CHAPTER 5

Primitive, Reference, and Value Types

In this chapter:

In this chapter, I'll discuss the different kinds of types you'll run into as a Microsoft .NET Framework developer. It is crucial for all developers to be familiar with the different behaviors that these types exhibit. When I was first learning the .NET Framework, I didn't fully understand the difference between primitive, reference, and value types. This lack of clarity led me to unwittingly introduce subtle bugs and performance issues into my code. By explaining the differences between the types here, I'm hoping to save you some of the headaches that I experienced while getting up to speed.

Programming Language Primitive Types

Certain data types are so commonly used that many compilers allow code to manipulate them using simplified syntax. For example, you could allocate an integer by using the following syntax.

```
System.Int32 a = new System.Int32();
```

But I'm sure you'd agree that declaring and initializing an integer by using this syntax is rather cumbersome. Fortunately, many compilers (including C#) allow you to use syntax similar to the following instead.

```
int a = 0;
```

This syntax certainly makes the code more readable and generates identical Intermediate Language (IL) to that which is generated when `System.Int32` is used. Any data types the compiler directly supports are called primitive types. Primitive types map directly to types existing in the

Framework Class Library (FCL). For example, in C#, an `int` maps directly to the `System.Int32` type. Because of this, the following four lines of code all compile correctly and produce exactly the same IL.

```
int          a = 0;                  // Most convenient syntax
System.Int32 a = 0;                  // Convenient syntax
int          a = new int();          // Inconvenient syntax
System.Int32 a = new System.Int32(); // Most inconvenient syntax
```

Table 5-1 shows the FCL types that have corresponding primitives in C#. For the types that are compliant with the Common Language Specification (CLS), other languages will offer similar primitive types. However, languages aren't required to offer any support for the non–CLS-compliant types.

TABLE 5-1 C# Primitives with Corresponding FCL Types

Primitive Type	FCL Type	CLS-Compliant	Description
`sbyte`	`System.SByte`	No	Signed 8-bit value
`byte`	`System.Byte`	Yes	Unsigned 8-bit value
`short`	`System.Int16`	Yes	Signed 16-bit value
`ushort`	`System.UInt16`	No	Unsigned 16-bit value
`int`	`System.Int32`	Yes	Signed 32-bit value
`uint`	`System.UInt32`	No	Unsigned 32-bit value
`long`	`System.Int64`	Yes	Signed 64-bit value
`ulong`	`System.UInt64`	No	Unsigned 64-bit value
`char`	`System.Char`	Yes	16-bit Unicode character (`char` never represents an 8-bit value as it would in unmanaged C++.)
`float`	`System.Single`	Yes	IEEE 32-bit floating point value
`double`	`System.Double`	Yes	IEEE 64-bit floating point value
`bool`	`System.Boolean`	Yes	A true/false value
`decimal`	`System.Decimal`	Yes	A 128-bit high-precision floating-point value commonly used for financial calculations in which rounding errors can't be tolerated. Of the 128 bits, 1 bit represents the sign of the value, 96 bits represent the value itself, and 8 bits represent the power of 10 to divide the 96-bit value by (can be anywhere from 0 to 28). The remaining bits are unused.
`string`	`System.String`	Yes	An array of characters
`object`	`System.Object`	Yes	Base type of all types
`dynamic`	`System.Object`	Yes	To the common language runtime (CLR), `dynamic` is identical to `object`. However, the C# compiler allows dynamic variables to participate in dynamic dispatch by using a simplified syntax. For more information, see "The `dynamic` Primitive Type" section at the end of this chapter.

Another way to think of this is that the C# compiler automatically assumes that you have the following `using` directives (as discussed in Chapter 4, "Type Fundamentals") in all of your source code files.

```
using sbyte  = System.SByte;
using byte   = System.Byte;
using short  = System.Int16;
using ushort = System.UInt16;
using int    = System.Int32;
using uint   = System.UInt32;
...
```

The C# language specification states, "As a matter of style, use of the keyword is favored over use of the complete system type name." I disagree with the language specification; I prefer to use the FCL type names and completely avoid the primitive type names. In fact, I wish that compilers didn't even offer the primitive type names and forced developers to use the FCL type names instead. Here are my reasons:

- I've seen a number of developers confused, not knowing whether to use `string` or `String` in their code. Because in C# `string` (a keyword) maps exactly to `System.String` (an FCL type), there is no difference and either can be used. Similarly, I've heard some developers say that `int` represents a 32-bit integer when the application is running on a 32-bit operating system and that it represents a 64-bit integer when the application is running on a 64-bit operating system. This statement is absolutely false: in C#, an `int` always maps to `System.Int32`, and therefore it represents a 32-bit integer regardless of the operating system the code is running on. If programmers would use `Int32` in their code, then this potential confusion is also eliminated.

- In C#, `long` maps to `System.Int64`, but in a different programming language, `long` could map to an `Int16` or `Int32`. In fact, C++/CLI does treat `long` as an `Int32`. Someone reading source code in one language could easily misinterpret the code's intention if he or she were used to programming in a different programming language. In fact, most languages won't even treat `long` as a keyword and won't compile code that uses it.

- The FCL has many methods that have type names as part of their method names. For example, the `BinaryReader` type offers methods such as `ReadBoolean`, `ReadInt32`, `ReadSingle`, and so on, and the `System.Convert` type offers methods such as `ToBoolean`, `ToInt32`, `ToSingle`, and so on. Although it's legal to write the following code, the line with `float` feels very unnatural to me, and it's not obvious that the line is correct.

  ```
  BinaryReader br = new BinaryReader(...);
  float  val = br.ReadSingle();   // OK, but feels unnatural
  Single val = br.ReadSingle();   // OK and feels good
  ```

- Many programmers that use C# exclusively tend to forget that other programming languages can be used against the CLR, and because of this, C#-isms creep into the class library code. For example, Microsoft's FCL is almost exclusively written in C# and developers on the FCL team have now introduced methods into the library such as `Array`'s `GetLongLength`, which returns an `Int64` value that is a `long` in C# but not in other languages (like C++/CLI). Another example is `System.Linq.Enumerable`'s `LongCount` method.

For all of these reasons, I'll use the FCL type names throughout this book.

In many programming languages, you would expect the following code to compile and execute correctly.

```
Int32  i = 5;   // A 32-bit value
Int64  l = i;   // Implicit cast to a 64-bit value
```

However, based on the casting discussion presented in Chapter 4, you wouldn't expect this code to compile. After all, `System.Int32` and `System.Int64` are different types, and neither one is derived from the other. Well, you'll be happy to know that the C# compiler does compile this code correctly, and it runs as expected. Why? The reason is that the C# compiler has intimate knowledge of primitive types and applies its own special rules when compiling the code. In other words, the compiler recognizes common programming patterns and produces the necessary IL to make the written code work as expected. Specifically, the C# compiler supports patterns related to casting, literals, and operators, as shown in the following examples.

First, the compiler is able to perform implicit or explicit casts between primitive types such as the following.

```
Int32  i = 5;         // Implicit cast from Int32  to Int32
Int64  l = i;         // Implicit cast from Int32  to Int64
Single s = i;         // Implicit cast from Int32  to Single
Byte   b = (Byte) i;  // Explicit cast from Int32  to Byte
Int16  v = (Int16) s; // Explicit cast from Single to Int16
```

C# allows implicit casts if the conversion is "safe," that is, no loss of data is possible, such as converting an `Int32` to an `Int64`. But C# requires explicit casts if the conversion is potentially unsafe. For numeric types, "unsafe" means that you could lose precision or magnitude as a result of the conversion. For example, converting from `Int32` to `Byte` requires an explicit cast because precision might be lost from large `Int32` numbers; converting from `Single` to `Int16` requires a cast because `Single` can represent numbers of a larger magnitude than `Int16` can.

Be aware that different compilers can generate different code to handle these cast operations. For example, when casting a `Single` with a value of 6.8 to an `Int32`, some compilers could generate code to put a 6 in the `Int32`, and others could perform the cast by rounding the result up to 7. By the way, C# always truncates the result. For the exact rules that C# follows for casting primitive types, see the "Conversions" section in the C# language specification.

In addition to casting, primitive types can be written as literals. A literal is considered to be an instance of the type itself, and therefore, you can call instance methods by using the instance as shown here.

```
Console.WriteLine(123.ToString() + 456.ToString());  // "123456"
```

Also, if you have an expression consisting of literals, the compiler is able to evaluate the expression at compile time, improving the application's performance.

```
Boolean found = false;    // Generated code sets found to 0
Int32 x = 100 + 20 + 3;   // Generated code sets x to 123
String s = "a " + "bc";   // Generated code sets s to "a bc"
```

Finally, the compiler automatically knows how and in what order to interpret operators (such as +, -, *, /, %, &, ^, |, ==, !=, >, <, >=, <=, <<, >>, ~, !, ++, --, and so on) when used in code.

```
Int32 x = 100;                      // Assignment operator
Int32 y = x + 23;                   // Addition and assignment operators
Boolean lessThanFifty = (y < 50);   // Less-than and assignment operators
```

Checked and Unchecked Primitive Type Operations

Programmers are well aware that many arithmetic operations on primitives could result in an overflow.

```
Byte b = 100;
b = (Byte) (b + 200);          // b now contains 44 (or 2C in Hex).
```

Important When performing the preceding arithmetic operation, the first step requires that all operand values be expanded to 32-bit values (or 64-bit values if any operand requires more than 32 bits). So b and 200 (values requiring less than 32 bits) are first converted to 32-bit values and then added together. The result is a 32-bit value (300 in decimal, or 12C in hexadecimal) that must be cast to a Byte before the result can be stored back in the variable b. C# doesn't perform this cast for you implicitly, which is why the `Byte` cast on the second line of the preceding code is required.

In most programming scenarios, this silent overflow is undesirable and if not detected causes the application to behave in strange and unusual ways. In some rare programming scenarios (such as calculating a hash value or a checksum), however, this overflow is not only acceptable but is also desired.

Different languages handle overflows in different ways. C and C++ don't consider overflows to be an error and allow the value to wrap; the application continues running. Microsoft Visual Basic, on the other hand, always considers overflows to be errors and throws an exception when it detects one.

The CLR offers IL instructions that allow the compiler to choose the desired behavior. The CLR has an instruction called `add` that adds two values together. The `add` instruction performs no overflow checking. The CLR also has an instruction called `add.ovf` that also adds two values together. However, `add.ovf` throws a `System.OverflowException` if an overflow occurs. In addition to these two IL instructions for the `add` operation, the CLR also has similar IL instructions for subtraction (`sub`/`sub.ovf`), multiplication (`mul`/`mul.ovf`), and data conversions (`conv`/`conv.ovf`).

