CHAPTER 19

Nullable Value Types

In this chapter:

As you know, a variable of a value type can never be `null`; it always contains the value type's value itself. In fact, this is why they call these types *value* types. Unfortunately, there are some scenarios in which this is a problem. For example, when designing a database, it's possible to define a column's data type to be a 32-bit integer that would map to the `Int32` data type of the Framework Class Library (FCL). But a column in a database can indicate that the value is nullable. That is, it is OK to have no value in the row's column. Working with database data by using the Microsoft .NET Framework can be quite difficult because in the common language runtime (CLR), there is no way to represent an `Int32` value as `null`.

Note Microsoft ADO.NET's table adapters do support nullable types. But unfortunately, the types in the `System.Data.SqlTypes` namespace are not replaced by nullable types, partially because there isn't a one-to-one correspondence between types. For example, the `SqlDecimal` type has a maximum of 38 digits, whereas the regular `Decimal` type can reach only 29. In addition, the `SqlString` type supports its own locale and compare options, which are not supported by the normal `String` type.

Here is another example. In Java, the `java.util.Date` class is a reference type, and therefore, a variable of this type can be set to `null`. However, in the CLR, a `System.DateTime` is a value type, and a `DateTime` variable can never be `null`. If an application written in Java wants to communicate a date/time to a web service running the CLR, there is a problem if the Java application sends `null` because the CLR has no way to represent this and operate on it.

To improve this situation, Microsoft added the concept of nullable value types to the CLR. To understand how they work, we first need to look at the `System.Nullable<T>` structure, which is defined in the FCL.

Here is the logical representation of how the System.Nullable<T> type is defined.

```
[Serializable, StructLayout(LayoutKind.Sequential)]
public struct Nullable<T> where T : struct {

   // These 2 fields represent the state
   private Boolean hasValue = false; // Assume null
   internal T value = default(T);    // Assume all bits zero

   public Nullable(T value) {
      this.value = value;
      this.hasValue = true;
   }

   public Boolean HasValue { get { return hasValue; } }

   public T Value {
      get {
         if (!hasValue) {
            throw new InvalidOperationException(
               "Nullable object must have a value.");
         }
         return value;
      }
   }

   public T GetValueOrDefault() { return value; }

   public T GetValueOrDefault(T defaultValue) {
      if (!HasValue) return defaultValue;
      return value;
   }

   public override Boolean Equals(Object other) {
      if (!HasValue) return (other == null);
      if (other == null) return false;
      return value.Equals(other);
   }

   public override int GetHashCode() {
      if (!HasValue) return 0;
      return value.GetHashCode();
   }

   public override string ToString() {
      if (!HasValue) return "";
      return value.ToString();
   }

   public static implicit operator Nullable<T>(T value) {
      return new Nullable<T>(value);
   }

   public static explicit operator T(Nullable<T> value) {
      return value.Value;
   }
}
```

As you can see, this class encapsulates the notion of a value type that can also be null. Because Nullable<T> is itself a value type, instances of it are still fairly lightweight. That is, instances can still be on the stack, and an instance is the same size as the original value type plus the size of a Boolean field. Notice that Nullable's type parameter, T, is constrained to struct. This was done because reference type variables can already be null.

So now, if you want to use a nullable Int32 in your code, you can write something like this.

```
Nullable<Int32> x = 5;
Nullable<Int32> y = null;
Console.WriteLine("x: HasValue={0}, Value={1}",  x.HasValue, x.Value);
Console.WriteLine("y: HasValue={0}, Value={1}",  y.HasValue, y.GetValueOrDefault());
```

When I compile and run this code, I get the following output.

```
x: HasValue=True, Value=5
y: HasValue=False, Value=0
```

C#'s Support for Nullable Value Types

Notice in the code that C# allows you to use fairly simple syntax to initialize the two Nullable<Int32> variables, x and y. In fact, the C# team wants to integrate nullable value types into the C# language, making them first-class citizens. To that end, C# offers a cleaner syntax for working with nullable value types. C# allows the code to declare and initialize the x and y variables to be written using question mark notation.

```
Int32? x = 5;
Int32? y = null;
```

In C#, Int32? is a synonym notation for Nullable<Int32>. But C# takes this further. C# allows you to perform conversions and casts on nullable instances. And C# also supports applying operators to nullable instances. The following code shows examples of these.

```
private static void ConversionsAndCasting() {
   // Implicit conversion from non-nullable Int32 to Nullable<Int32>
   Int32? a = 5;

   // Implicit conversion from 'null' to Nullable<Int32>
   Int32? b = null;  // Same as "Int32? b = new Int32?();" which sets HasValue to false

   // Explicit conversion from Nullable<Int32> to non-nullable Int32
   Int32 c = (Int32) a;

   // Casting between nullable primitive types
   Double? d = 5; // Int32->Double?  (d is 5.0 as a double)
   Double? e = b; // Int32?->Double? (e is null)
}
```

