When Dynamic Meets Nullable

By Bill Wagner
November 2012

Introduction

This column discusses interplay between the rules for the dynamic type and the rules for nullable types. This interplay caused quite a bit of confusion and extra work on a large app because the team did not have a complete understanding of how these different parts of the language interact. Part of this large application read input from a XML file whose layout made it difficult to parse and use. There was an XML attribute that could be a string, and integer, or a double, or missing entirely. To support missing attributes, the application used nullable types to represent the numeric value. Strings were used for string attribute. Another part of the application used those attributes to produce new items as text. The developers involved wrote a method that took parsed the attribute (as a parameter of type dynamic) and produced the text.

The problems they incurred are illustrated in this small program:

```
class Program

{

  public static string DisplayAttribute(dynamic attributeValue)

  {

    return attributeValue.ToString();

  }


  static void Main()

  {

    int first = 5;

    int? second = 15;

    int? third = default(int?);


    Console.WriteLine(first);

    Console.WriteLine(second);

    Console.WriteLine(third);
```

```
        Console.WriteLine("Second Test");
        Console.WriteLine(DisplayAttribute(first));
        Console.WriteLine(DisplayAttribute(second));
        Console.WriteLine(DisplayAttribute(third));
    }
}
```

What do you suppose the output is from this program? The first three lines are easy: 5, 15, and a blank line. The last three lines are bit trickier. You'll get 5, 15, and then the program throws an exception and exits. The rest of this column will discuss the language rules that define this behavior, and techniques that would help you avoid running afoul of this interplay.

Dynamic is System.Object and Metadata

The method throws an exception because of how two different types interact. There is no CLR type "dynamic". The dynamic behavior is implemented by classes in the framework. Runtime binders perform the dynamic lookup on variables of type dynamic. Those binders will inspect the type information on an object and determine how to resolve an operation on a dynamic object.

Language compilers (including the C# compiler) must generate the extra code that examines the object referenced by a dynamic variable and invoke the operation chosen by the runtime binder. Those variables that you called "dynamic" in your program must be represented by some real CLR type in IL. The C# compiler generates code for variables of type 'dynamic' as though they were given the type of System.Object. That means that dynamic values also obey some of the rules of System.Object.

The above description simplifies much of the work done to make dynamic variables behave as defined in the C# spec. As a C# developer, you need to remember a few simple items about programming with dynamic variables:

1. Variables of type dynamic are represented as System.Object in IL. They have additional metadata attributes associated with them, but they are System.Object. There is no dynamic type in the CLR.
2. The C# compiler generates all the code to determine what member to invoke on a dynamic object at runtime based on the member your program accesses.
3. The code generated by the C# compiler is almost certainly more performant than similar code you would generate to using reflection to create the dynamic invocation yourself.

Armed with this knowledge, we're ready to look at the other side of the problem demonstrated in the sample above.

Nullable types and null references

Nullable types exhibit some special conversion behavior, both to and from their underlying value types and for how they represent null. All of these operations make nullables behave more like their underlying type, but add the capability of storing the additional value "null". For example, the int type can store all values of integer numbers from int.MinValue through int.MaxValue. The int? type can store all values of integers numbers from int.MinValue through int.MaxValue, and can represent the absence of a value (often represented by null).

That statement about storing 'null' above is a bit misleading, although a useful abstraction. int? is a value type, and therefore variables of the type int? are not actually null references. Instead, the language supports conversions to and from the special value 'null' when nullable types are converted to and from reference types.

There is an implicit conversion from any value type to its nullable type (e.g. from int to int?, or from long to long?). There is also an explicit conversion from a nullable type to its underlying value type (e.g. from int? to int). That conversion is explicit because it is a narrowing conversion. You can see from these conversions that one goal for nullable types is to make them behave as much like their underlying types as possible. Those conversions make it easier to create interactions between nullable types and their underlying value types.

The boxing conversion behavior for nullables also obeys this rule that a nullable type should behave as much like its underlying type as possible. When a nullable type is boxed, the CLR automatically boxes the value of the underlying value type. In other words, the boxing conversion for a nullable int actually examines the underlying integer value stored in the int?, and then boxes that int. This behavior means that the boxed nullable automatically implements every interface supported by the underlying value type.

Of course, the boxing conversion can't do that when the nullable type does not contain a value. In those cases, the boxing conversion results in a null reference.

Now, you can see why the original sample didn't work as expected. Passing a variable that is a nullable type to a method whose formal parameter is of type dynamic requires that the nullable type be boxed in order to represent it as a System.Object. In the case where the HasValue property is true, the nullable type boxes the underlying type. Everything is fine. However, when the nullable type does not contain a value, the boxing conversion results in null. The code in the sample passes null to a method which does not expect that it would ever receive a null parameter.

Language Features Interact

The developer that first showed me this code was very surprised by what it does. That team had written methods with dynamic parameters before with no problems: as long as the actual types used had the expected members, the dynamic support in C# was exactly what was needed.

The surprising part in this code is caused by the fact that value types are boxed when they are converted to reference types. This is further complicated by the fact that nullable types have some special boxing behavior in order to make nullables act like their underlying value types, with the ability to represent a missing value. That combination breaks a fundamental assumption these developers had about converting objects to the dynamic type: they assumed that no conversions take place to treat a plain old CLR object as a dynamic type. They'd forgotten that all value types must be converted to a reference type using a boxing conversion in order to be used this way. They'd never learned that dynamic variables are represented as System.Object in IL. Their incomplete knowledge of the language features meant that they reached an incorrect conclusion about how a particular piece of code would work.

Overall, the C# language does have consistent behavior. Its rules govern all the features, and clearly define how different features of the language interact. As a professional, you owe it to yourself to understand the language specification enough to have some basic understanding of the entire language, and how it's features interaction. The more of an understanding you have, the less likely you'll be surprised by situations like this. You'll develop designs that are more likely to behave correctly.