C# allows the programmer to decide how overflows should be handled. By default, overflow checking is turned off. This means that the compiler generates IL code by using the versions of the add, subtract, multiply, and conversion instructions that don't include overflow checking. As a result, the code runs faster—but developers must be assured that overflows won't occur or that their code is designed to anticipate these overflows.

One way to get the C# compiler to control overflows is to use the /checked+ compiler switch. This switch tells the compiler to generate code that has the overflow-checking versions of the add, subtract, multiply, and conversion IL instructions. The code executes a little slower because the CLR is checking these operations to determine whether an overflow occurred. If an overflow occurs, the CLR throws an OverflowException.

In addition to having overflow checking turned on or off globally, programmers can control overflow checking in specific regions of their code. C# allows this flexibility by offering checked and unchecked operators. Here's an example that uses the unchecked operator.

```
UInt32 invalid = unchecked((UInt32) (-1));  // OK
```

And here is an example that uses the checked operator.

```
Byte b = 100;
b = checked((Byte) (b + 200));    // OverflowException is thrown
```

In this example, b and 200 are first converted to 32-bit values and are then added together; the result is 300. Then 300 is converted to a Byte due to the explicit cast; this generates the OverflowException. If the Byte were cast outside the checked operator, the exception wouldn't occur.

```
b = (Byte) checked(b + 200);      // b contains 44; no OverflowException
```

In addition to the checked and unchecked operators, C# also offers checked and unchecked statements. The statements cause all expressions within a block to be checked or unchecked.

```
checked {                         // Start of checked block
   Byte b = 100;
   b = (Byte) (b + 200);          // This expression is checked for overflow.
}                                 // End of checked block
```

In fact, if you use a checked statement block, you can now use the += operator with the Byte, which simplifies the code a bit.

```
checked {         // Start of checked block
   Byte b = 100;
   b += 200;      // This expression is checked for overflow.
}                 // End of checked block
```

Important Because the only effect that the checked operator and statement have is to determine which versions of the add, subtract, multiply, and data conversion IL instructions are produced, calling a method within a checked operator or statement has no impact on that method, as the following code demonstrates.

```
checked {
   // Assume SomeMethod tries to load 400 into a Byte.
   SomeMethod(400);
   // SomeMethod might or might not throw an OverflowException.
   // It would if SomeMethod were compiled with checked instructions.
}
```

In my experience, I've seen a lot of calculations produce surprising results. Typically, this is due to invalid user input, but it can also be due to values returned from parts of the system that a programmer just doesn't expect. And so, I now recommend that programmers do the following:

- Use signed data types (such as `Int32` and `Int64`) instead of unsigned numeric types (such as `UInt32` and `UInt64`) wherever possible. This allows the compiler to detect more overflow/underflow errors. In addition, various parts of the class library (such as `Array`'s and `String`'s `Length` properties) are hard-coded to return signed values, and less casting is required as you move these values around in your code. Fewer casts make source code cleaner and easier to maintain. In addition, unsigned numeric types are not CLS-compliant.
- As you write your code, explicitly use `checked` around blocks where an unwanted overflow might occur due to invalid input data, such as processing a request with data supplied from an end user or a client machine. You might want to catch `OverflowException` as well, so that your application can gracefully recover from these failures.
- As you write your code, explicitly use `unchecked` around blocks where an overflow is OK, such as calculating a checksum.
- For any code that doesn't use `checked` or `unchecked`, the assumption is that you do want an exception to occur on overflow, for example, calculating something (such as prime numbers) where the inputs are known, and overflows are bugs.

Now, as you develop your application, turn on the compiler's `/checked+` switch for debug builds. Your application will run more slowly because the system will be checking for overflows on any code that you didn't explicitly mark as `checked` or `unchecked`. If an exception occurs, you'll easily detect it and be able to fix the bug in your code. For the release build of your application, use the compiler's `/checked-` switch so that the code runs faster and overflow exceptions won't be generated. To change the Checked setting in Microsoft Visual Studio, display the properties for your project, select the Build tab, click Advanced, and then select the Check For Arithmetic Overflow/Underflow option, as shown in Figure 5-1.

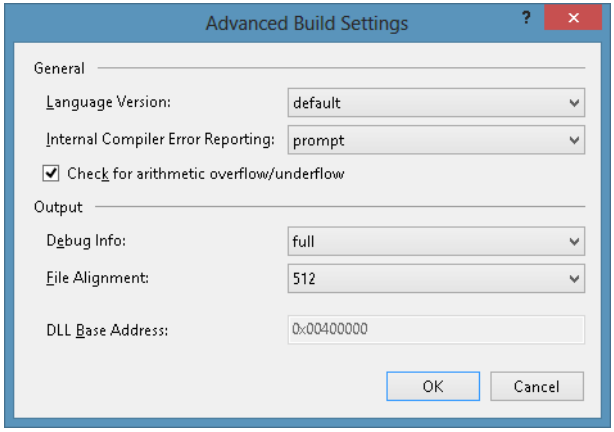

FIGURE 5-1 Changing the compiler's default setting for performing checked arithmetic by using Visual Studio's Advanced Build Settings dialog box.

If your application can tolerate the slight performance hit of always doing checked operations, then I recommend that you compile with the `/checked` command-line option even for a release build because this can prevent your application from continuing to run with corrupted data and possible security holes. For example, you might perform a multiplication to calculate an index into an array; it is much better to get an `OverflowException` as opposed to accessing an incorrect array element due to the math wrapping around.

Important The `System.Decimal` type is a very special type. Although many programming languages (C# and Visual Basic included) consider `Decimal` a primitive type, the CLR does not. This means that the CLR doesn't have IL instructions that know how to manipulate a `Decimal` value. If you look up the `Decimal` type in the .NET Framework SDK documentation, you'll see that it has public static methods called `Add`, `Subtract`, `Multiply`, `Divide`, and so on. In addition, the `Decimal` type provides operator overload methods for `+`, `-`, `*`, `/`, and so on.

When you compile code that uses `Decimal` values, the compiler generates code to call `Decimal`'s members to perform the actual operation. This means that manipulating `Decimal` values is slower than manipulating CLR primitive values. Also, because there are no IL instructions for manipulating `Decimal` values, the `checked` and `unchecked` operators, statements, and compiler switches have no effect. Operations on `Decimal` values always throw an `OverflowException` if the operation can't be performed safely.

Similarly, the `System.Numerics.BigInteger` type is also special in that it internally uses an array of `UInt32`s to represent an arbitrarily large integer whose value has no upper or lower bound. Therefore, operations on a `BigInteger` never result in an `OverflowException`. However, a `BigInteger` operation may throw an `OutOfMemoryException` if the value gets too large and there is insufficient available memory to resize the array.

Reference Types and Value Types

The CLR supports two kinds of types: reference types and value types. Although most types in the FCL are reference types, the types that programmers use most often are value types. Reference types are always allocated from the managed heap, and the C# `new` operator returns the memory address of the object—the memory address refers to the object's bits. You need to bear in mind some performance considerations when you're working with reference types. First, consider these facts:

- The memory must be allocated from the managed heap.
- Each object allocated on the heap has some additional overhead members associated with it that must be initialized.
- The other bytes in the object (for the fields) are always set to zero.
- Allocating an object from the managed heap could force a garbage collection to occur.

If every type were a reference type, an application's performance would suffer greatly. Imagine how poor performance would be if every time you used an `Int32` value, a memory allocation occurred! To improve performance for simple, frequently used types, the CLR offers lightweight types called value types. Value type instances are usually allocated on a thread's stack (although they can also be embedded as a field in a reference type object). The variable representing the instance doesn't contain a pointer to an instance; the variable contains the fields of the instance itself. Because the variable contains the instance's fields, a pointer doesn't have to be dereferenced to manipulate the instance's fields. Value type instances don't come under the control of the garbage collector, so their use reduces pressure in the managed heap and reduces the number of collections an application requires over its lifetime.

The .NET Framework SDK documentation clearly indicates which types are reference types and which are value types. When looking up a type in the documentation, any type called a class is a reference type. For example, the `System.Exception` class, the `System.IO.FileStream` class, and the `System.Random` class are all reference types. On the other hand, the documentation refers to each value type as a structure or an enumeration. For example, the `System.Int32` structure, the `System.Boolean` structure, the `System.Decimal` structure, the `System.TimeSpan` structure, the `System.DayOfWeek` enumeration, the `System.IO.FileAttributes` enumeration, and the `System.Drawing.FontStyle` enumeration are all value types.

All of the structures are immediately derived from the `System.ValueType` abstract type. `System.ValueType` is itself immediately derived from the `System.Object` type. By definition, all value types must be derived from `System.ValueType`. All enumerations are derived from the `System.Enum` abstract type, which is itself derived from `System.ValueType`. The CLR and all programming languages give enumerations special treatment. For more information about enumerated types, refer to Chapter 15, "Enumerated Types and Bit Flags."

Even though you can't choose a base type when defining your own value type, a value type can implement one or more interfaces if you choose. In addition, all value types are sealed, which prevents a value type from being used as a base type for any other reference type or value type. So, for example, it's not possible to define any new types using `Boolean`, `Char`, `Int32`, `UInt64`, `Single`, `Double`, `Decimal`, and so on as base types.

Important For many developers (such as unmanaged C/C++ developers), reference types and value types will seem strange at first. In unmanaged C/C++, you declare a type, and then the code that uses the type gets to decide if an instance of the type should be allocated on the thread's stack or in the application's heap. In managed code, the developer defining the type indicates where instances of the type are allocated; the developer using the type has no control over this.

The following code and Figure 5-2 demonstrate how reference types and value types differ.

```
// Reference type (because of 'class')
class  SomeRef { public Int32 x; }

// Value type (because of 'struct')
struct SomeVal { public Int32 x; }

static void ValueTypeDemo() {
   SomeRef r1 = new SomeRef();   // Allocated in heap
   SomeVal v1 = new SomeVal();   // Allocated on stack
   r1.x = 5;                     // Pointer dereference
   v1.x = 5;                     // Changed on stack
   Console.WriteLine(r1.x);      // Displays "5"
   Console.WriteLine(v1.x);      // Also displays "5"
   // The left side of Figure 5-2 reflects the situation
   // after the lines above have executed.

   SomeRef r2 = r1;              // Copies reference (pointer) only
   SomeVal v2 = v1;              // Allocate on stack & copies members
   r1.x = 8;                     // Changes r1.x and r2.x
   v1.x = 9;                     // Changes v1.x, not v2.x
   Console.WriteLine(r1.x);      // Displays "8"
   Console.WriteLine(r2.x);      // Displays "8"
   Console.WriteLine(v1.x);      // Displays "9"
   Console.WriteLine(v2.x);      // Displays "5"
   // The right side of Figure 5-2 reflects the situation
   // after ALL of the lines above have executed.
}
```

In this code, the `SomeVal` type is declared using `struct` instead of the more common `class`. In C#, types declared using `struct` are value types, and types declared using `class` are reference types. As you can see, the behavior of reference types and value types differs quite a bit. As you use types in your code, you must be aware of whether the type is a reference type or a value type because it can greatly affect how you express your intentions in the code.

