CHAPTER 27

Nullable Types

Null values are a useful programming construct beyond simply recording the initial state of a class variable. In the database world, null conveys a notion of "no value recorded." In a survey, the answer to a question may be *yes* or *no*, but if a response isn't provided, isn't available, or isn't relevant to a particular respondent, you can record their response as *null* rather than a definite *yes* or *no*. Because many applications deal with data that's stored in databases, where null values can be quite common, the ability to express null values becomes quite important.

The inability to store nulls in value types can be a significant problem in C#. Value types, by definition, consist of bits that either are allocated on the stack when they're declared as a local variable or are allocated inline when they form part of heap-allocated types. Because a value type is always physically allocated, it has no ability to express an unallocated or null value type. Further, since value types don't have the luxury of an object header (like reference types do), they don't have a spare header bit that can indicate an instance is null.

Dealing with the inability to express nullability becomes particularly troublesome when dealing with data that originates from databases. Databases typically don't have separate concepts for reference and value types, and all data types can be null if the schema allows. Before nullable types, recording null values in value types was a difficult task. The following options existed:

* You could choose a special value to indicate logical null. For signed integers, `-1` was often a reasonable choice; for floating points, NaN was popular. Finally, for dates, `DateTime.MinValue` was often used.
* You could use a reference type that stored the value type internally and leave it as null when appropriate. The types in `System.Data.SqlTypes` and various third-party libraries were used in this role.
* You could maintain a boolean variable for every value type variable that could potentially be null. The boolean would keep track of whether the accompanying value type was logically null.

All of these alternatives had problems. Special values were often hard to define and had the unfortunate tendency to end up back in the database if a developer wasn't careful with the update logic. This in turn created the painful problem of catering to both physical and logical nulls. Using reference types was effective, but this data-tier local implementation decision had a tendency of spreading through all tiers of an application, which meant that the performance benefits offered by value types weren't available. The third alternative was generally the cleanest and simplest option, but it was cumbersome to implement, particularly with method return values, and it also wasted space.

Because developers needed a general-purpose solution to value type nullability, and because the designers of C# 2.0 were empowered by the addition of generics, they extended the framework library to include the `Nullable<T>` value type. `Nullable<T>` can take any value type as a type parameter, and it internally stores a boolean to track logical nullness. This approach is essentially the same as the third option from the previous list but is simpler from a logistical point of view, as only a single variable needs to be tracked.

`Nullable<T>` has various retrieval and informational methods:

```
Nullable<int> i = null;
bool b = i.HasValue; //will be false
int j = Nullable.GetValueOrDefault<int>(i); //returns 0
//j = i.Value; //will throw an exception
//j = (int)i; //will throw an exception
i = 123;
j = Nullable.GetValueOrDefault<int>(i); //returns 123
j = (int)123; //works fine
```

As you can see in the example, `Nullable<T>` allows both `null` and values that are legal for the type parameter to be assigned to a `Nullable<T>` instance. You can use the `GetValueOrDefault` method to retrieve the value, with the default value for the value type returned if the instance is null. You can also use a cast to convert to a non-nullable instance, but an exception will occur if the cast instance is logically null.

C# Language Nullable Types

The designers of the C# language figured the ability to convey nullability in value types was important enough to elevate nullable types from a generic type in the framework library into a fully fledged language feature. C# language nullable types map directly to the `Nullable<T>` framework library type, so any other language that supports generics can use C# nullable types. You express nullable types by adding a question mark after the name of a value type, which indicates that the instance declared by the declaration can be nullable:

```
int? i = null; //same as Nullable<int> i = null;
int? j = 0; // same as Nullable<int> j = 0;
```

To be a genuinely useful feature, nullable types need to act and feel like the value type they represent. Thankfully, operations and conversion that work on the value type will also work on the reference type. This means the following code will work with no compile-time or runtime errors:

```
int? i = 1;
int? j = 2;
int? k = i + j;  //addition operator works fine
double? d = k; //no problem with implicit conversion to double
short? s = (short?)d; //explicit cast required to short?
                      //because of possible data loss
```

For the case where none of the values in the equation is null, life is simple: the operations and conversions behave in the same way as they would have if the values hadn’t been nullable types. When null values are present, the situation becomes a little more interesting. The basic rule is that the presence of a null value anywhere in the chain results in a null result. Take the following example:

```
int? i = 1;
int? h = 2;
int? j = null;
int? k = i + j + h + 3;
double? d = k;
short? s = (short?)d;
```

In this case, the fact that `j` is null and is used in the addition results in `k`, `d`, and `s` ending up null.

