CHAPTER 4

Type Fundamentals

In this chapter:

In this chapter, I will introduce information that is fundamental to working with types and the common language runtime (CLR). In particular, I'll discuss the minimum set of behaviors that you can expect every type to have. I'll also describe type safety, namespaces, assemblies, and the various ways you can cast objects from one type to another. Finally, I'll conclude this chapter with an explanation of how types, objects, thread stacks, and the managed heap all relate to one another at run time.

All Types Are Derived from System.Object

The runtime requires every type to ultimately be derived from the System.Object type. This means that the following two type definitions are identical.

```
// Implicitly derived from Object              // Explicitly derived from Object
class Employee {                               class Employee : System.Object {
   ...                                            ...
}                                     }
```

Because all types are ultimately derived from System.Object, you are guaranteed that every object of every type has a minimum set of methods. Specifically, the System.Object class offers the public instance methods listed in Table 4-1.

In addition, types that derive from System.Object have access to the protected methods listed in Table 4-2.

TABLE 4-1 Public Methods of System.Object

Public Method	Description
Equals	Returns true if two objects have the same value. For more information about this method, see the "Object Equality and Identity" section in Chapter 5, "Primitive, Reference, and Value Types."
GetHashCode	Returns a hash code for this object's value. A type should override this method if its objects are to be used as a key in a hash table collection, like Dictionary. The method should provide a good distribution for its objects. It is unfortunate that this method is defined in Object because most types are never used as keys in a hash table; this method should have been defined in an interface. For more information about this method, see the "Object Hash Codes" section in Chapter 5.
ToString	The default implementation returns the full name of the type (this.GetType().FullName). However, it is common to override this method so that it returns a String object containing a representation of the object's state. For example, the core types, such as Boolean and Int32, override this method to return a string representation of their values. It is also common to override this method for debugging purposes; you can call it and get a string showing the values of the object's fields. In fact, Microsoft Visual Studio's debugger calls this function automatically to show you a string representation of an object. Note that ToString is expected to be aware of the CultureInfo associated with the calling thread. Chapter 14, "Chars, Strings, and Working with Text," discusses ToString in greater detail.
GetType	Returns an instance of a Type-derived object that identifies the type of the object used to call GetType. The returned Type object can be used with the reflection classes to obtain metadata information about the object's type. Reflection is discussed in Chapter 23, "Assembly Loading and Reflection." The GetType method is nonvirtual, which prevents a class from overriding this method and lying about its type, violating type safety.

TABLE 4-2 Protected Methods of System.Object

Protected Method	Description
MemberwiseClone	This nonvirtual method creates a new instance of the type and sets the new object's instance fields to be identical to the this object's instance fields. A reference to the new instance is returned.
Finalize	This virtual method is called when the garbage collector determines that the object is garbage and before the memory for the object is reclaimed. Types that require cleanup when collected should override this method. I'll talk about this important method in much more detail in Chapter 21, "The Managed Heap and Garbage Collection."

The CLR requires all objects to be created using the new operator. The following line shows how to create an Employee object.

```
Employee e = new Employee("ConstructorParam1");
```

Here's what the new operator does:

1. It calculates the number of bytes required by all instance fields defined in the type and all of its base types up to and including System.Object (which defines no instance fields of its own). Every object on the heap requires some additional members—called the type object pointer and the sync block index—used by the CLR to manage the object. The bytes for these additional members are added to the size of the object.

2. It allocates memory for the object by allocating the number of bytes required for the specified type from the managed heap; all of these bytes are then set to zero (0).
3. It initializes the object's type object pointer and sync block index members.
4. The type's instance constructor is called, passing it any arguments (the string "ConstructorParam1" in the preceding example) specified in the call to new. Most compilers automatically emit code in a constructor to call a base class's constructor. Each constructor is responsible for initializing the instance fields defined by the type whose constructor is being called. Eventually, System.Object's constructor is called, and this constructor method does nothing but return.

After new has performed all of these operations, it returns a reference (or pointer) to the newly created object. In the preceding code example, this reference is saved in the variable e, which is of type Employee.

By the way, the new operator has no complementary delete operator; that is, there is no way to explicitly free the memory allocated for an object. The CLR uses a garbage-collected environment (described in Chapter 21) that automatically detects when objects are no longer being used or accessed and frees the object's memory automatically.

Casting Between Types

One of the most important features of the CLR is type safety. At run time, the CLR always knows what type an object is. You can always discover an object's exact type by calling the GetType method. Because this method is nonvirtual, it is impossible for a type to spoof another type. For example, the Employee type can't override the GetType method and have it return a type of SuperHero.

Developers frequently find it necessary to cast an object to various types. The CLR allows you to cast an object to its type or to any of its base types. Your choice of programming language dictates how to expose casting operations to the developer. For example, C# doesn't require any special syntax to cast an object to any of its base types, because casts to base types are considered safe implicit conversions. However, C# does require the developer to explicitly cast an object to any of its derived types because such a cast could fail at run time. The following code demonstrates casting to base and derived types.

```
// This type is implicitly derived from System.Object.
internal class Employee {
   ...
}


public sealed class Program {
   public static void Main() {
      // No cast needed since new returns an Employee object
      // and Object is a base type of Employee.
      Object o = new Employee();
      // Cast required since Employee is derived from Object.
      // Other languages (such as Visual Basic) might not require
      // this cast to compile.
      Employee e = (Employee) o;
   }
}
```

This example shows what is necessary for your compiler to compile your code. Now I'll explain what happens at run time. At run time, the CLR checks casting operations to ensure that casts are always to the object's actual type or any of its base types. For example, the following code will compile, but at run time, an `InvalidCastException` will be thrown.

```
internal class Employee {
   ...
}
internal class Manager : Employee {
   ...
}


public sealed class Program {
   public static void Main() {
      // Construct a Manager object and pass it to PromoteEmployee.
      // A Manager IS-A Object: PromoteEmployee runs OK.
      Manager m = new Manager();
      PromoteEmployee(m);

      // Construct a DateTime object and pass it to PromoteEmployee.
      // A DateTime is NOT derived from Employee. PromoteEmployee
      // throws a System.InvalidCastException exception.
      DateTime newYears = new DateTime(2013, 1, 1);
      PromoteEmployee(newYears);
   }


   public static void PromoteEmployee(Object o) {
      // At this point, the compiler doesn't know exactly what
      // type of object o refers to. So the compiler allows the
      // code to compile. However, at run time, the CLR does know
      // what type o refers to (each time the cast is performed) and
      // it checks whether the object's type is Employee or any type
      // that is derived from Employee.
      Employee e = (Employee) o;
      ...
   }
}
```

In the Main method, a Manager object is constructed and passed to PromoteEmployee. This code compiles and executes because Manager is ultimately derived from Object, which is what Promote-Employee expects. Once inside PromoteEmployee, the CLR confirms that o refers to an object that is either an Employee or a type that is derived from Employee. Because Manager is derived from Employee, the CLR performs the cast and allows PromoteEmployee to continue executing.