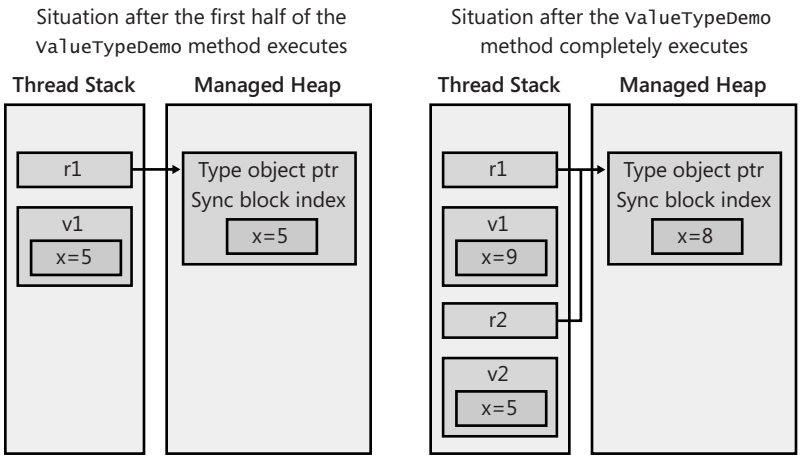

FIGURE 5-2 Visualizing the memory as the code executes.

In the preceding code, you saw this line.

```
SomeVal v1 = new SomeVal();   // Allocated on stack
```

The way this line is written makes it look as if a `SomeVal` instance will be allocated on the managed heap. However, the C# compiler knows that `SomeVal` is a value type and produces code that allocates the `SomeVal` instance on the thread's stack. C# also ensures that all of the fields in the value type instance are zeroed.

The preceding line could have been written like this instead.

```
SomeVal v1;   // Allocated on stack
```

This line also produces IL that allocates the instance on the thread's stack and zeroes the fields. The only difference is that C# "thinks" that the instance is initialized if you use the `new` operator. The following code will make this point clear.

```
// These two lines compile because C# thinks that
// v1's fields have been initialized to 0.
SomeVal v1 = new SomeVal();
Int32 a = v1.x;

// These two lines don't compile because C# doesn't think that
// v1's fields have been initialized to 0.
SomeVal v1;
Int32 a = v1.x;  // error CS0170: Use of possibly unassigned field 'x'
```

When designing your own types, consider carefully whether to define your types as value types instead of reference types. In some situations, value types can give better performance. In particular, you should declare a type as a value type if all the following statements are true:

- The type acts as a primitive type. Specifically, this means that it is a fairly simple type that has no members that modify any of its instance fields. When a type offers no members that alter its fields, we say that the type is immutable. In fact, it is recommended that many value types mark all their fields as `readonly` (discussed in Chapter 7, "Constants and Fields").
- The type doesn't need to inherit from any other type.
- The type won't have any other types derived from it.

The size of instances of your type is also a condition to take into account because by default, arguments are passed by value, which causes the fields in value type instances to be copied, hurting performance. Again, a method that returns a value type causes the fields in the instance to be copied into the memory allocated by the caller when the method returns, hurting performance. So, in addition to the previous conditions, you should declare a type as a value type if one of the following statements is true:

- Instances of the type are small (approximately 16 bytes or less).
- Instances of the type are large (greater than 16 bytes) and are not passed as method parameters or returned from methods.

The main advantage of value types is that they're not allocated as objects in the managed heap. Of course, value types have several limitations of their own when compared to reference types. Here are some of the ways in which value types and reference types differ:

- Value type objects have two representations: an unboxed form and a boxed form (discussed in the next section). Reference types are always in a boxed form.
- Value types are derived from `System.ValueType`. This type offers the same methods as defined by `System.Object`. However, `System.ValueType` overrides the `Equals` method so that it returns `true` if the values of the two objects' fields match. In addition, `System.ValueType` overrides the `GetHashCode` method to produce a hash code value by using an algorithm that takes into account the values in the object's instance fields. Due to performance issues with this default implementation, when defining your own value types, you should override and provide explicit implementations for the `Equals` and `GetHashCode` methods. I'll cover the `Equals` and `GetHashCode` methods at the end of this chapter.
- Because you can't define a new value type or a new reference type by using a value type as a base class, you shouldn't introduce any new virtual methods into a value type. No methods can be abstract, and all methods are implicitly sealed (can't be overridden).
- Reference type variables contain the memory address of objects in the heap. By default, when a reference type variable is created, it is initialized to `null`, indicating that the reference type variable doesn't currently point to a valid object. Attempting to use a `null` reference type variable causes a `NullReferenceException` to be thrown. By contrast, value type variables always contain a value of the underlying type, and all members of the value type are initialized to 0. Because a value type variable isn't a pointer, it's not possible to generate a `NullReferenceException` when accessing a value type. The CLR does offer a special feature that adds the notion of nullability to a value type. This feature, called nullable types, is discussed in Chapter 19, "Nullable Value Types."
- When you assign a value type variable to another value type variable, a field-by-field copy is made. When you assign a reference type variable to another reference type variable, only the memory address is copied.
- Because of the previous point, two or more reference type variables can refer to a single object in the heap, allowing operations on one variable to affect the object referenced by the other variable. On the other hand, value type variables are distinct objects, and it's not possible for operations on one value type variable to affect another.
- Because unboxed value types aren't allocated on the heap, the storage allocated for them is freed as soon as the method that defines an instance of the type is no longer active as opposed to waiting for a garbage collection.

How the CLR Controls the Layout of a Type's Fields

To improve performance, the CLR is capable of arranging the fields of a type any way it chooses. For example, the CLR might reorder fields in memory so that object references are grouped together and data fields are properly aligned and packed. However, when you define a type, you can tell the CLR whether it must keep the type's fields in the same order as the developer specified them or whether it can reorder them as it sees fit.

You tell the CLR what to do by applying the `System.Runtime.InteropServices.StructLayoutAttribute` attribute on the class or structure you're defining. To this attribute's constructor, you can pass `LayoutKind.Auto` to have the CLR arrange the fields, `LayoutKind.Sequential` to have the CLR preserve your field layout, or `LayoutKind.Explicit` to explicitly arrange the fields in memory by using offsets. If you don't explicitly specify the `StructLayoutAttribute` on a type that you're defining, your compiler selects whatever layout it determines is best.

You should be aware that Microsoft's C# compiler selects `LayoutKind.Auto` for reference types (classes) and `LayoutKind.Sequential` for value types (structures). It is obvious that the C# compiler team believes that structures are commonly used when interoperating with unmanaged code, and for this to work, the fields must stay in the order defined by the programmer. However, if you're creating a value type that has nothing to do with interoperability with unmanaged code, you could override the C# compiler's default. Here's an example.

```
using System;
using System.Runtime.InteropServices;

// Let the CLR arrange the fields to improve
// performance for this value type.
[StructLayout(LayoutKind.Auto)]
internal struct SomeValType {
   private readonly Byte m_b;
   private readonly Int16 m_x;
   ...
}
```

The `StructLayoutAttribute` also allows you to explicitly indicate the offset of each field by passing `LayoutKind.Explicit` to its constructor. Then you apply an instance of the `System.Runtime.InteropServices.FieldOffsetAttribute` attribute to each field passing to this attribute's constructor an `Int32` indicating the offset (in bytes) of the field's first byte from the beginning of the instance. Explicit layout is typically used to simulate what would be a union in unmanaged C/C++ because you can have multiple fields starting at the same offset in memory.

Here is an example.

```
using System;
using System.Runtime.InteropServices;

// The developer explicitly arranges the fields of this value type.
[StructLayout(LayoutKind.Explicit)]
internal struct SomeValType {
   [FieldOffset(0)]
   private readonly Byte m_b; // The m_b and m_x fields overlap each

   [FieldOffset(0)]
   private readonly Int16 m_x; // other in instances of this type
}
```

It should be noted that it is illegal to define a type in which a reference type and a value type overlap. It is possible to define a type in which multiple reference types overlap at the same starting offset; however, this is unverifiable. It is legal to define a type in which multiple value types overlap; however, all of the overlapping bytes must be accessible via public fields for the type to be verifiable.

Boxing and Unboxing Value Types

Value types are lighter weight than reference types because they are not allocated as objects in the managed heap, not garbage collected, and not referred to by pointers. However, in many cases, you must get a reference to an instance of a value type. For example, let's say that you wanted to create an `ArrayList` object (a type defined in the `System.Collections` namespace) to hold a set of `Point` structures. The code might look like this.

```
// Declare a value type.
struct Point {
   public Int32 x, y;
}

public sealed class Program {
   public static void Main() {
      ArrayList a = new ArrayList();
      Point p;            // Allocate a Point (not in the heap).
      for (Int32 i = 0; i < 10; i++) {
         p.x = p.y = i;   // Initialize the members in the value type.
         a.Add(p);        // Box the value type and add the
                          // reference to the Arraylist.
      }
      ...
   }
}
```

With each iteration of the loop, a `Point`'s value type fields are initialized. Then the `Point` is stored in the `ArrayList`. But let's think about this for a moment. What is actually being stored in the `ArrayList`? Is it the `Point` structure, the address of the `Point` structure, or something else entirely?

To get the answer, you must look up ArrayList's Add method and see what type its parameter is defined as. In this case, the Add method is prototyped as follows.

```
public virtual Int32 Add(Object value);
```

From this, you can plainly see that Add takes an Object as a parameter, indicating that Add requires a reference (or pointer) to an object on the managed heap as a parameter. But in the preceding code, I'm passing p, a Point, which is a value type. For this code to work, the Point value type must be converted into a true heap-managed object, and a reference to this object must be obtained.

It's possible to convert a value type to a reference type by using a mechanism called boxing. Internally, here's what happens when an instance of a value type is boxed:

1. Memory is allocated from the managed heap. The amount of memory allocated is the size required by the value type's fields plus the two additional overhead members (the type object pointer and the sync block index) required by all objects on the managed heap.
2. The value type's fields are copied to the newly allocated heap memory.
3. The address of the object is returned. This address is now a reference to an object; the value type is now a reference type.

The C# compiler automatically produces the IL code necessary to box a value type instance, but you still need to understand what's going on internally so that you're aware of code size and performance issues.

In the preceding code, the C# compiler detected that I was passing a value type to a method that requires a reference type, and it automatically emitted code to box the object. So at run time, the fields currently residing in the Point value type instance p are copied into the newly allocated Point object. The address of the boxed Point object (now a reference type) is returned and is then passed to the Add method. The Point object will remain in the heap until it is garbage collected. The Point value type variable (p) can be reused because the ArrayList never knows anything about it. Note that the lifetime of the boxed value type extends beyond the lifetime of the unboxed value type.