C# also allows you to apply operators to nullable instances. The following code shows examples of this.

```
private static void Operators() {
   Int32? a = 5;
   Int32? b = null;

   // Unary operators (+  ++  -  --  !  ~)
   a++;    // a = 6
   b = -b; // b = null

   // Binary operators (+  -  *  /  %  &  |  ^  <<  >>)
   a = a + 3;  // a = 9
   b = b * 3;  // b = null;

   // Equality operators (==  !=)
   if (a == null) { /* no  */ } else { /* yes */ }
   if (b == null) { /* yes */ } else { /* no  */ }
   if (a != b)    { /* yes */ } else { /* no  */ }

   // Comparison operators (<>  <=  >=)
   if (a < b)     { /* no  */ } else { /* yes */ }
}
```

Here is how C# interprets the operators:

- **Unary operators (+, ++, -, --, ! , ~)** If the operand is `null`, the result is `null`.
- **Binary operators (+, -, *, /, %, &, |, ^, <<, >>)** If either operand is `null`, the result is `null`. However, an exception is made when the & and | operators are operating on `Boolean?` operands, so that the behavior of these two operators gives the same behavior as demonstrated by SQL's three-valued logic. For these two operators, if neither operand is `null`, the operator performs as expected, and if both operands are `null`, the result is `null`. The special behavior comes into play when just one of the operands is `null`. The following table lists the results produced by these two operators for all combinations of `true`, `false`, and `null`.

Operand1 →Operand2 ↓	**`true`**	**`false`**	**`null`**

`True`	`& = true`
	`| = true`
		`& = false`
		`| = true`
			`& = null`
			`| = true`

`False`	`& = false`
	`| = true`
		`& = false`
		`| = false`
			`& = false`
			`| = null`

`Null`	`& = null`
	`| = true`
		`& = false`
		`| = null`
			`& = null`
			`| = null`

- **Equality operators (==, !=)** If both operands are `null`, they are equal. If one operand is `null`, they are not equal. If neither operand is `null`, compare the values to determine if they are equal.
- **Relational operators (<, >, <=, >=)** If either operand is `null`, the result is `false`. If neither operand is `null`, compare the values.

You should be aware that manipulating nullable instances does generate a lot of code. For example, see the following method.

```
private static Int32? NullableCodeSize(Int32? a, Int32? b) {
   return a + b;
}
```

When I compile this method, there is quite a bit of resulting Intermediate Language (IL) code, which also makes performing operations on nullable types slower than performing the same operation on non-nullable types. Here is the C# equivalent of the compiler-produced IL code.

```
private static Nullable<Int32> NullableCodeSize(Nullable<Int32> a, Nullable<Int32> b) {

   Nullable<Int32> nullable1 = a;
   Nullable<Int32> nullable2 = b;
   if (!(nullable1.HasValue & nullable2.HasValue)) {
      return new Nullable<Int32>();
   }
   return new Nullable<Int32>(nullable1.GetValueOrDefault() + nullable2.GetValueOrDefault());
}
```

Finally, let me point out that you can define your own value types that overload the various operators previously mentioned. I discuss how to do this in the "Operator Overload Methods" section in Chapter 8, "Methods." If you then use a nullable instance of your own value type, the compiler does the right thing and invokes your overloaded operator. For example, suppose that you have a `Point` value type that defines overloads for the == and != operators as follows.

```
using System;

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

   public static Boolean operator==(Point p1, Point p2) {
      return (p1.m_x == p2.m_x) && (p1.m_y == p2.m_y);
   }

   public static Boolean operator!=(Point p1, Point p2) {
      return !(p1 == p2);
   }
}
```

At this point, you can use nullable instances of the `Point` type and the compiler will invoke your overloaded operators.

```
internal static class Program {
   public static void Main() {
      Point? p1 = new Point(1, 1);
      Point? p2 = new Point(2, 2);

      Console.WriteLine("Are points equal? " + (p1 == p2).ToString());
      Console.WriteLine("Are points not equal? " + (p1 != p2).ToString());
   }
}
```

When I build and run the preceding code, I get the following output.

```
Are points equal? False
Are points not equal? True
```

C#'s Null-Coalescing Operator

C# has an operator called the *null-coalescing operator (??)*, which takes two operands. If the operand on the left is not `null`, the operand's value is returned. If the operand on the left is `null`, the value of the right operand is returned. The null-coalescing operator offers a very convenient way to set a variable's default value.