After PromoteEmployee returns, Main constructs a DateTime object and passes it to Promote-Employee. Again, DateTime is derived from Object, and the compiler compiles the code that calls PromoteEmployee with no problem. However, inside PromoteEmployee, the CLR checks the cast and detects that o refers to a DateTime object and is therefore not an Employee or any type derived from Employee. At this point, the CLR can't allow the cast and throws a System.InvalidCastException.

If the CLR allowed the cast, there would be no type safety, and the results would be unpredictable, including the possibility of application crashes and security breaches caused by the ability of types to easily spoof other types. Type spoofing is the cause of many security breaches and compromises an application's stability and robustness. Type safety is therefore an extremely important part of the CLR.

By the way, the proper way to declare the PromoteEmployee method would be to specify an Employee type instead of an Object type as its parameter so that the compiler produces a compile-time error, saving the developer from waiting until a runtime exception occurs to discover a problem. I used Object so that I could demonstrate how the C# compiler and the CLR deal with casting and type-safety.

Casting with the C# is and as Operators

Another way to cast in the C# language is to use the is operator. The is operator checks whether an object is compatible with a given type, and the result of the evaluation is a Boolean: true or false. The is operator will never throw an exception. The following code demonstrates.

```
Object o = new Object();
Boolean b1 = (o is Object);   // b1 is true.
Boolean b2 = (o is Employee); // b2 is false.
```

If the object reference is null, the is operator always returns false because there is no object available to check its type.

The is operator is typically used as follows.

```
if (o is Employee) {
   Employee e = (Employee) o;
   // Use e within the remainder of the 'if' statement.
}
```

In this code, the CLR is actually checking the object's type twice: The `is` operator first checks to see if o is compatible with the `Employee` type. If it is, inside the `if` statement, the CLR again verifies that o refers to an `Employee` when performing the cast. The CLR's type checking improves security, but it certainly comes at a performance cost, because the CLR must determine the actual type of the object referred to by the variable (o), and then the CLR must walk the inheritance hierarchy, checking each base type against the specified type (`Employee`). Because this programming paradigm is quite common, C# offers a way to simplify this code and improve its performance by providing an `as` operator.

```
Employee e = o as Employee;
if (e != null) {
   // Use e within the 'if' statement.
}
```

In this code, the CLR checks if o is compatible with the `Employee` type, and if it is, `as` returns a non-`null` reference to the same object. If o is not compatible with the `Employee` type, the as operator returns `null`. Notice that the `as` operator causes the CLR to verify an object's type just once. The `if` statement simply checks whether `e` is `null`; this check can be performed faster than verifying an object's type.

The `as` operator works just as casting does except that the `as` operator will never throw an exception. Instead, if the object can't be cast, the result is `null`. You'll want to check to see whether the resulting reference is `null`, or attempting to use the resulting reference will cause a `System.NullReferenceException` to be thrown. The following code demonstrates.

```
Object o = new Object();    // Creates a new Object object
Employee e = o as Employee; // Casts o to an Employee
// The cast above fails: no exception is thrown, but e is set to null.

e.ToString();  // Accessing e throws a NullReferenceException.
```

To make sure you understand everything just presented, take the following quiz. Assume that these two class definitions exist.

```
internal class B {     // Base class
}

internal class D : B { // Derived class
}
```

Now examine the lines of C# code in Table 4-3. For each line, decide whether the line would compile and execute successfully (marked OK in Table 4-3), cause a compile-time error (CTE), or cause a run-time error (RTE).

TABLE 4-3 Type-Safety Quiz

Statement	**OK**	**CTE**	**RTE**
`Object o1 = new Object();`	✓		
`Object o2 = new B();`	✓		
`Object o3 = new D();`	✓		
`Object o4 = o3;`	✓		
`B b1 = new B();`	✓		
`B b2 = new D();`	✓		
`D d1 = new D();`	✓		
`B b3 = new Object();`		✓	
`D d2 = new Object();`		✓	
`B b4 = d1;`	✓		
`D d3 = b2;`		✓	
`D d4 = (D) d1;`	✓		
`D d5 = (D) b2;`	✓		
`D d6 = (D) b1;`			✓
`B b5 = (B) o1;`			✓
`B b6 = (D) b2;`	✓		

Note C# allows a type to define conversion operator methods, as discussed in the "Conversion Operator Methods" section of Chapter 8, "Methods." These methods are invoked only when using a cast expression; they are never invoked when using C#'s `as` or `is` operator.

Namespaces and Assemblies

Namespaces allow for the logical grouping of related types, and developers typically use them to make it easier to locate a particular type. For example, the `System.Text` namespace defines a bunch of types for performing string manipulations, and the `System.IO` namespace defines a bunch of types for performing I/O operations. Here's some code that constructs a `System.IO.FileStream` object and a `System.Text.StringBuilder` object.

```
public sealed class Program {
   public static void Main() {
      System.IO.FileStream fs = new System.IO.FileStream(...);
      System.Text.StringBuilder sb = new System.Text.StringBuilder();
   }
}
```

As you can see, the code is pretty verbose; it would be nice if there were some shorthand way to refer to the `FileStream` and `StringBuilder` types to reduce typing. Fortunately, many compilers do offer mechanisms to reduce programmer typing. The C# compiler provides this mechanism via the `using` directive. The following code is identical to the previous example.

```
using System.IO;    // Try prepending "System.IO."
using System.Text;  // Try prepending "System.Text."

public sealed class Program {
   public static void Main() {
      FileStream fs = new FileStream(...);
      StringBuilder sb = new StringBuilder();
   }
}
```

To the compiler, a namespace is simply an easy way of making a type's name longer and more likely to be unique by preceding the name with some symbols separated by dots. So the compiler interprets the reference to `FileStream` in this example to mean `System.IO.FileStream`. Similarly, the compiler interprets the reference to `StringBuilder` to mean `System.Text.StringBuilder`.

Using the C# `using` directive is entirely optional; you're always welcome to type out the fully qualified name of a type if you prefer. The C# `using` directive instructs the compiler to try prepending different prefixes to a type name until a match is found.