Note It should be noted that the FCL now includes a new set of generic collection classes that make the non-generic collection classes obsolete. For example, you should use the System.Collections.Generic.List<T> class instead of the System.Collections.ArrayList class. The generic collection classes offer many improvements over the non-generic equivalents. For example, the API has been cleaned up and improved, and the performance of the collection classes has been greatly improved as well. But one of the biggest improvements is that the generic collection classes allow you to work with collections of value types without requiring that items in the collection be boxed/unboxed. This in itself greatly improves performance because far fewer objects will be created on the managed heap, thereby reducing the number of garbage collections required by your application. Furthermore, you will get compile-time type safety, and your source code will be cleaner due to fewer casts. This will all be explained in further detail in Chapter 12, "Generics."

Now that you know how boxing works, let's talk about unboxing. Let's say that you want to grab the first element out of the `ArrayList` by using the following code.

```
Point p = (Point) a[0];
```

Here you're taking the reference (or pointer) contained in element 0 of the `ArrayList` and trying to put it into a `Point` value type instance, `p`. For this to work, all of the fields contained in the boxed `Point` object must be copied into the value type variable, `p`, which is on the thread's stack. The CLR accomplishes this copying in two steps. First, the address of the `Point` fields in the boxed `Point` object is obtained. This process is called unboxing. Then, the values of these fields are copied from the heap to the stack-based value type instance.

Unboxing is not the exact opposite of boxing. The unboxing operation is much less costly than boxing. Unboxing is really just the operation of obtaining a pointer to the raw value type (data fields) contained within an object. In effect, the pointer refers to the unboxed portion in the boxed instance. So, unlike boxing, unboxing doesn't involve the copying of any bytes in memory. Having made this important clarification, it is important to note that an unboxing operation is typically followed by copying the fields.

Obviously, boxing and unboxing/copy operations hurt your application's performance in terms of both speed and memory, so you should be aware of when the compiler generates code to perform these operations automatically and try to write code that minimizes this code generation.

Internally, here's exactly what happens when a boxed value type instance is unboxed:

1. If the variable containing the reference to the boxed value type instance is `null`, a `NullReferenceException` is thrown.
2. If the reference doesn't refer to an object that is a boxed instance of the desired value type, an `InvalidCastException` is thrown.[1]

The second item in the preceding list means that the following code will not work as you might expect.

```
public static void Main() {
   Int32  x = 5;
   Object o = x;         // Box x; o refers to the boxed object
   Int16  y = (Int16) o; // Throws an InvalidCastException
}
```

Logically, it makes sense to take the boxed `Int32` that `o` refers to and cast it to an `Int16`. However, when unboxing an object, the cast must be to the exact unboxed value type—`Int32` in this case. Here's the correct way to write this code.

```
public static void Main() {
   Int32  x = 5;
   Object o = x;                // Box x; o refers to the boxed object
   Int16  y = (Int16)(Int32) o; // Unbox to the correct type and cast
}
```

[1] The CLR also allows you to unbox a value type into a nullable version of the same value type. This is discussed in Chapter 19.

I mentioned earlier that an unboxing operation is frequently followed immediately by a field copy. Let's take a look at some C# code demonstrating that unbox and copy operations work together.

```
public static void Main() {
   Point p;
   p.x = p.y = 1;
   Object o = p;   // Boxes p; o refers to the boxed instance

   p = (Point) o;  // Unboxes o AND copies fields from boxed
                   // instance to stack variable
}
```

On the last line, the C# compiler emits an IL instruction to unbox `o` (get the address of the fields in the boxed instance) and another IL instruction to copy the fields from the heap to the stack-based variable `p`.

Now look at this code.

```
public static void Main() {
   Point p;
   p.x = p.y = 1;
   Object o = p;   // Boxes p; o refers to the boxed instance

   // Change Point's x field to 2
   p = (Point) o;  // Unboxes o AND copies fields from boxed
                   // instance to stack variable
   p.x = 2;        // Changes the state of the stack variable
   o = p;          // Boxes p; o refers to a new boxed instance
}
```

The code at the bottom of this fragment is intended only to change `Point`'s `x` field from `1` to `2`. To do this, an unbox operation must be performed, followed by a field copy, followed by changing the field (on the stack), followed by a boxing operation (which creates a whole new boxed instance in the managed heap). Hopefully, you see the impact that boxing and unboxing/copying operations have on your application's performance.

Some languages, such as C++/CLI, allow you to unbox a boxed value type without copying the fields. Unboxing returns the address of the unboxed portion of a boxed object (ignoring the object's type object pointer and sync block index overhead). You can now use this pointer to manipulate the unboxed instance's fields (which happen to be in a boxed object on the heap). For example, the previous code would be much more efficient if written in C++/CLI, because you could change the value of `Point`'s `x` field within the already boxed `Point` instance. This would avoid both allocating a new object on the heap and copying all of the fields twice!

Important If you're the least bit concerned about your application's performance, you must be aware of when the compiler produces the code that performs these operations. Unfortunately, many compilers implicitly emit code to box objects, so it is not obvious when you write code that boxing is occurring. If I am concerned about the performance of a particular algorithm, I always use a tool such as ILDasm.exe to view the IL code for my methods and see where the box IL instructions are.

Let's look at a few more examples that demonstrate boxing and unboxing.

```
public static void Main() {
   Int32  v = 5;            // Create an unboxed value type variable.
   Object o = v;            // o refers to a boxed Int32 containing 5.
   v = 123;                 // Changes the unboxed value to 123

   Console.WriteLine(v + ", " + (Int32) o); // Displays "123, 5"
}
```

In this code, can you guess how many boxing operations occur? You might be surprised to discover that the answer is three! Let's analyze the code carefully to really understand what's going on. To help you understand, I've included the IL code generated for the `Main` method shown in the preceding code. I've commented the code so that you can easily see the individual operations.

```
.method public hidebysig static void  Main() cil managed
{
  .entrypoint
  // Code size       45 (0x2d)
  .maxstack  3
  .locals init ([0]int32 v,
           [1] object o)
  // Load 5 into v.
  IL_0000:  ldc.i4.5
  IL_0001:  stloc.0

  // Box v and store the reference pointer in o.
  IL_0002:  ldloc.0
  IL_0003:  box        [mscorlib]System.Int32
  IL_0008:  stloc.1

  // Load 123 into v.
  IL_0009:  ldc.i4.s   123
  IL_000b:  stloc.0

  // Box v and leave the pointer on the stack for Concat.
  IL_000c:  ldloc.0
  IL_000d:  box        [mscorlib]System.Int32

  // Load the string on the stack for Concat.
  IL_0012:  ldstr      ", "

  // Unbox o: Get the pointer to the In32's field on the stack.
  IL_0017:  ldloc.1
  IL_0018:  unbox.any  [mscorlib]System.Int32

  // Box the Int32 and leave the pointer on the stack for Concat.
  IL_001d:  box        [mscorlib]System.Int32

  // Call Concat.
  IL_0022:  call       string [mscorlib]System.String::Concat(object,
                                                              object,
                                                              object)
```

```
  // The string returned from Concat is passed to WriteLine.
  IL_0027:  call       void [mscorlib]System.Console::WriteLine(string)

  // Return from Main terminating this application.
  IL_002c:  ret
} // end of method App::Main
```

First, an `Int32` unboxed value type instance (v) is created on the stack and initialized to 5. Then a variable (o) typed as `Object` is created, and is initialized to point to v. But because reference type variables must always point to objects in the heap, C# generated the proper IL code to box and store the address of the boxed copy of v in o. Now the value 123 is placed into the unboxed value type instance `v`; this has no effect on the boxed `Int32` value, which keeps its value of 5.

Next is the call to the `WriteLine` method. `WriteLine` wants a `String` object passed to it, but there is no string object. Instead, these three items are available: an unboxed `Int32` value type instance (v), a `String` (which is a reference type), and a reference to a boxed `Int32` value type instance (o) that is being cast to an unboxed `Int32`. These must somehow be combined to create a `String`.

To create a `String`, the C# compiler generates code that calls `String`'s static `Concat` method. There are several overloaded versions of the `Concat` method, all of which perform identically—the only difference is in the number of parameters. Because a string is being created from the concatenation of three items, the compiler chooses the following version of the `Concat` method.

```
public static String Concat(Object arg0, Object arg1, Object arg2);
```

For the first parameter, `arg0`, v is passed. But v is an unboxed value parameter and `arg0` is an `Object`, so v must be boxed and the address to the boxed v is passed for `arg0`. For the `arg1` parameter, the `","` string is passed as a reference to a `String` object. Finally, for the `arg2` parameter, o (a reference to an `Object`) is cast to an `Int32`. This requires an unboxing operation (but no copy operation), which retrieves the address of the unboxed `Int32` contained inside the boxed `Int32`. This unboxed `Int32` instance must be boxed again and the new boxed instance's memory address passed for `Concat`'s `arg2` parameter.

The `Concat` method calls each of the specified objects' `ToString` method and concatenates each object's string representation. The `String` object returned from `Concat` is then passed to `WriteLine` to show the final result.

I should point out that the generated IL code is more efficient if the call to `WriteLine` is written as follows.

```
Console.WriteLine(v + ", " + o);// Displays "123, 5"
```

This line is identical to the earlier version except that I've removed the `(Int32)` cast that preceded the variable o. This code is more efficient because o is already a reference type to an `Object` and its address can simply be passed to the `Concat` method. So, removing the cast saved two operations: an unbox and a box. You can easily see this savings by rebuilding the application and examining the generated IL code, as shown in the following code.

```
.method public hidebysig static void  Main() cil managed
{
  .entrypoint
  // Code size       35 (0x23)
  .maxstack  3
  .locals init ([0] int32 v,
           [1] object o)

  // Load 5 into v.
  IL_0000:  ldc.i4.5
  IL_0001:  stloc.0

  // Box v and store the reference pointer in o.
  IL_0002:  ldloc.0
  IL_0003:  box        [mscorlib]System.Int32
  IL_0008:  stloc.1

  // Load 123 into v.
  IL_0009:  ldc.i4.s   123
  IL_000b:  stloc.0

  // Box v and leave the pointer on the stack for Concat.
  IL_000c:  ldloc.0
  IL_000d:  box        [mscorlib]System.Int32

  // Load the string on the stack for Concat.
  IL_0012:  ldstr      ", "

  // Load the address of the boxed Int32 on the stack for Concat.
  IL_0017:  ldloc.1

  // Call Concat.
  IL_0018:  call       string [mscorlib]System.String::Concat(object,
                                                              object,
                                                              object)

  // The string returned from Concat is passed to WriteLine.
  IL_001d:  call       void [mscorlib]System.Console::WriteLine(string)

  // Return from Main terminating this application.
  IL_0022:  ret
} // end of method App::Main
```

A quick comparison of the IL for these two versions of the `Main` method shows that the version without the `(Int32)` cast is 10 bytes smaller than the version with the cast. The extra unbox/box steps in the first version are obviously generating more code. An even bigger concern, however, is that the extra boxing step allocates an additional object from the managed heap that must be garbage collected in the future. Certainly, both versions give identical results, and the difference in speed isn't noticeable, but extra, unnecessary boxing operations occurring in a loop cause the performance and memory usage of your application to be seriously degraded.