A cool feature of the null-coalescing operator is that it can be used with reference types as well as nullable value types. Here is some code that demonstrates the use of the null-coalescing operator.

```
private static void NullCoalescingOperator() {
   Int32? b = null;

   // The following line is equivalent to:
   // x = (b.HasValue) ? b.Value : 123
   Int32 x = b ?? 123;
   Console.WriteLine(x);  // "123"

   // The following line is equivalent to:
   // String temp = GetFilename();
   // filename = (temp != null) ? temp : "Untitled";
   String filename = GetFilename() ?? "Untitled";
}
```

Some people argue that the null-coalescing operator is simply syntactic sugar for the `?:` operator, and that the C# compiler team should not have added this operator to the language. However, the null-coalescing operator offers two significant syntactic improvements. The first is that the ?? operator works better with expressions.

```
Func<String> f = () => SomeMethod() ?? "Untitled";
```

This code is much easier to read and understand than the following line, which requires variable assignments and multiple statements.

```
Func<String> f = () => { var temp = SomeMethod();
  return temp != null ? temp : "Untitled";};
```

The second improvement is that ?? works better in composition scenarios. For example, the following single line:

```
String s = SomeMethod1() ?? SomeMethod2() ?? "Untitled";
```

is far easier to read and understand than this chunk of code.

```
String s;
var sm1 = SomeMethod1();
if (sm1 != null) s = sm1;
else {
   var sm2 = SomeMethod2();
   if (sm2 != null) s = sm2;
   else s = "Untitled";
}
```

The CLR Has Special Support for Nullable Value Types

The CLR has built-in support for nullable value types. This special support is provided for boxing, unboxing, calling `GetType`, calling interface methods, and it is given to nullable types to make them fit more seamlessly into the CLR. This also makes them behave more naturally and as most developers would expect. Let's take a closer look at the CLR's special support for nullable types.

Boxing Nullable Value Types

Imagine a `Nullable<Int32>` variable that is logically set to `null`. If this variable is passed to a method prototyped as expecting an `Object`, the variable must be boxed, and a reference to the boxed `Nullable<Int32>` is passed to the method. This is not ideal because the method is now being passed a non-`null` value even though the `Nullable<Int32>` variable logically contained the value of `null`. To fix this, the CLR executes some special code when boxing a nullable variable to keep up the illusion that nullable types are first-class citizens in the environment.

Specifically, when the CLR is boxing a `Nullable<T>` instance, it checks to see if it is `null`, and if so, the CLR doesn't actually box anything, and `null` is returned. If the nullable instance is not `null`, the CLR takes the value out of the nullable instance and boxes it. In other words, a `Nullable<Int32>` with a value of 5 is boxed into a boxed-`Int32` with a value of 5. Here is some code that demonstrates this behavior.

```
// Boxing Nullable<T> is null or boxed T
Int32? n = null;
Object o = n;  // o is null
Console.WriteLine("o is null={0}", o == null);  // "True"

n = 5;
o = n;   // o refers to a boxed Int32
Console.WriteLine("o's type={0}", o.GetType()); // "System.Int32"
```

Unboxing Nullable Value Types

The CLR allows a boxed value type T to be unboxed into a T or a Nullable<T>. If the reference to the boxed value type is null, and you are unboxing it to a Nullable<T>, the CLR sets Nullable<T>'s value to null. Here is some code to demonstrate this behavior.

```
// Create a boxed Int32
Object o = 5;

// Unbox it into a Nullable<Int32> and into an Int32
Int32? a = (Int32?) o;  // a = 5
Int32  b = (Int32)  o;  // b = 5

// Create a reference initialized to null
o = null;

// "Unbox" it into a Nullable<Int32> and into an Int32
a = (Int32?) o;       // a = null
b = (Int32)  o;       // NullReferenceException
```

Calling GetType via a Nullable Value Type

When calling GetType on a Nullable<T> object, the CLR actually lies and returns the type T instead of the type Nullable<T>. Here is some code that demonstrates this behavior.

```
Int32? x = 5;

// The following line displays "System.Int32"; not "System.Nullable<Int32>"
Console.WriteLine(x.GetType());
```

Calling Interface Methods via a Nullable Value Type

In the following code, I'm casting n, a Nullable<Int32>, to IComparable<Int32>, an interface type. However, the Nullable<T> type does not implement the IComparable<Int32> interface as Int32 does. The C# compiler allows this code to compile anyway, and the CLR's verifier considers this code verifiable to allow you a more convenient syntax.

```
Int32? n = 5;
Int32 result = ((IComparable) n).CompareTo(5);  // Compiles & runs OK
Console.WriteLine(result);                      // 0
```

If the CLR didn't provide this special support, it would be more cumbersome for you to write code to call an interface method on a nullable value type. You'd have to cast the unboxed value type first before casting to the interface to make the call.

```
Int32 result = ((IComparable) (Int32) n).CompareTo(5);  // Cumbersome
```