Important The CLR doesn't know anything about namespaces. When you access a type, the CLR needs to know the full name of the type (which can be a really long name containing periods) and which assembly contains the definition of the type so that the runtime can load the proper assembly, find the type, and manipulate it.

In the previous code example, the compiler needs to ensure that every type referenced exists and that my code is using that type in the correct way: calling methods that exist, passing the right number of arguments to these methods, ensuring that the arguments are the right type, using the method's return value correctly, and so on. If the compiler can't find a type with the specified name in the source files or in any referenced assemblies, it prepends `System.IO.` to the type name and checks if the generated name matches an existing type. If the compiler still can't find a match, it prepends `System.Text.` to the type's name. The two `using` directives shown earlier allow me to simply type `FileStream` and `StringBuilder` in my code—the compiler automatically expands the references to `System.IO.FileStream` and `System.Text.StringBuilder`. I'm sure you can easily imagine how much typing this saves, as well as how much cleaner your code is to read.

When checking for a type's definition, the compiler must be told which assemblies to examine by using the `/reference` compiler switch as discussed in Chapter 2, "Building, Packaging, Deploying, and Administering Applications and Types," and Chapter 3, "Shared Assemblies and Strongly Named Assemblies." The compiler will scan all of the referenced assemblies looking for the type's definition.

After the compiler finds the proper assembly, the assembly information and the type information is emitted into the resulting managed module's metadata. To get the assembly information, you must pass the assembly that defines any referenced types to the compiler. The C# compiler, by default, automatically looks in the MSCorLib.dll assembly even if you don't explicitly tell it to. The MSCorLib.dll assembly contains the definitions of all of the core Framework Class Library (FCL) types, such as `Object`, `Int32`, `String`, and so on.

As you might imagine, there are some potential problems with the way that compilers treat namespaces: it's possible to have two (or more) types with the same name in different namespaces. Microsoft strongly recommends that you define unique names for types. However, in some cases, it's simply not possible. The runtime encourages the reuse of components. Your application might take advantage of a component that Microsoft created and another component that Wintellect created. These two companies might both offer a type called `Widget`—Microsoft's `Widget` does one thing, and Wintellect's `Widget` does something entirely different. In this scenario, you had no control over the naming of the types, so you can differentiate between the two widgets by using their fully qualified names when referencing them. To reference Microsoft's `Widget`, you would use `Microsoft.Widget`, and to reference Wintellect's `Widget`, you would use `Wintellect.Widget`. In the following code, the reference to `Widget` is ambiguous, so the C# compiler generates the following message: `error CS0104: 'Widget'` is an ambiguous `reference between 'Microsoft.Widget' and 'Wintellect.Widget'`.

```
using Microsoft;  // Try prepending "Microsoft."
using Wintellect; // Try prepending "Wintellect."

public sealed class Program {
   public static void Main() {
      Widget w = new Widget();// An ambiguous reference
   }
}
```

To remove the ambiguity, you must explicitly tell the compiler which `Widget` you want to create.

```
using Microsoft;  // Try prepending "Microsoft."
using Wintellect; // Try prepending "Wintellect."

public sealed class Program {
   public static void Main() {
      Wintellect.Widget w = new Wintellect.Widget(); // Not ambiguous
   }
}
```

There's another form of the C# `using` directive that allows you to create an alias for a single type or namespace. This is handy if you have just a few types that you use from a namespace and don't want to pollute the global namespace with all of a namespace's types. The following code demonstrates another way to solve the ambiguity problem shown in the preceding code.

```
using Microsoft;  // Try prepending "Microsoft."
using Wintellect; // Try prepending "Wintellect."
```

```
// Define WintellectWidget symbol as an alias to Wintellect.Widget
using WintellectWidget = Wintellect.Widget;

public sealed class Program {
   public static void Main() {
      WintellectWidget w = new WintellectWidget(); // No error now
   }
}
```

These methods of disambiguating a type are useful, but in some scenarios, you need to go further. Imagine that the Australian Boomerang Company (ABC) and the Alaskan Boat Corporation (ABC) are each creating a type, called `BuyProduct`, which they intend to ship in their respective assemblies. It's likely that both companies would create a namespace called ABC that contains a type called `Buy-Product`. Anyone who tries to develop an application that needs to buy both boomerangs and boats would be in for some trouble unless the programming language provides a way to programmatically distinguish between the assemblies, not just between the namespaces. Fortunately, the C# compiler offers a feature called extern aliases that gives you a way to work around this rarely occurring problem. Extern aliases also give you a way to access a single type from two (or more) different versions of the same assembly. For more information about extern aliases, see the C# Language Specification.

In your library, when you're designing types that you expect third parties to use, you should define these types in a namespace so that compilers can easily disambiguate them. In fact, to reduce the likelihood of conflict, you should use your full company name (not an acronym or abbreviation) to be your top-level namespace name. Referring to the Microsoft .NET Framework SDK documentation, you can see that Microsoft uses a namespace of "Microsoft" for Microsoft-specific types. (See the `Microsoft.CSharp`, `Microsoft.VisualBasic`, and `Microsoft.Win32` namespaces as examples.)

Creating a namespace is simply a matter of writing a namespace declaration into your code as follows (in C#).

```
namespace CompanyName {
   public sealed class A {              // TypeDef: CompanyName.A
   }

   namespace X {
      public sealed class B { ... }     // TypeDef: CompanyName.X.B
   }
}
```

The comment on the right of the preceding class definitions indicates the real name of the type the compiler will emit into the type definition metadata table; this is the real name of the type from the CLR's perspective.

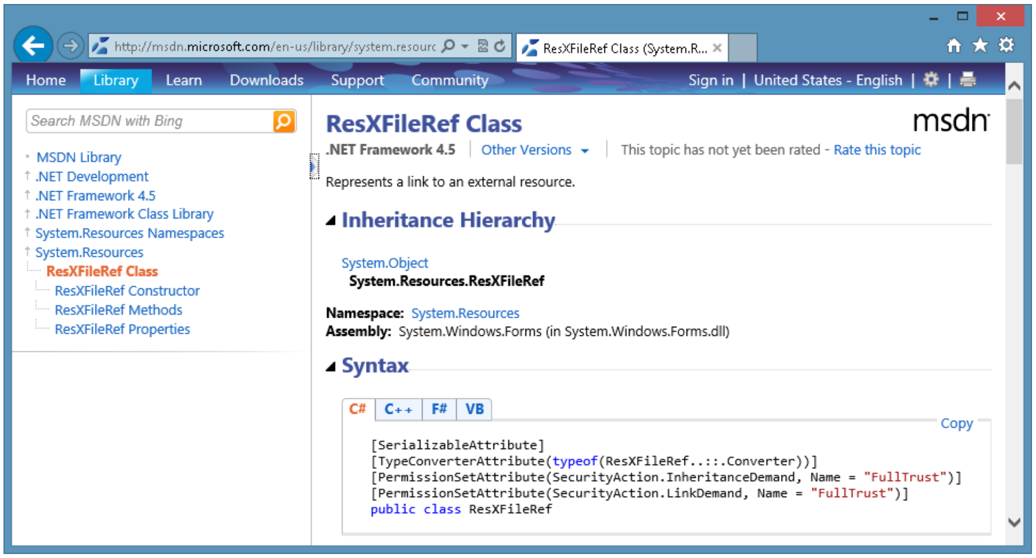
Some compilers don't support namespaces at all, and other compilers are free to define what "namespace" means to a particular language. In C#, the `namespace` directive simply tells the compiler to prefix each type name that appears in source code with the namespace name so that programmers can do less typing.