You can improve the previous code even more by calling `WriteLine` like this.

```
Console.WriteLine(v.ToString() + ", " + o);    // Displays "123, 5"
```

Now `ToString` is called on the unboxed value type instance `v`, and a `String` is returned. String objects are already reference types and can simply be passed to the `Concat` method without requiring any boxing.

Let's look at yet another example that demonstrates boxing and unboxing.

```
public static void Main() {
   Int32 v = 5;               // Create an unboxed value type variable.
   Object o = v;              // o refers to the boxed version of v.

   v = 123;                   // Changes the unboxed value type to 123
   Console.WriteLine(v);      // Displays "123"

   v = (Int32) o;             // Unboxes and copies o into v
   Console.WriteLine(v);      // Displays "5"
}
```

How many boxing operations do you count in this code? The answer is one. The reason that there is only one boxing operation is that the `System.Console` class defines a `WriteLine` method that accepts an `Int32` as a parameter.

```
public static void WriteLine(Int32 value);
```

In the two preceding calls to `WriteLine`, the variable `v`, an `Int32` unboxed value type instance, is passed by value. Now it may be that `WriteLine` will box this `Int32` internally, but you have no control over that. The important thing is that you've done the best you could and have eliminated the boxing from your own code.

If you take a close look at the FCL, you'll notice many overloaded methods that differ based on their value type parameters. For example, the `System.Console` type offers several overloaded versions of the `WriteLine` method.

```
public static void WriteLine(Boolean);
public static void WriteLine(Char);
public static void WriteLine(Char[]);
public static void WriteLine(Int32);
public static void WriteLine(UInt32);
public static void WriteLine(Int64);
public static void WriteLine(UInt64);
public static void WriteLine(Single);
public static void WriteLine(Double);
public static void WriteLine(Decimal);
public static void WriteLine(Object);
public static void WriteLine(String);
```

You'll also find a similar set of overloaded methods for `System.Console`'s `Write` method, `System.IO.BinaryWriter`'s `Write` method, `System.IO.TextWriter`'s `Write` and `WriteLine` methods, `System.Runtime.Serialization.SerializationInfo`'s `AddValue` method, `System.Text.StringBuilder`'s `Append` and `Insert` methods, and so on. Most of these methods offer overloaded versions for the sole purpose of reducing the number of boxing operations for the common value types.

If you define your own value type, these FCL classes will not have overloads of these methods that accept your value type. Furthermore, there are a bunch of value types already defined in the FCL for which overloads of these methods do not exist. If you call a method that does not have an overload for the specific value type that you are passing to it, you will always end up calling the overload that takes an `Object`. Passing a value type instance as an `Object` will cause boxing to occur, which will adversely affect performance. If you are defining your own class, you can define the methods in the class to be generic (possibly constraining the type parameters to be value types). Generics give you a way to define a method that can take any kind of value type without having to box it. Generics are discussed in Chapter 12.

One last point about boxing: if you know that the code that you're writing is going to cause the compiler to box a single value type repeatedly, your code will be smaller and faster if you manually box the value type. Here's an example.

```
using System;

public sealed class Program {
   public static void Main() {
      Int32 v = 5;   // Create an unboxed value type variable.

#if INEFFICIENT
      // When compiling the following line, v is boxed
      // three times, wasting time and memory.
      Console.WriteLine("{0}, {1}, {2}", v, v, v);
#else
      // The lines below have the same result, execute
      // much faster, and use less memory.
      Object o = v;  // Manually box v (just once).

      // No boxing occurs to compile the following line.
      Console.WriteLine("{0}, {1}, {2}", o, o, o);
#endif
   }
}
```

If this code is compiled with the `INEFFICIENT` symbol defined, the compiler will generate code to box v three times, causing three objects to be allocated from the heap! This is extremely wasteful because each object will have exactly the same contents: 5. If the code is compiled without the `INEFFICIENT` symbol defined, v is boxed just once, so only one object is allocated from the heap. Then, in the call to `Console.WriteLine`, the reference to the single boxed object is passed three times. This second version executes much faster and allocates less memory from the heap.

In these examples, it's fairly easy to recognize when an instance of a value type requires boxing. Basically, if you want a reference to an instance of a value type, the instance must be boxed. Usually this happens because you have a value type instance and you want to pass it to a method that requires a reference type. However, this situation isn't the only one in which you'll need to box an instance of a value type.

Recall that unboxed value types are lighter-weight types than reference types for two reasons:

- They are not allocated on the managed heap.
- They don't have the additional overhead members that every object on the heap has: a type object pointer and a sync block index.

Because unboxed value types don't have a sync block index, you can't have multiple threads synchronize their access to the instance by using the methods of the `System.Threading.Monitor` type (or by using C#'s `lock` statement).

Even though unboxed value types don't have a type object pointer, you can still call virtual methods (such as `Equals`, `GetHashCode`, or `ToString`) inherited or overridden by the type. If your value type overrides one of these virtual methods, then the CLR can invoke the method nonvirtually because value types are implicitly sealed and cannot have any types derived from them. In addition, the value type instance being used to invoke the virtual method is not boxed. However, if your override of the virtual method calls into the base type's implementation of the method, then the value type instance does get boxed when calling the base type's implementation so that a reference to a heap object gets passed to the `this` pointer into the base method.

However, calling a nonvirtual inherited method (such as `GetType` or `MemberwiseClone`) always requires the value type to be boxed because these methods are defined by `System.Object`, so the methods expect the `this` argument to be a pointer that refers to an object on the heap.

In addition, casting an unboxed instance of a value type to one of the type's interfaces requires the instance to be boxed, because interface variables must always contain a reference to an object on the heap. (I'll talk about interfaces in Chapter 13, "Interfaces.") The following code demonstrates.

```
using System;

internal struct Point : IComparable {
   private readonly Int32 m_x, m_y;

   // Constructor to easily initialize the fields
   public Point(Int32 x, Int32 y) {
      m_x = x;
      m_y = y;
   }

   // Override ToString method inherited from System.ValueType
   public override String ToString() {
      // Return the point as a string. Note: calling ToString prevents boxing
      return String.Format("({0}, {1})", m_x.ToString(), m_y.ToString());
   }

   // Implementation of type-safe CompareTo method
   public Int32 CompareTo(Point other) {
      // Use the Pythagorean Theorem to calculate
      // which point is farther from the origin (0, 0)
      return Math.Sign(Math.Sqrt(m_x * m_x + m_y * m_y)
         - Math.Sqrt(other.m_x * other.m_x + other.m_y * other.m_y));
   }
```

```
   // Implementation of IComparable's CompareTo method
   public Int32 CompareTo(Object o) {
      if (GetType() != o.GetType()) {
         throw new ArgumentException("o is not a Point");
      }
      // Call type-safe CompareTo method
      return CompareTo((Point) o);
   }
}


public static class Program {
   public static void Main() {
      // Create two Point instances on the stack.
      Point p1 = new Point(10, 10);
      Point p2 = new Point(20, 20);

      // p1 does NOT get boxed to call ToString (a virtual method).
      Console.WriteLine(p1.ToString());// "(10, 10)"

      // p DOES get boxed to call GetType (a non-virtual method).
      Console.WriteLine(p1.GetType());// "Point"

      // p1 does NOT get boxed to call CompareTo.
      // p2 does NOT get boxed because CompareTo(Point) is called.
      Console.WriteLine(p1.CompareTo(p2));// "-1"

      // p1 DOES get boxed, and the reference is placed in c.
      IComparable c = p1;
      Console.WriteLine(c.GetType());// "Point"

      // p1 does NOT get boxed to call CompareTo.
      // Because CompareTo is not being passed a Point variable,
      // CompareTo(Object) is called, which requires a reference to
      // a boxed Point.
      // c does NOT get boxed because it already refers to a boxed Point.
      Console.WriteLine(p1.CompareTo(c));// "0"

      // c does NOT get boxed because it already refers to a boxed Point.
      // p2 does get boxed because CompareTo(Object) is called.
      Console.WriteLine(c.CompareTo(p2));// "-1"

      // c is unboxed, and fields are copied into p2.
      p2 = (Point) c;

      // Proves that the fields got copied into p2.
      Console.WriteLine(p2.ToString());// "(10, 10)"
   }
}
```

This code demonstrates several scenarios related to boxing and unboxing:

- **Calling ToString** In the call to `ToString`, `p1` doesn't have to be boxed. At first, you'd think that `p1` would have to be boxed because `ToString` is a virtual method that is inherited from the base type, `System.ValueType`. Normally, to call a virtual method, the CLR needs to determine the object's type in order to locate the type's method table. Because `p1` is an unboxed

value type, there's no type object pointer. However, the just-in-time (JIT) compiler sees that `Point` overrides the `ToString` method, and it emits code that calls `ToString` directly (non-virtually) without having to do any boxing. The compiler knows that polymorphism can't come into play here because `Point` is a value type, and no type can derive from it to provide another implementation of this virtual method. Note that if `Point`'s `ToString` method internally calls `base.ToString()`, then the value type instance would be boxed when calling `System.ValueType`'s `ToString` method.

- **Calling GetType** In the call to the nonvirtual `GetType` method, `p1` does have to be boxed. The reason is that the `Point` type inherits `GetType` from `System.Object`. So to call `GetType`, the CLR must use a pointer to a type object, which can be obtained only by boxing `p1`.

- **Calling CompareTo (first time)** In the first call to `CompareTo`, `p1` doesn't have to be boxed because `Point` implements the `CompareTo` method, and the compiler can just call it directly.

  Note that a `Point` variable (`p2`) is being passed to `CompareTo`, and therefore the compiler calls the overload of `CompareTo` that accepts a `Point` parameter. This means that `p2` will be passed by value to `CompareTo` and no boxing is necessary.

- **Casting to IComparable** When casting `p1` to a variable (`c`) that is of an interface type, `p1` must be boxed because interfaces are reference types by definition. So `p1` is boxed, and the pointer to this boxed object is stored in the variable `c`. The following call to `GetType` proves that `c` does refer to a boxed `Point` on the heap.