User-defined operators work in much the same way as the built-in operators. If no null values are present, the user-defined operations work in the same way as they would with the non-nullable form of the types. If one or more null values are present anywhere in the chain of operations, the result of the user-defined operation will be null.

SQL Language Differences and Similarities

In the C language, two null pointers are equal, as pointer comparison is simply a comparison of two integral values. C# follows this pattern, and for equality operations, two null values *are* defined as being equal, a behavior that's in contrast to SQL logic where null isn't equal to null but instead *is* null.

During the C# 2.0 design process, the designers discussed this decision at length and considered many options. One option was to support both the programming and database versions of equality (`null == null` and `null != null`), but this would have required an additional set of operators. The C# design team ultimately decided that approach would add too much complexity. They decided using the database version of equality in code would be surprising to the majority of C# programmers.

The following code highlights the different behavior between the two languages:

```
//C#
int? i = null;
int? j = null;
bool b = i == j;
//b with be true


--T_SQL
declare @i int
declare @j int
select @i = null
select @j = null

declare @b bit
if @i = @j
 select @b = 1
else
 select @b = 0
--@b will be zero.
```

In contrast to the difference in behavior between C# and SQL nullable types, significant efforts have been made to ensure compatibility in several areas. The nullable `bool` type has predefined operators for bitwise AND (`&`) and OR (`|`) operations that ensure it has the same behavior as the equivalent SQL operations. This means a bitwise operation on a `bool?` instance doesn't necessary result in a null result, even if one of the inputs into the operation is null. Clearly, a major exception to the main rule exists—a null anywhere in a calculation chain results in the outcome of the calculation being null. The following are a few examples:

```
bool? i = true;
bool? j = false;
bool? k = null;

bool? res1 = i | k; //returns true, NOT null
bool? res2 = j & k; //returns false, NOT null
```

Null Coalescing Operator

Just as the C# language provides a more streamlined syntax for expressing nullable types, a more streamlined syntax exists for providing a default alternative if a nullable type doesn't hold a value. Rather than using the `HasValue` method and the `?:` ternary conditional operator to provide a default alternative, you can use the null coalescing operator (`??`)—simply place the default value after the `??` operator:

```
int? x = null;
int y = x ?? 2; //x is null, y will be 2
//same as:
y = x.HasValue ? x. GetValueOrDefault() : 2;

x = 3;
y = x ?? 2; //x has a value and it will be returned, y will be 3
```

You have no need to cast the nullable type instance when using the coalescing operator; the compiler will generate a call to `GetValueOrDefault`, which returns an instance of the non-nullable value type.

Design Guidelines

When user interfaces and data tiers are written to present and manage data that's persisted to a database, nullable types make the exercise of dealing with value types that can be null much easier. To achieve the smoothest development process with database code, one of the essential elements is a consistent and reliable approach for mapping database types to C# types and dealing with the changes, additions, and deletions in these types that inevitably occur during the development cycle.

Because nullability is one of a number of attributes of any particular database field, and because dealing with changes in nullability is no different from dealing with name and data type changes, the idea of making a value type nullable in C# when it isn't nullable in the database just in case the underlying database schema changes makes little sense. If the nullability of a field changes, simply use the same established change process you'd use if an `int` field changed to a `long` field. Using nullable types unnecessarily is detrimental to performance—memory is wasted storing a boolean value for each nullable type and processor cycles are wasted checking this boolean when calculations involving the nullable type are performed.

Don't mix nullability concepts within an assembly. If you're currently using special values, `System.Data.SqlTypes` types, or third-party nullability libraries to deal with nullability, don't introduce `Nullable<T>` or C# nullable types without migrating existing code to use these new features.