How Namespaces and Assemblies Relate

Be aware that a namespace and an assembly (the file that implements a type) aren't necessarily related. In particular, the various types belonging to a single namespace might be implemented in multiple assemblies. For example, the `System.IO.FileStream` type is implemented in the MSCorLib.dll assembly, and the `System.IO.FileSystemWatcher` type is implemented in the System.dll assembly.

A single assembly can contain types in different namespaces. For example, the `System.Int32` and `System.Text.StringBuilder` types are both in the MSCorLib.dll assembly.

When you look up a type in the .NET Framework SDK documentation, the documentation will clearly indicate the namespace that the type belongs to and also the assembly that the type is implemented in. In Figure 4-1, you can clearly see (right above the Syntax section) that the `ResXFileRef` type is part of the `System.Resources` namespace and that the type is implemented in the System.Windows.Forms.dll assembly. To compile code that references the `ResXFileRef` type, you'd add a `using System.Resources;` directive to your source code, and you'd use the `/r:System.Windows.Forms.dll` compiler switch.



ResXFileRef Class

.NET Framework 4.5 | Other Versions | This topic has not yet been rated - Rate this topic

Represents a link to an external resource.

Inheritance Hierarchy

System.Object
System.Resources.ResXFileRef

Namespace: System.Resources
Assembly: System.Windows.Forms (in System.Windows.Forms.dll)

Syntax

```
[SerializableAttribute]
[TypeConverterAttribute(typeof(ResXFileRef..::.Converter))]
[PermissionSetAttribute(SecurityAction.InheritanceDemand, Name = "FullTrust")]
[PermissionSetAttribute(SecurityAction.LinkDemand, Name = "FullTrust")]
public class ResXFileRef
```

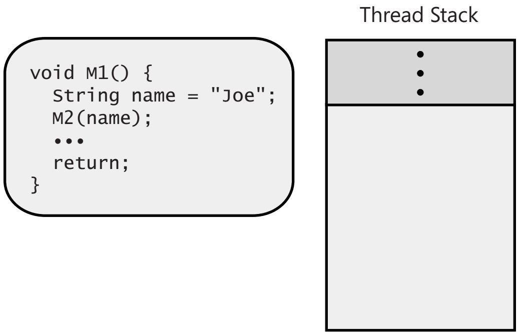
FIGURE 4-1 SDK documentation showing namespace and assembly information for a type.

How Things Relate at Run Time

In this section, I'm going to explain the relationship at run time between types, objects, a thread's stack, and the managed heap. Furthermore, I will also explain the difference between calling static methods, instance methods, and virtual methods. Let's start off with some fundamentals of computers.

What I'm about to describe is not specific to the CLR at all, but I'm going to describe it so that we have a working foundation, and then I'll modify the discussion to incorporate CLR-specific information.

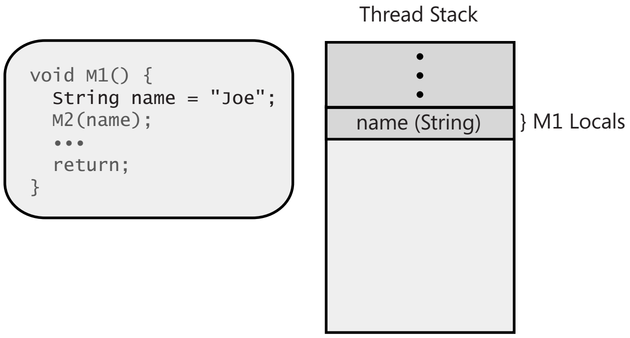
Figure 4-2 shows a single Windows process that has the CLR loaded into it. In this process there may be many threads. When a thread is created, it is allocated a 1-MB stack. This stack space is used for passing arguments to a method and for local variables defined within a method. In Figure 4-2, the memory for one thread's stack is shown (on the right). Stacks build from high-memory addresses to low-memory addresses. In the figure, this thread has been executing some code, and its stack has some data on it already (shown as the shaded area at the top of the stack). Now, imagine that the thread has executed some code that calls the `M1` method.



```
void M1() {
  String name = "Joe";
  M2(name);
  •••
  return;
}
```

FIGURE 4-2 A thread's stack with the M1 method about to be called.

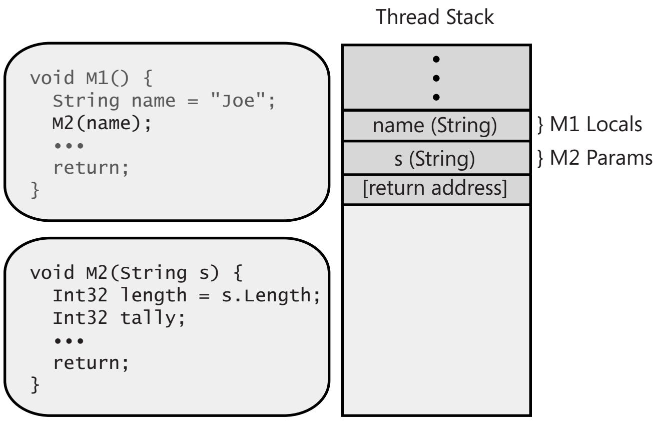
All but the simplest of methods contain some *prologue code*, which initializes a method before it can start doing its work. These methods also contain *epilogue code*, which cleans up a method after it has performed its work so that it can return to its caller. When the `M1` method starts to execute, its prologue code allocates memory for the local `name` variable from the thread's stack (see Figure 4-3).



```
void M1() {
  String name = "Joe";
  M2(name);
  •••
  return;
}
```

FIGURE 4-3 Allocating M1's local variable on the thread's stack.