- **Calling CompareTo (second time)** In the second call to `CompareTo`, `p1` doesn't have to be boxed because `Point` implements the `CompareTo` method, and the compiler can just call it directly. Note that an `IComparable` variable (`c`) is being passed to `CompareTo`, and therefore, the compiler calls the overload of `CompareTo` that accepts an `Object` parameter. This means that the argument passed must be a pointer that refers to an object on the heap. Fortunately, `c` does refer to a boxed `Point`, and therefore, that memory address in `c` can be passed to `CompareTo`, and no additional boxing is necessary.

- **Calling CompareTo (third time)** In the third call to `CompareTo`, `c` already refers to a boxed `Point` object on the heap. Because `c` is of the `IComparable` interface type, you can call only the interface's `CompareTo` method that requires an `Object` parameter. This means that the argument passed must be a pointer that refers to an object on the heap. So `p2` is boxed, and the pointer to this boxed object is passed to `CompareTo`.

- **Casting to Point** When casting `c` to a `Point`, the object on the heap referred to by `c` is unboxed, and its fields are copied from the heap to `p2`, an instance of the `Point` type residing on the stack.

I realize that all of this information about reference types, value types, and boxing might be overwhelming at first. However, a solid understanding of these concepts is critical to any .NET Framework developer's long-term success. Trust me: having a solid grasp of these concepts will allow you to build efficient applications faster and easier.

Changing Fields in a Boxed Value Type by Using Interfaces (and Why You Shouldn't Do This)

Let's have some fun and see how well you understand value types, boxing, and unboxing. Examine the following code, and see whether you can figure out what it displays on the console.

```
using System;

// Point is a value type.
internal struct Point {
   private Int32 m_x, m_y;

   public Point(Int32 x, Int32 y) {
      m_x = x;
      m_y = y;
   }

   public void Change(Int32 x, Int32 y) {
      m_x = x; m_y = y;
   }

   public override String ToString() {
      return String.Format("({0}, {1})", m_x.ToString(), m_y.ToString());
   }
}

public sealed class Program {
   public static void Main() {
      Point p = new Point(1, 1);

      Console.WriteLine(p);

      p.Change(2, 2);
      Console.WriteLine(p);

      Object o = p;
      Console.WriteLine(o);

      ((Point) o).Change(3, 3);
      Console.WriteLine(o);
   }
}
```

Very simply, `Main` creates an instance (`p`) of a `Point` value type on the stack and sets its `m_x` and `m_y` fields to 1. Then, `p` is boxed before the first call to `WriteLine`, which calls `ToString` on the boxed `Point`, and `(1, 1)` is displayed as expected. Then, `p` is used to call the `Change` method, which changes the values of `p`'s `m_x` and `m_y` fields on the stack to 2. The second call to `WriteLine` requires `p` to be boxed again and displays `(2, 2)`, as expected.

Now, `p` is boxed a third time, and `o` refers to the boxed `Point` object. The third call to `WriteLine` again shows `(2, 2)`, which is also expected. Finally, I want to call the `Change` method to update the fields in the boxed `Point` object. However, `Object` (the type of the variable `o`) doesn't know anything about the `Change` method, so I must first cast `o` to a `Point`. Casting `o` to a `Point` unboxes `o` and

copies the fields in the boxed `Point` to a temporary `Point` on the thread's stack! The `m_x` and `m_y` fields of this temporary point are changed to 3 and 3, but the boxed `Point` isn't affected by this call to `Change`. When `WriteLine` is called the fourth time, `(2, 2)` is displayed again. Many developers do not expect this.

Some languages, such as C++/CLI, let you change the fields in a boxed value type, but C# does not. However, you can fool C# into allowing this by using an interface. The following code is a modified version of the previous code.

```
using System;

// Interface defining a Change method
internal interface IChangeBoxedPoint {
   void Change(Int32 x, Int32 y);
}


// Point is a value type.
internal struct Point : IChangeBoxedPoint {
   private Int32 m_x, m_y;

   public Point(Int32 x, Int32 y) {
      m_x = x;
      m_y = y;
   }

   public void Change(Int32 x, Int32 y) {
      m_x = x; m_y = y;
   }

   public override String ToString() {
      return String.Format("({0}, {1})", m_x.ToString(), m_y.ToString());
   }
}

public sealed class Program {
   public static void Main() {
      Point p = new Point(1, 1);

      Console.WriteLine(p);

      p.Change(2, 2);
      Console.WriteLine(p);

      Object o = p;
      Console.WriteLine(o);

      ((Point) o).Change(3, 3);
      Console.WriteLine(o);

      // Boxes p, changes the boxed object and discards it
      ((IChangeBoxedPoint) p).Change(4, 4);
      Console.WriteLine(p);
```

```
      // Changes the boxed object and shows it
      ((IChangeBoxedPoint) o).Change(5, 5);
      Console.WriteLine(o);
   }
}
```

This code is almost identical to the previous version. The main difference is that the `Change` method is defined by the `IChangeBoxedPoint` interface, and the `Point` type now implements this interface. Inside `Main`, the first four calls to `WriteLine` are the same and produce the same results I had before (as expected). However, I've added two more examples at the end of `Main`.

In the first example, the unboxed `Point`, `p`, is cast to an `IChangeBoxedPoint`. This cast causes the value in `p` to be boxed. `Change` is called on the boxed value, which does change its `m_x` and `m_y` fields to 4 and 4, but after `Change` returns, the boxed object is immediately ready to be garbage collected. So the fifth call to `WriteLine` displays `(2, 2)`. Many developers won't expect this result.

In the last example, the boxed `Point` referred to by `o` is cast to an `IChangeBoxedPoint`. No boxing is necessary here because `o` is already a boxed `Point`. Then `Change` is called, which does change the boxed `Point`'s `m_x` and `m_y` fields. The interface method `Change` has allowed me to change the fields in a boxed `Point` object! Now, when `WriteLine` is called, it displays `(5, 5)` as expected. The purpose of this whole example is to demonstrate how an interface method is able to modify the fields of a boxed value type. In C#, this isn't possible without using an interface method.

Important Earlier in this chapter, I mentioned that value types should be immutable: that is, they should not define any members that modify any of the type's instance fields. In fact, I recommended that value types have their fields marked as `readonly` so that the compiler will issue errors should you accidentally write a method that attempts to modify a field. The previous example should make it very clear to you why value types should be immutable. The unexpected behaviors shown in the previous example all occur when attempting to call a method that modifies the value type's instance fields. If after constructing a value type, you do not call any methods that modify its state, you will not get confused when all of the boxing and unboxing/field copying occurs. If the value type is immutable, you will end up just copying the same state around, and you will not be surprised by any of the behaviors you see.

A number of developers reviewed the chapters of this book. After reading through some of my code samples (such as the preceding one), these reviewers would tell me that they've sworn off value types. I must say that these little value type nuances have cost me days of debugging time, which is why I spend time pointing them out in this book. I hope you'll remember some of these nuances and that you'll be prepared for them if and when they strike you and your code. Certainly, you shouldn't be scared of value types. They are useful, and they have their place. After all, a program needs a little `Int32` love now and then. Just keep in mind that value types and reference types have very different behaviors depending on how they're used. In fact, you should take the preceding code and declare

the Point as a class instead of a struct to appreciate the different behavior that results. Finally, you'll be very happy to know that the core value types that ship in the FCL—Byte, Int32, UInt32, Int64, UInt64, Single, Double, Decimal, BigInteger, Complex, all enums, and so on—are all immutable, so you should experience no surprising behavior when using any of these types.

Object Equality and Identity

Frequently, developers write code to compare objects with one another. This is particularly true when placing objects in a collection and you're writing code to sort, search, or compare items in a collection. In this section, I'll discuss object equality and identity, and I'll also discuss how to define a type that properly implements object equality.

The System.Object type offers a virtual method named Equals, whose purpose is to return true if two objects contain the same value. The implementation of Object's Equals method looks like this.

```
public class Object {
   public virtual Boolean Equals(Object obj) {

      // If both references point to the same object,
      // they must have the same value.
      if (this == obj) return true;

      // Assume that the objects do not have the same value.
      return false;
   }
}
```

At first, this seems like a reasonable default implementation of Equals: it returns true if the this and obj arguments refer to the same exact object. This seems reasonable because Equals knows that an object must have the same value as itself. However, if the arguments refer to different objects, Equals can't be certain if the objects contain the same values, and therefore, false is returned. In other words, the default implementation of Object's Equals method really implements identity, not value equality.

Unfortunately, as it turns out, Object's Equals method is not a reasonable default, and it should have never been implemented this way. You immediately see the problem when you start thinking about class inheritance hierarchies and how to properly override Equals. Here is how to properly implement an Equals method internally:

1. If the obj argument is null, return false because the current object identified by this is obviously not null when the nonstatic Equals method is called.
2. If the this and obj arguments refer to the same object, return true. This step can improve performance when comparing objects with many fields.

3. If the `this` and `obj` arguments refer to objects of different types, return `false`. Obviously, checking if a `String` object is equal to a `FileStream` object should result in a `false` result.

4. For each instance field defined by the type, compare the value in the `this` object with the value in the `obj` object. If any fields are not equal, return `false`.

5. Call the base class's `Equals` method so it can compare any fields defined by it. If the base class's `Equals` method returns `false`, return `false`; otherwise, return `true`.

So Microsoft should have implemented `Object`'s `Equals` like this.

```
public class Object {
   public virtual Boolean Equals(Object obj) {
      // The given object to compare to can't be null
      if (obj == null) return false;

     // If objects are different types, they can't be equal.
      if (this.GetType() != obj.GetType()) return false;

      // If objects are same type, return true if all of their fields match
      // Because System.Object defines no fields, the fields match
      return true;
   }
}
```

But, because Microsoft didn't implement `Equals` this way, the rules for how to implement `Equals` are significantly more complicated than you would think. When a type overrides `Equals`, the override should call its base class's implementation of `Equals` unless it would be calling `Object`'s implementation. This also means that because a type can override `Object`'s `Equals` method, this `Equals` method can no longer be called to test for identity. To fix this, `Object` offers a static `ReferenceEquals` method, which is implemented like this.

```
public class Object {
   public static Boolean ReferenceEquals(Object objA, Object objB) {
      return (objA == objB);
   }
}
```

You should always call `ReferenceEquals` if you want to check for identity (if two references point to the same object). You shouldn't use the C# `==` operator (unless you cast both operands to `Object` first) because one of the operands' types could overload the `==` operator, giving it semantics other than identity.

As you can see, the .NET Framework has a very confusing story when it comes to object equality and identity. By the way, `System.ValueType` (the base class of all value types) does override `Object`'s `Equals` method and is correctly implemented to perform a value equality check (not an identity check). Internally, `ValueType`'s `Equals` is implemented this way:

1. If the `obj` argument is `null`, return `false`.

2. If the `this` and `obj` arguments refer to objects of different types, return `false`.

3. For each instance field defined by the type, compare the value in the `this` object with the value in the `obj` object by calling the field's `Equals` method. If any fields are not equal, return `false`.

