 00000001 0097db60 00000000 clr!Thread::DoAppropriateWait+0x65
04c0f684 66f5c28a ffffffff 00000001 00000000 clr!CLREventBase::WaitEx+0x128
04c0f698 670fd055 ffffffff 00000001 00000000 clr!CLREventBase::Wait+0x1a
04c0f724 670fd154 00939428 ffffffff f2e05698 clr!AwareLock::EnterEpilogHelper+0xac
04c0f764 670fd24f 00939428 00939428 00050172 clr!AwareLock::EnterEpilog+0x48
04c0f77c 670fce93 f2e05674 04c0f8b4 0097db4c clr!AwareLock::Enter+0x4a
04c0f7ec 670fd580 ffffffff f2e05968 04c0f8b4 clr!AwareLock::Contention+0x221
04c0f894 002e0259 02444390 00000000 00000000 clr!JITutil_MonReliableContention+0x8a
```

The synchronization object used is 25c, which is a handle to an event:

```
0:004> dd 04c0f454 L1
04c0f454 0000025c
0:004> !handle 25c f
Handle 25c
 Type Event
 Attributes 0
 GrantedAccess  0x1f0003:
   Delete,ReadControl,WriteDac,WriteOwner,Synch
   QueryState,ModifyState
 HandleCount 2
 PointerCount 4
 Name <none>
 Object Specific Information
   Event Type Auto Reset
   Event is Waiting
```

And finally, if we inspect the raw sync block memory assigned to this object, the hash code and synchronization mechanism handle are clearly visible:

```
0:004> dd 0097db4c
0097db4c 00000003 00000001 0092c698 00000001
0097db5c 80000001 0000025c 0000000d 00000000
0097db6c 00000000 00000000 00000000 02bf8098
0097db7c 00000000 00000003 00000000 00000001
```

A final subtlety worth mentioning is that in the previous example, we forced the subsequent creation of a sync block by calling `GetHashCode` before locking the object. As of CLR 2.0, there is a special optimization aimed to conserve time and memory that does not create a sync block if the object has not been associated with a sync block before. Instead, the CLR uses a mechanism called *thin lock*. When an object is locked for the first time and there is no contention yet (i.e., no other thread has attempted to lock the object), the CLR stores in the object header word the managed thread ID of the object's current owning thread. For example, here is the object header word of an object locked by the application's main thread before there is any contention for the lock:

```
0:004> dd 02384390-4
0238438c 00000001 00423870 00000000 00000000
```

Here, the thread with managed thread ID 1 is the application's main thread, as is evident from the output of the `!Threads` command:

```
0:004> !Threads
ThreadCount: 2
UnstartedThread: 0
BackgroundThread: 1
PendingThread: 0
DeadThread: 0
Hosted Runtime: no
   Lock
   ID OSID ThreadOBJ State GC Mode GC Alloc Context Domain Count Apt Exception
  0 1 12f0 0033ce80 2a020 Preemptive 02385114:00000000 00334850 2 MTA
  2 2 23bc 00348eb8 2b220 Preemptive 00000000:00000000 00334850 0 MTA (Finalizer)
```

Thin locks are also reported by the SOS !DumpObj command, which indicates the owner thread for an object whose header contains a thin lock. Similarly, the `!DumpHeap -thinlock` command can output all the thin locks currently present in the managed heap:

```
0:004> !dumpheap -thinlock
 Address MT Size
02384390 00423870 12 ThinLock owner 1 (0033ce80) Recursive 0
02384758 5b70f98c 16 ThinLock owner 1 (0033ce80) Recursive 0
Found 2 objects.
0:004> !DumpObj 02384390
Name: Employee
MethodTable: 00423870
EEClass: 004213d4
Size: 12(0xc) bytes
File: D:\Development\...\App.exe
Fields:
   MT Field Offset Type VT Attr Value Name
00423970 4000001 4 CompanyPolicy 0 static 00000000 _policy
ThinLock owner 1 (0033ce80), Recursive 0
```

When another thread attempts to lock the object, it will spin for a short time waiting for the thin lock to be released (i.e., the owner thread information to disappear from the object header word). If, after a certain time threshold, the lock is not released, it is converted to a sync block, the sync block index is stored within the object header word, and from that point on, the threads block on the Win32 synchronization mechanism as usual.