Then, `M1` calls the `M2` method, passing in the `name` local variable as an argument. This causes the address in the `name` local variable to be pushed on the stack (see Figure 4-4). Inside the `M2` method, the stack location will be identified using the parameter variable named `s`. (Note that some architectures pass arguments via registers to improve performance, but this distinction is not important for this discussion.) Also, when a method is called, the address indicating where the called method should return to in the calling method is pushed on the stack (also shown in Figure 4-4).



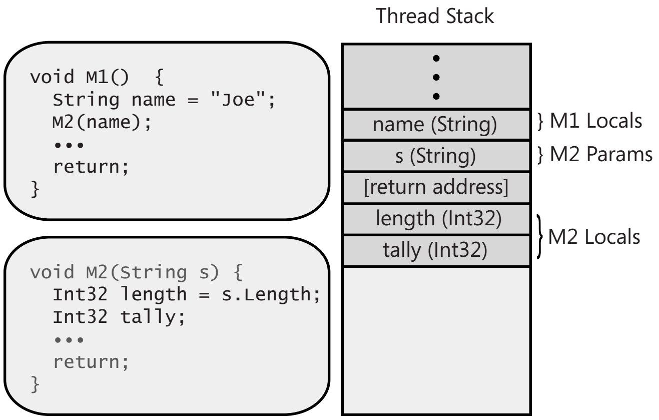
```
void M1() {
  String name = "Joe";
  M2(name);
  •••
  return;
}
```

```
void M2(String s) {
  Int32 length = s.Length;
  Int32 tally;
  •••
  return;
}
```

FIGURE 4-4 M1 pushes arguments and the return address on the thread's stack when calling M2.

When the M2 method starts to execute, its prologue code allocates memory for the local `length` and `tally` variables from the thread's stack (see Figure 4-5). Then the code inside method M2 executes. Eventually, M2 gets to its return statement, which causes the CPU's instruction pointer to be set to the return address in the stack, and M2's stack frame is unwound so that it looks the way it did in Figure 4-3. At this point, M1 is continuing to execute its code that immediately follows the call to M2, and its stack frame accurately reflects the state needed by M1.

Eventually, M1 will return back to its caller by setting the CPU's instruction pointer to be set to the return address (not shown on the figures, but it would be just above the `name` argument on the stack), and M1's stack frame is unwound so that it looks the way it did in Figure 4-2. At this point, the method that called M1 continues to execute its code that immediately follows the call to M1, and its stack frame accurately reflects the state needed by that method.



```
void M1()  {
  String name = "Joe";
  M2(name);
  •••
  return;
}
```

```
void M2(String s) {
  Int32 length = s.Length;
  Int32 tally;
  •••
  return;
}
```

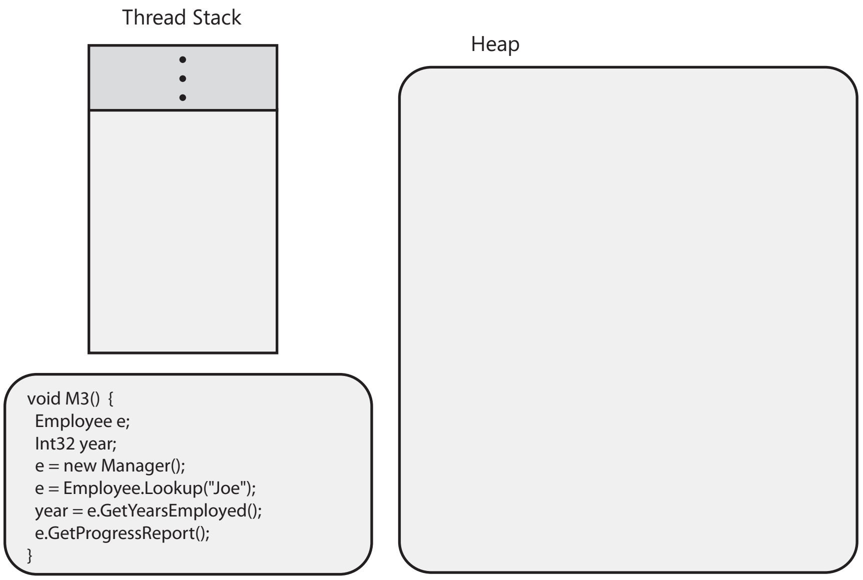
FIGURE 4-5 Allocating M2's local variables on the thread's stack.

Now, let's start gearing the discussion toward the CLR. Let's say that we have these two class definitions.

```
internal class Employee {
   public         Int32     GetYearsEmployed()   { ... }
   public virtual String    GetProgressReport()  { ... }
   public static  Employee  Lookup(String name)  { ... }
}

internal sealed class Manager : Employee {
   public override String   GetProgressReport()  { ... }
}
```

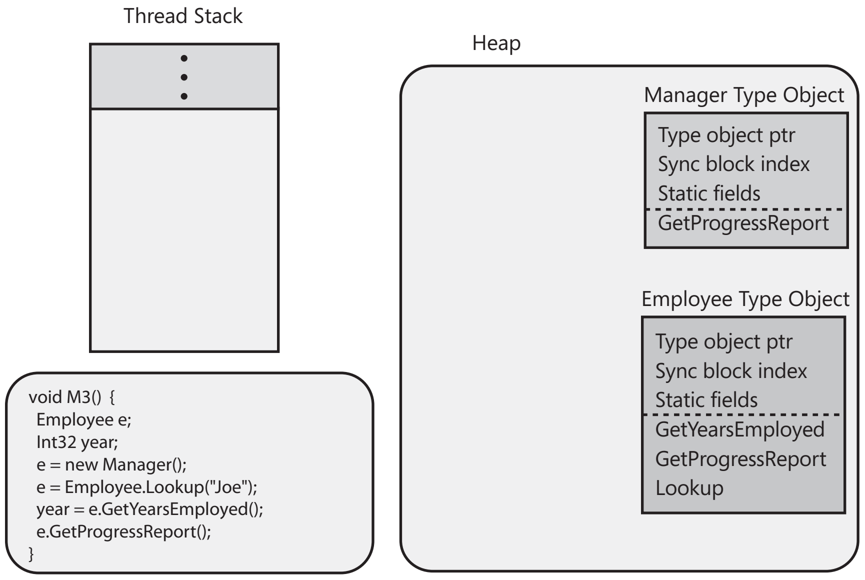
Our Windows process has started, the CLR is loaded into it, the managed heap is initialized, and a thread has been created (along with its 1 MB of stack space). This thread has already executed some code, and this code has decided to call the `M3` method. All of this is shown in Figure 4-6. The `M3` method contains code that demonstrates how the CLR works; this is not code that you would normally write, because it doesn't actually do anything useful.



```
void M3() {
  Employee e;
  Int32 year;
  e = new Manager();
  e = Employee.Lookup("Joe");
  year = e.GetYearsEmployed();
  e.GetProgressReport();
}
```

FIGURE 4-6 The CLR loaded in a process, its heap initialized, and a thread's stack with the `M3` method about to be called.