4. Return `true`. `Object`'s `Equals` method is not called by `ValueType`'s `Equals` method.

Internally, `ValueType`'s `Equals` method uses reflection (covered in Chapter 23, "Assembly Loading and Reflection") to accomplish step 3. Because the CLR's reflection mechanism is slow, when defining your own value type, you should override `Equals` and provide your own implementation to improve the performance of value equality comparisons that use instances of your type. Of course, in your own implementation, do not call `base.Equals`.

When defining your own type, if you decide to override `Equals`, you must ensure that it adheres to the four properties of equality:

- `Equals` must be reflexive; that is, `x.Equals(x)` must return `true`.
- `Equals` must be symmetric; that is, `x.Equals(y)` must return the same value as `y.Equals(x)`.
- `Equals` must be transitive; that is, if `x.Equals(y)` returns `true` and `y.Equals(z)` returns `true`, then `x.Equals(z)` must also return `true`.
- `Equals` must be consistent. Provided that there are no changes in the two values being compared, `Equals` should consistently return `true` or `false`.

If your implementation of `Equals` fails to adhere to all of these rules, your application will behave in strange and unpredictable ways.

When overriding the `Equals` method, there are a couple more things that you'll probably want to do:

- **Have the type implement the `System.IEquatable<T>` interface's `Equals` method** This generic interface allows you to define a type-safe `Equals` method. Usually, you'll implement the `Equals` method that takes an `Object` parameter to internally call the type-safe `Equals` method.
- **Overload the == and != operator methods** Usually, you'll implement these operator methods to internally call the type-safe `Equals` method.

Furthermore, if you think that instances of your type will be compared for the purposes of sorting, you'll want your type to also implement `System.IComparable`'s `CompareTo` method and `System.IComparable<T>`'s type-safe `CompareTo` method. If you implement these methods, you'll also want to overload the various comparison operator methods (<, <=, >, >=) and implement these methods internally to call the type-safe `CompareTo` method.

Object Hash Codes

The designers of the FCL decided that it would be incredibly useful if any instance of any object could be placed into a hash table collection. To this end, `System.Object` provides a virtual `GetHashCode` method so that an `Int32` hash code can be obtained for any and all objects.

If you define a type and override the `Equals` method, you should also override the `GetHashCode` method. In fact, Microsoft's C# compiler emits a warning if you define a type that overrides `Equals` without also overriding `GetHashCode`. For example, compiling the following type yields this warning: `warning CS0659: 'Program' overrides Object.Equals(object o) but does not override Object.GetHashCode()`.

```
public sealed class Program {
   public override Boolean Equals(Object obj) { ... }
}
```

The reason a type that defines `Equals` must also define `GetHashCode` is that the implementation of the `System.Collections.Hashtable` type, the `System.Collections.Generic.Dictionary` type, and some other collections require that any two objects that are equal must have the same hash code value. So if you override `Equals`, you should override `GetHashCode` to ensure that the algorithm you use for calculating equality corresponds to the algorithm you use for calculating the object's hash code.

Basically, when you add a key/value pair to a collection, a hash code for the key object is obtained first. This hash code indicates which "bucket" the key/value pair should be stored in. When the collection needs to look up a key, it gets the hash code for the specified key object. This code identifies the "bucket" that is now searched sequentially, looking for a stored key object that is equal to the specified key object. Using this algorithm of storing and looking up keys means that if you change a key object that is in a collection, the collection will no longer be able to find the object. If you intend to change a key object in a hash table, you should remove the original key/value pair, modify the key object, and then add the new key/value pair back into the hash table.

Defining a `GetHashCode` method can be easy and straightforward. But depending on your data types and the distribution of data, it can be tricky to come up with a hashing algorithm that returns a well-distributed range of values. Here's a simple example that will probably work just fine for `Point` objects.

```
internal sealed class Point {
   private readonly Int32 m_x, m_y;
   public override Int32 GetHashCode() {
      return m_x ^ m_y;  // m_x XOR'd with m_y
   }
   ...
}
```

When selecting an algorithm for calculating hash codes for instances of your type, try to follow these guidelines:

- Use an algorithm that gives a good random distribution for the best performance of the hash table.
- Your algorithm can also call the base type's `GetHashCode` method, including its return value. However, you don't generally want to call `Object`'s or `ValueType`'s `GetHashCode` method, because the implementation in either method doesn't lend itself to high-performance hashing algorithms.
- Your algorithm should use at least one instance field.
- Ideally, the fields you use in your algorithm should be immutable; that is, the fields should be initialized when the object is constructed, and they should never again change during the object's lifetime.
- Your algorithm should execute as quickly as possible.
- Objects with the same value should return the same code. For example, two `String` objects with the same text should return the same hash code value.

`System.Object`'s implementation of the `GetHashCode` method doesn't know anything about its derived type and any fields that are in the type. For this reason, `Object`'s `GetHashCode` method returns a number that is guaranteed not to change for the lifetime of the object.

Important If you're implementing your own hash table collection for some reason, or you're implementing any piece of code in which you'll be calling `GetHashCode`, you should never, ever persist hash code values. The reason is that hash code values are subject to change. For example, a future version of a type might use a different algorithm for calculating the object's hash code.

There is a company that was not heeding this important warning. On their website, users could create new accounts by selecting a user name and a password. The website then took the password `String`, called `GetHashCode`, and persisted the hash code value in a database. When users logged back on to the website, they entered their password. The website would call `GetHashCode` again and compare the hash code value with the stored value in the database. If the hash codes matched, the user would be granted access. Unfortunately, when the company upgraded to a new version of the CLR, `String`'s `GetHashCode` method had changed, and it now returned a different hash code value. The end result was that no user was able to log on to the website anymore!

The dynamic Primitive Type

C# is a type-safe programming language. This means that all expressions resolve into an instance of a type and the compiler will generate only code that is attempting to perform an operation that is valid for this type. The benefit of a type-safe programming language over a non–type-safe programming language is that many programmer errors are detected at compile time, helping to ensure that the code is correct before you attempt to execute it. In addition, compile-time languages can typically produce smaller and faster code because they make more assumptions at compile time and bake those assumptions into the resulting IL and metadata.

However, there are also many occasions when a program has to act on information that it doesn't know about until it is running. Although you can use type-safe programming languages (like C#) to interact with this information, the syntax tends to be clumsy, especially because you tend to work a lot with strings, and performance is hampered as well. If you are writing a pure C# application, then the only occasion you have for working with runtime-determined information is when you are using reflection (discussed in Chapter 23). However, many developers also use C# to communicate with components that are not implemented in C#. Some of these components could be .NET-dynamic languages such as Python or Ruby, or COM objects that support the `IDispatch` interface (possibly implemented in native C or C++), or HTML Document Object Model (DOM) objects (implemented using various languages and technologies). Communicating with HTML DOM objects is particularly useful when building a Microsoft Silverlight application.

To make it easier for developers using reflection or communicating with other components, the C# compiler offers you a way to mark an expression's type as `dynamic`. You can also put the result of an expression into a variable and you can mark a variable's type as `dynamic`. This `dynamic` expression/variable can then be used to invoke a member such as a field, a property/indexer, a method, delegate, and unary/binary/conversion operators. When your code invokes a member by using a dynamic expression/variable, the compiler generates special IL code that describes the desired operation. This special code is referred to as the payload. At run time, the payload code determines the exact operation to execute based on the actual type of the object now referenced by the `dynamic` expression/variable.

Here is some code to demonstrate what I'm talking about.

```
internal static class DynamicDemo {
   public static void Main() {
      dynamic value;
      for (Int32 demo = 0; demo < 2; demo++) {
         value = (demo == 0) ? (dynamic) 5 : (dynamic) "A";
         value = value + value;
         M(value);
      }
   }

   private static void M(Int32 n)  { Console.WriteLine("M(Int32): " + n); }
   private static void M(String s) { Console.WriteLine("M(String): " + s); }
}
```

When I execute `Main`, I get the following output.

```
M(Int32): 10
M(String): AA
```

To understand what's happening, let's start by looking at the + operator. This operator has operands of the `dynamic` type. Because `value` is `dynamic`, the C# compiler emits payload code that will examine the actual type of `value` at run time and determine what the + operator should actually do.

The first time the + operator evaluates, value contains `5` (an `Int32`) and the result is `10` (also an `Int32`). This puts this result in the `value` variable. Then, the `M` method is called, passing it `value`. For the call to `M`, the compiler will emit payload code that will, at run time, examine the actual type of the argument being passed to `M` and determine which overload of the `M` method to call. When `value` contains an `Int32`, the overload of `M` that takes an `Int32` parameter is called.

The second time the + operator evaluates, value contains "A" (a `String`) and the result is "AA" (the result of concatenating "A" with itself). Then, the `M` method is called again, passing it `value`. This time, the payload code determines that the actual type being passed to `M` is a `String` and calls the overload of `M` that takes a `String` parameter.

When the type of a field, method parameter, or method return type is specified as `dynamic`, the compiler converts this type to the `System.Object` type and applies an instance of `System.Runtime.CompilerServices.DynamicAttribute` to the field, parameter, or return type in metadata. If a local variable is specified as `dynamic`, then the variable's type will also be of type `Object`, but the `DynamicAttribute` is not applied to the local variable because its usage is self-contained within the method. Because `dynamic` is really the same as `Object`, you cannot write methods whose signature differs only by `dynamic` and `Object`.

It is also possible to use `dynamic` when specifying generic type arguments to a generic class (reference type), a structure (value type), an interface, a delegate, or a method. When you do this, the compiler converts `dynamic` to `Object` and applies `DynamicAttribute` to the various pieces of metadata where it makes sense. Note that the generic code that you are using has already been compiled and will consider the type to be `Object`; no dynamic dispatch will be performed because the compiler did not produce any payload code in the generic code.

Any expression can implicitly be cast to `dynamic` because all expressions result in a type that is derived from `Object`.[2] Normally, the compiler does not allow you to write code that implicitly casts an expression from `Object` to another type; you must use explicit cast syntax. However, the compiler does allow you to cast an expression from `dynamic` to another type by using implicit cast syntax.

```
Object o1 = 123;        // OK: Implicit cast from Int32 to Object (boxing)
Int32 n1 = o;           // Error: No implicit cast from Object to Int32
Int32 n2 = (Int32) o;   // OK: Explicit cast from Object to Int32 (unboxing)

dynamic d1 = 123;       // OK: Implicit cast from Int32 to dynamic (boxing)
Int32 n3 = d1;          // OK: Implicit cast from dynamic to Int32 (unboxing)
```

[2] And, as always, value types will be boxed.

Although the compiler allows you to omit the explicit cast when casting from dynamic to some other type, the CLR will validate the cast at run time to ensure that type safety is maintained. If the object's type is not compatible with the cast, the CLR will throw an InvalidCastException exception.