As the just-in-time (JIT) compiler converts `M3`'s Intermediate Language (IL) code into native CPU instructions, it notices all of the types that are referred to inside `M3: Employee`, `Int32`, `Manager`, and `String` (because of `"Joe"`). At this time, the CLR ensures that the assemblies that define these types are loaded. Then, using the assembly's metadata, the CLR extracts information about these types and creates some data structures to represent the types themselves. The data structures for the `Employee` and Manager type objects are shown in Figure 4-7. Because this thread already executed some code prior to calling M3, let's assume that the Int32 and String type objects have already been created (which is likely because these are commonly used types), and so I won't show them in the figure.

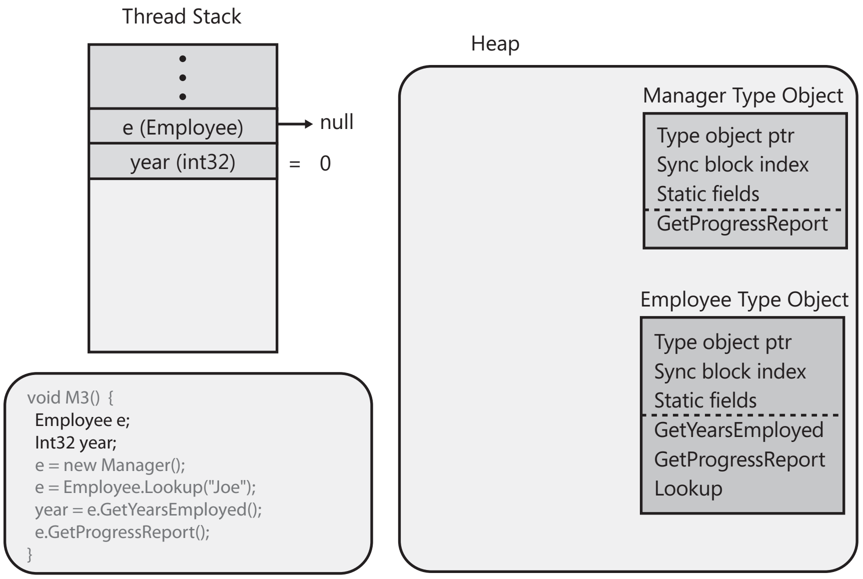


```
void M3() {
  Employee e;
  Int32 year;
  e = new Manager();
  e = Employee.Lookup("Joe");
  year = e.GetYearsEmployed();
  e.GetProgressReport();
}
```

FIGURE 4-7 The Employee and Manager type objects are created just as M3 is being called.

Let's take a moment to discuss these type objects. As discussed earlier in this chapter, all objects on the heap contain two overhead members: the type object pointer and the sync block index. As you can see, the Employee and Manager type objects have both of these members. When you define a type, you can define static data fields within it. The bytes that back these static data fields are allocated within the type objects themselves. Finally, inside each type object is a method table with one entry per method defined within the type. This is the method table that was discussed in Chapter 1, "The CLR's Execution Model." Because the Employee type defines three methods (GetYears-Employed, GetProgressReport, and Lookup), there are three entries in Employee's method table. Because the Manager type defines one method (an override of GetProgressReport), there is just one entry in Manager's method table.

Now, after the CLR has ensured that all of the type objects required by the method are created and the code for M3 has been compiled, the CLR allows the thread to execute M3's native code. When M3's prologue code executes, memory for the local variables must be allocated from the thread's stack, as shown in Figure 4-8. By the way, the CLR automatically initializes all local variables to null or 0 (zero) as part of the method's prologue code. However, the C# compiler issues a Use of unassigned local variable error message if you write code that attempts to read from a local variable that you have not explicitly initialized in your source code.

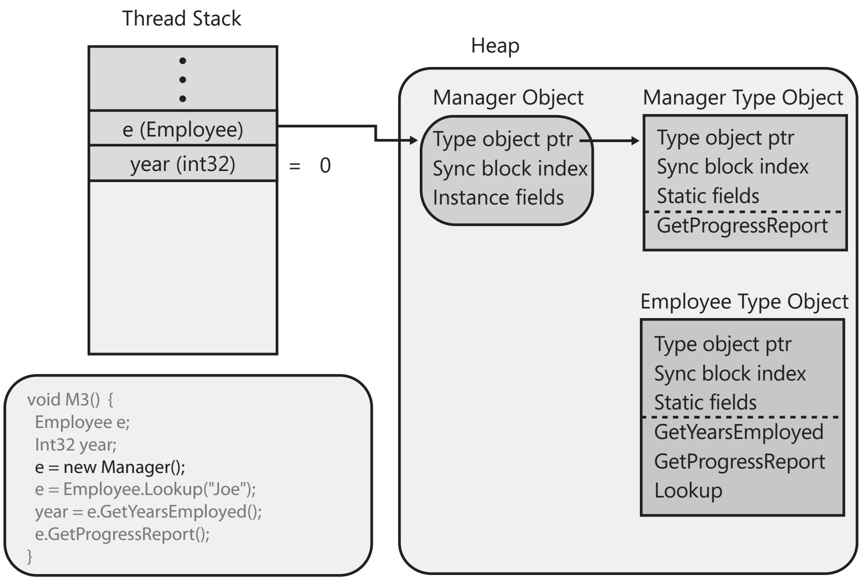


```
void M3() {
  Employee e;
  Int32 year;
  e = new Manager();
  e = Employee.Lookup("Joe");
  year = e.GetYearsEmployed();
  e.GetProgressReport();
}
```

FIGURE 4-8 Allocating `M3`'s local variables on the thread's stack.

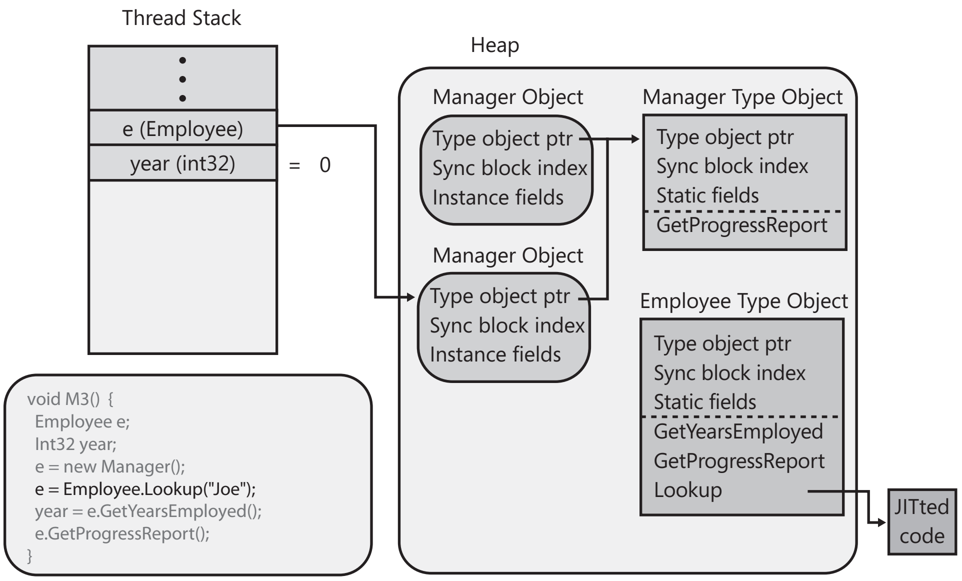
Then, `M3` executes its code to construct a `Manager` object. This causes an instance of the `Manager` type, a `Manager` object, to be created in the managed heap, as shown in Figure 4-9. As you can see, the `Manager` object—as do all objects—has a type object pointer and sync block index. This object also contains the bytes necessary to hold all of the instance data fields defined by the `Manager` type, as well as any instance fields defined by any base classes of the `Manager` type (in this case, `Employee` and `Object`). Whenever a new object is created on the heap, the CLR automatically initializes the internal type object pointer member to refer to the object's corresponding type object (in this case, the `Manager` type object). Furthermore, the CLR initializes the sync block index and sets all of the object's instance fields to `null` or `0` (zero) prior to calling the type's constructor, a method that will likely modify some of the instance data fields. The `new` operator returns the memory address of the `Manager` object, which is saved in the variable `e` (on the thread's stack).

The next line of code in `M3` calls `Employee`'s static `Lookup` method. When calling a static method, the JIT compiler locates the type object that corresponds to the type that defines the static method. Then, the JIT compiler locates the entry in the type object's method table that refers to the method being called, JITs the method (if necessary), and calls the JITted code. For our discussion, let's say that `Employee`'s `Lookup` method queries a database to find Joe. Let's also say that the database indicates that Joe is a manager at the company, and therefore, internally, the `Lookup` method constructs a new `Manager` object on the heap, initializes it for Joe, and returns the address of this object. The address is saved in the local variable `e`. The result of this operation is shown in Figure 4-10.



```
void M3() {
  Employee e;
  Int32 year;
  e = new Manager();
  e = Employee.Lookup("Joe");
  year = e.GetYearsEmployed();
  e.GetProgressReport();
}
```

FIGURE 4-9 Allocating and initializing a `Manager` object.

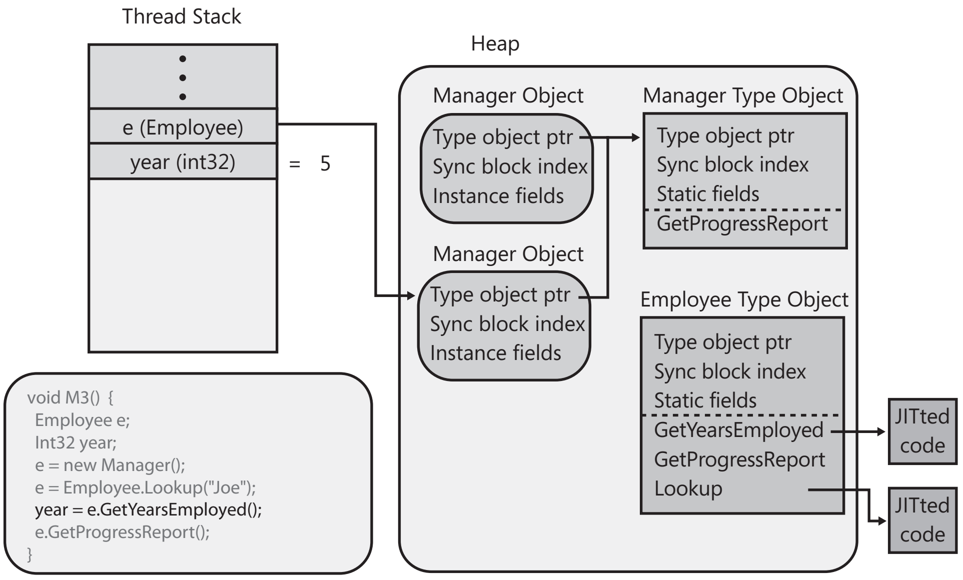


```
void M3() {
  Employee e;
  Int32 year;
  e = new Manager();
  e = Employee.Lookup("Joe");
  year = e.GetYearsEmployed();
  e.GetProgressReport();
}
```

FIGURE 4-10 Employee's static `Lookup` method allocates and initializes a `Manager` object for Joe.

Note that e no longer refers to the first Manager object that was created. In fact, because no variable refers to this object, it is a prime candidate for being garbage collected in the future, which will reclaim (free) the memory used by this object.

The next line of code in M3 calls Employee's nonvirtual instance GetYearsEmployed method. When calling a nonvirtual instance method, the JIT compiler locates the type object that corresponds to the type of the variable being used to make the call. In this case, the variable e is defined as an Employee. (If the Employee type didn't define the method being called, the JIT compiler walks down the class hierarchy toward Object looking for this method. It can do this because each type object has a field in it that refers to its base type; this information is not shown in the figures.) Then, the JIT compiler locates the entry in the type object's method table that refers to the method being called, JITs the method (if necessary), and then calls the JITted code. For our discussion, let's say that Employee's GetYearsEmployed method returns 5 because Joe has been employed at the company for five years. The integer is saved in the local variable year. The result of this operation is shown in Figure 4-11.