Note that the result of evaluating a dynamic expression is a dynamic expression. Examine this code.

```
dynamic d = 123;
var result = M(d);  // Note: 'var result' is the same as 'dynamic result'
```

Here, the compiler allows the code to compile because it doesn't know at compile time which M method it will call. Therefore, it also does not know what type of result M will return. And so, the compiler assumes that the result variable is of type dynamic itself. You can verify this by placing your mouse over var in the Visual Studio editor; the IntelliSense window will indicate 'dynamic: Represents an object whose operations will be resolved at runtime.' If the M method invoked at run time has a return type of void, a Microsoft.CSharp.RuntimeBinder.RuntimeBinderException exception is thrown.

Important Do not confuse dynamic and var. Declaring a local variable using var is just a syntactical shortcut that has the compiler infer the specific data type from an expression. The var keyword can be used only for declaring local variables inside a method, whereas the dynamic keyword can be used for local variables, fields, and arguments. You cannot cast an expression to var but you can cast an expression to dynamic. You must explicitly initialize a variable declared using var, whereas you do not have to initialize a variable declared with dynamic. For more information about C#'s var, see the "Implicitly Typed Local Variables" section in Chapter 9, "Parameters."

However, when converting from dynamic to another static type, the result's type is, of course, the static type. Similarly, when constructing a type by passing one or more dynamic arguments to its constructor, the result is the type of object you are constructing.

```
dynamic d = 123;
var x = (Int32) d;         // Conversion: 'var x' is the same as 'Int32 x'
var dt = new DateTime(d);  // Construction: 'var dt' is the same as 'DateTime dt'
```

If a dynamic expression is specified as the collection in a foreach statement or as a resource in a using statement, the compiler will generate code that attempts to cast the expression to the non-generic System.IEnumerable interface or to the System.IDisposable interface, respectively. If the cast succeeds, the expression is used and the code runs just fine. If the cast fails, a Microsoft.CSharp.RuntimeBinder.RuntimeBinderException exception is thrown.

Important A dynamic expression is really the same type as `System.Object`. The compiler assumes that whatever operation you attempt on the expression is legal, so the compiler will not generate any warnings or errors. However, exceptions will be thrown at run time if you attempt to execute an invalid operation. In addition, Visual Studio cannot offer any IntelliSense support to help you write code against a dynamic expression. You cannot define an extension method (discussed in Chapter 8, "Methods") that extends dynamic, although you can define one that extends Object. And, you cannot pass a lambda expression or anonymous method (both discussed in Chapter 17, "Delegates") as an argument to a dynamic method call because the compiler cannot infer the types being used.

Here is an example of some C# code that uses COM `IDispatch` to create a Microsoft Excel workbook and places a string in cell A1.

```
using Microsoft.Office.Interop.Excel;
...
public static void Main() {
   Application excel = new Application();
   excel.Visible = true;
   excel.Workbooks.Add(Type.Missing);
   ((Range)excel.Cells[1, 1]).Value = "Text in cell A1"; // Put this string in cell A1
}
```

Without the `dynamic` type, the value returned from `excel.Cells[1, 1]` is of type `Object`, which must be cast to the `Range` type before its `Value` property can be accessed. However, when producing a runtime callable wrapper assembly for a COM object, any use of `VARIANT` in the COM method is really converted to `dynamic`; this is called dynamification. Therefore, because `excel.Cells[1, 1]` is of type `dynamic`, you do not have to explicitly cast it to the `Range` type before its `Value` property can be accessed. Dynamification can greatly simplify code that interoperates with COM objects. Here is the simpler code.

```
using Microsoft.Office.Interop.Excel;
...
public static void Main() {
   Application excel = new Application();
   excel.Visible = true;
   excel.Workbooks.Add(Type.Missing);
   excel.Cells[1, 1].Value = "Text in cell A1"; // Put this string in cell A1
}
```

The following code shows how to use reflection to call a method ("Contains") on a `String` target ("Jeffrey Richter") passing it a `String` argument ("ff") and storing the `Boolean` result in a local variable (`result`).

```
Object target = "Jeffrey Richter";
Object arg = "ff";

// Find a method on the target that matches the desired argument types
Type[] argTypes = new Type[] { arg.GetType() };
MethodInfo method = target.GetType().GetMethod("Contains", argTypes);

// Invoke the method on the target passing the desired arguments
Object[] arguments = new Object[] { arg };
Boolean result = Convert.ToBoolean(method.Invoke(target, arguments));
```

Using C#'s `dynamic` type, this code can be rewritten with greatly improved syntax.

```
dynamic target = "Jeffrey Richter";
dynamic arg = "ff";
Boolean result = target.Contains(arg);
```

Earlier, I mentioned that the C# compiler emits payload code that, at run time, figures out what operation to perform based on the actual type of an object. This payload code uses a class known as a runtime binder. Different programming languages define their own runtime binders that encapsulate the rules of that language. The code for the C# runtime binder is in the `Microsoft.CSharp.dll` assembly, and you must reference this assembly when you build projects that use the `dynamic` keyword. This assembly is referenced in the compiler's default response file, CSC.rsp. It is the code in this assembly that knows to produce code (at run time) that performs addition when the + operator is applied to two `Int32` objects and concatenation when applied to two `String` objects.

At run time, the `Microsoft.CSharp.dll` assembly will have to load into the AppDomain, which hurts your application's performance and increases memory consumption. `Microsoft.CSharp.dll` also loads `System.dll` and `System.Core.dll`. If you are using `dynamic` to help you interoperate with COM components, then `System.Dynamic.dll` will also load. And when the payload code executes, it generates dynamic code at run time; this code will be in an in-memory assembly called "Anonymously Hosted DynamicMethods Assembly." The purpose of this code is to improve the performance of dynamic dispatch in scenarios where a particular call site is making many invocations using dynamic arguments that have the same runtime type.

Due to all the overhead associated with C#'s built-in dynamic evaluation feature, you should consciously decide that you are getting sufficient syntax simplification from the dynamic feature to make it worth the extra performance hit of loading all these assemblies and the extra memory that they consume. If you have only a couple places in your program where you need `dynamic` behavior, it might be more efficient to just do it the old-fashioned way, by calling reflection methods (for managed objects) or with manual casting (for COM objects).

At run time, the C# runtime binder resolves a dynamic operation according to the runtime type of the object. The binder first checks to see if the type implements the `IDynamicMetaObjectProvider` interface. If the object does implement this interface, then the interface's `GetMetaObject` method is

called, which returns a DynamicMetaObject-derived type. This type can process all of the member, method, and operator bindings for the object. Both the IDynamicMetaObjectProvider interface and the DynamicMetaObject base class are defined in the System.Dynamic namespace, and both are in the System.Core.dll assembly.

Dynamic languages, such as Python and Ruby, endow their types with DynamicMetaObject-derived types so that they can be accessed in a way appropriate for them when manipulated from other programming languages (like C#). Similarly, when accessing a COM component, the C# runtime binder will use a DynamicMetaObject-derived type that knows how to communicate with a COM component. The COM DynamicMetaObject-derived type is defined in the System.Dynamic.dll assembly.

If the type of the object being used in the dynamic expression does not implement the IDynamicMetaObjectProvider interface, then the C# compiler treats the object like an instance of an ordinary C#-defined type and performs operations on the object using reflection.

One of the limitations of dynamic is that you can only use it to access an object's instance members because the dynamic variable must refer to an object. But, there are occasions when it would be useful to dynamically invoke static members of a type where the type is determined at run time. To accomplish this, I have created a StaticMemberDynamicWrapper class that derives from System.Dynamic.DynamicObject, which implements the IDynamicMetaObjectProvider interface. The class internally uses quite a bit of reflection (covered in Chapter 23, "Assembly Loading and Reflection"). Here is the code for my StaticMemberDynamicWrapper class.

```
internal sealed class StaticMemberDynamicWrapper : DynamicObject {
   private readonly TypeInfo m_type;
   public StaticMemberDynamicWrapper(Type type) { m_type = type.GetTypeInfo(); }

   public override IEnumerable<String> GetDynamicMemberNames() {
      return m_type.DeclaredMembers.Select(mi => mi.Name);
   }

   public override Boolean TryGetMember(GetMemberBinder binder, out object result) {
      result = null;
      var field = FindField(binder.Name);
      if (field != null) { result = field.GetValue(null); return true; }

      var prop = FindProperty(binder.Name, true);
      if (prop != null) { result = prop.GetValue(null, null); return true; }
      return false;
   }

   public override Boolean TrySetMember(SetMemberBinder binder, object value) {
      var field = FindField(binder.Name);
      if (field != null) { field.SetValue(null, value); return true; }

      var prop = FindProperty(binder.Name, false);
      if (prop != null) { prop.SetValue(null, value, null); return true; }
      return false;
   }
```

```
   public override Boolean TryInvokeMember(InvokeMemberBinder binder, Object[] args,
      out Object result) {
      MethodInfo method = FindMethod(binder.Name);
      if (method == null) { result = null; return false; }
      result = method.Invoke(null, args);
      return true;
   }

      private MethodInfo FindMethod(String name, Type[] paramTypes) {
         return m_type.DeclaredMethods.FirstOrDefault(mi => mi.IsPublic && mi.IsStatic
                && mi.Name == name
            && ParametersMatch(mi.GetParameters(), paramTypes));
      }

      private Boolean ParametersMatch(ParameterInfo[] parameters, Type[] paramTypes) {
         if (parameters.Length != paramTypes.Length) return false;
         for (Int32 i = 0; i < parameters.Length; i++)
            if (parameters[i].ParameterType != paramTypes[i]) return false;
         return true;
      }

   private FieldInfo FindField(String name) {
      return m_type.DeclaredFields.FirstOrDefault(fi => fi.IsPublic && fi.IsStatic
             && fi.Name == name);
   }

   private PropertyInfo FindProperty(String name, Boolean get) {
      if (get)
         return m_type.DeclaredProperties.FirstOrDefault(
            pi => pi.Name == name && pi.GetMethod != null &&
            pi.GetMethod.IsPublic && pi.GetMethod.IsStatic);

      return m_type.DeclaredProperties.FirstOrDefault(
         pi => pi.Name == name && pi.SetMethod != null &&
            pi.SetMethod.IsPublic && pi.SetMethod.IsStatic);
   }
}
```

To invoke a static member dynamically, construct an instance of this class by passing in the `Type` you want it to operate on and put the reference in a `dynamic` variable. Then, invoke the desired static member by using instance member syntax. Here is an example of how to invoke `String`'s static `Concat(String, String)` method.

```
dynamic stringType = new StaticMemberDynamicWrapper(typeof(String));
var r = stringType.Concat("A", "B");  // dynamically invoke String's static Concat method
Console.WriteLine(r);                 // Displays "AB"
```