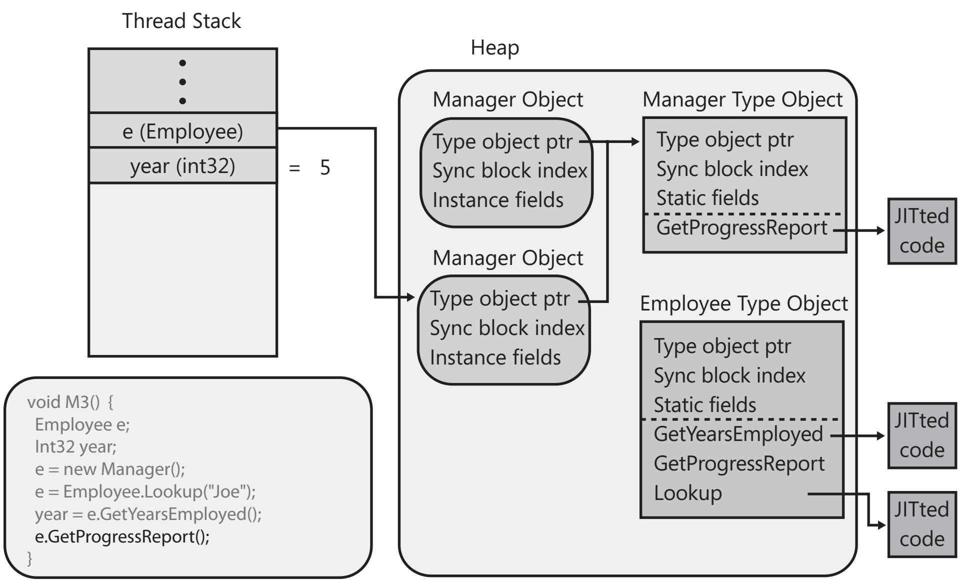


```
void M3() {
  Employee e;
  Int32 year;
  e = new Manager();
  e = Employee.Lookup("Joe");
  year = e.GetYearsEmployed();
  e.GetProgressReport();
}
```

FIGURE 4-11 Employee's nonvirtual instance GetYearsEmployed method is called, returning 5.

The next line of code in M3 calls Employee's virtual instance GetProgressReport method. When calling a virtual instance method, the JIT compiler produces some additional code in the method, which will be executed each time the method is invoked. This code will first look in the variable being used to make the call and then follow the address to the calling object. In this case, the variable e points to the Manager object representing "Joe." Then, the code will examine the object's internal type object pointer member; this member refers to the actual type of the object. The code then locates the entry in the type object's method table that refers to the method being called, JITs the method (if necessary), and calls the JITted code. For our discussion, Manager's GetProgressReport implementation is called because `e` refers to a `Manager` object. The result of this operation is shown in Figure 4-12.

Note that if `Employee`'s `Lookup` method had discovered that Joe was just an `Employee` and not a `Manager`, `Lookup` would have internally constructed an `Employee` object whose type object pointer member would have referred to the `Employee` type object, causing `Employee`'s implementation of `GetProgressReport` to execute instead of `Manager`'s implementation.

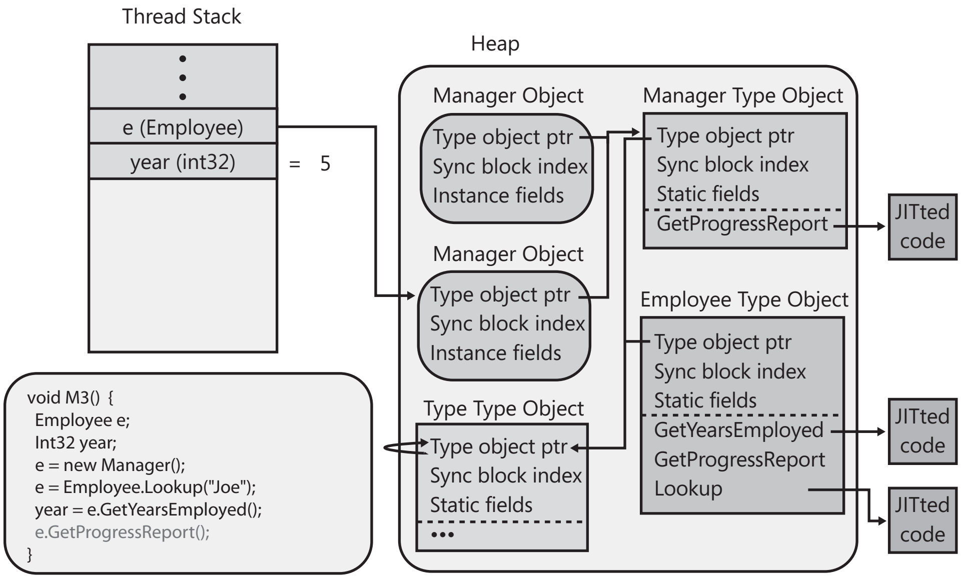


```
void M3() {
  Employee e;
  Int32 year;
  e = new Manager();
  e = Employee.Lookup("Joe");
  year = e.GetYearsEmployed();
  e.GetProgressReport();
}
```

FIGURE 4-12 Employee's virtual instance `GetProgressReport` method is called, causing `Manager`'s override of this method to execute.

At this point, we have discussed the relationship between source code, IL, and JITted code. We have also discussed the thread's stack, arguments, local variables, and how these arguments and variables refer to objects on the managed heap. You also see how objects contain a pointer to their type object (containing the static fields and method table). We have also discussed how the JIT compiler determines how to call static methods, nonvirtual instance methods, and virtual instance methods. All of this should give you great insight into how the CLR works, and this insight should help you when architecting and implementing your types, components, and applications. Before ending this chapter, I'd like to give you just a little more insight as to what is going on inside the CLR.

You'll notice that the `Employee` and `Manager` type objects both contain type object pointer members. This is because type objects are actually objects themselves. When the CLR creates type objects, the CLR must initialize these members. "To what?" you might ask. Well, when the CLR starts running in a process, it immediately creates a special type object for the `System.Type` type (defined in MSCorLib.dll). The `Employee` and `Manager` type objects are "instances" of this type, and therefore, their type object pointer members are initialized to refer to the `System.Type` type object, as shown in Figure 4-13.



```
void M3() {
  Employee e;
  Int32 year;
  e = new Manager();
  e = Employee.Lookup("Joe");
  year = e.GetYearsEmployed();
  e.GetProgressReport();
}
```

FIGURE 4-13 The `Employee` and `Manager` type objects are instances of the `System.Type` type.

Of course, the `System.Type` type object is an object itself and therefore also has a type object pointer member in it, and it is logical to ask what this member refers to. It refers to itself because the `System.Type` type object is itself an "instance" of a type object. And now you should understand the CLR's complete type system and how it works. By the way, `System.Object`'s `GetType` method simply returns the address stored in the specified object's type object pointer member. In other words, the `GetType` method returns a pointer to an object's type object, and this is how you can determine the true type of any object in the system (including type objects).