The default value for a XAML dependency property should be immutable

devblogs.microsoft.com/oldnewthing/20191002-00

October 2, 2019

Raymond Chen

When you define a custom XAML dependency property, one of the things you do is specify the default value. You can do this by providing the default value directly:

```
DependencyProperty.Register("MyProperty",
    propertyType, ownerType,
    new PropertyMetadata(defaultValue));
```

UWP XAML also lets you provide a callback that produces the default value:

```
DependencyProperty.Register("MyProperty",
    propertyType, ownerType,
    PropertyMetadata.Create(createDefaultValue));
```

What's not entirely obvious is that the default value statically provided at registration or provided dynamically at runtime should be an immutable object, if you know what's good for you.

The default value of a dependency property is the value used if no value has been set: Nobody has explicitly assigned a value, there is no active binding, no active animation, no active style, nothing.

When XAML needs the default value of the dependency property, it uses the static default property, if provided. Otherwise, it calls the `createDefaultValue` delegate, if provided. Otherwise, it uses `null`.

For concreteness, let's say that we have a `Widget` object and a dependency property of type `Light`.

Let's first look at the case where you provide a static mutable value for the default value of the dependency property.

```
// Code in italics is wrong.
class Light
{
  public Color Color { get; set; }; // read-write property
}

class Widget
{
 ...

 public Widget()
 {
  InitializeComponent();
 }

 // The default is a red light.
 public static readonly DependencyProperty FrontLightProperty =
    DependencyProperty.Register("FrontLight",
    typeof(Light), typeof(Widget),
    new PropertyMetadata(new Light { Color = Color.Red }));

 // Provide convenient access to the dependency property.
 public Light FrontLight {
  get => (Light)GetValue(FrontLightProperty);
  set => SetValue(FrontLightProperty, value);
 }
}

void Example()
{
  var a = new Widget();
  var b = new Widget();

  var aColor = a.FrontLight.Color; // aColor is Red
  var bColor = b.FrontLight.Color; // bColor is Red

  a.FrontLight.Color = Color.Blue;
  var bColor2 = b.FrontLight.Color; // bColor2 is Blue (!)
}
```

Since no custom value of `FrontLight` has been set, `a.FrontLight` and `b.FrontLight` both contain the default value, which is the red `Light` provided to the `PropertyMetadata` constructor.

But since `Light` objects are mutable, the properties of that default value can be changed, and those changes are visible to both `a` and `b`, because `a.FrontLight` and `b.FrontLight` are the same object.

“Aha,” you may think, “that’s where the callback version comes in handy.” If you set up the callback to produce a unique object, then `a.FrontLight` and `b.FrontLight` will be different objects, and they won’t affect each other.

```
// Code in italics is wrong.
class Widget
{
 ...

 public Widget()
 {
  InitializeComponent();
 }

 // The default is a red light.
 public static readonly DependencyProperty FrontLightProperty =
    DependencyProperty.Register("FrontLight",
    typeof(Light), typeof(Widget),
    PropertyMetadata.Create(() => new Light { Color = Color.Red }));

 // Provide convenient access to the dependency property.
 public Light FrontLight {
  get => (Light)GetValue(FrontLightProperty);
  set => SetValue(FrontLightProperty, value);
 }
}

void Example()
{
  var aColor = a.FrontLight.Color; // aColor is Red
  var bColor = b.FrontLight.Color; // bColor is Red

  a.FrontLight.Color = Color.Blue;
  var bColor2 = b.FrontLight.Color; // bColor2 is Red

  var aColor2 = a.FrontLight.Color; // aColor2 is Red (!)

  if (a.FrontLight != a.FrontLight) TheWorldHasGoneMad(); // executes!
}
```

This time, we have the callback create a brand new red `Light`. When we evaluate `a.FrontLight.Color`, the first thing that XAML needs to do is obtain the value of `a.FrontLight`, and since nobody has set a value, XAML asks the callback to produce the default. The callback creates a fresh red `Light` and returns it, and that brand new light is the value of the `a.FrontLight` property. The `Example` function reads the color and gets `Red`.

The same thing happens when you read the color from `b.FrontLight.Color`.

So far so good. The `a` and `b` objects each have separate `FrontLight` lights.

Okay, now things get interesting: When we try to change the color of `a`'s `FrontLight` to `Blue`, the assignment succeeds, and it doesn't affect the color of `b`'s `FrontLight`, but when we try to read the color of `a`'s `FrontLight`, we get `Red` again. What happened?

In the assignment `a.FrontLight.Color = Color.Blue`, the first thing that happens is the fetch of the `FrontLight` property of the `a` object. And since we *still* haven't set a value for the `FrontLight` property, XAML uses the default value. And XAML does that by calling the callback, and the callback creates a brand new `Light`. Until a value for the `FrontLight` property is set, *every attempt to read the `FrontLight` property will produce a new red `Light`*. In other words, the default value is not cached. XAML calls the callback each time it needs the default value.

The code sets the color of this brand new light to `Blue`. But that brand new light isn't being saved anywhere. It doesn't become the value of the `FrontLight` property. It ends up orphaned, waiting for the next GC to clean it up.

Later, you read `a.FrontLight.Color`. Again, the first step is to retrieve the `FrontLight` property from `a`, and since no value has been set yet, XAML asks the callback to produce the default value, and the callback creates a brand new red `Light`. Every fetch of the `FrontLight` property produces a brand new red `Light`.

This also explains why the final statement reports that the world has gone mad: The two reads of the `FrontLight` property each produced a different red `Light`.

Producing a brand new object in response to the default value callback is a bad idea.

The intended purpose of the callback is to let you have a default value that is itself a dependency object. Since dependency objects have thread affinity, you cannot use the same object for all callers, because the callers may come from different threads. The callback lets you look in some per-thread storage to obtain the correct object for that thread. In the worst case, you can have your own private cache of objects indexed by thread.[1]

Okay, but what if the dependency property's type is a mutable type. One solution is to make `null` the default value of the property. As immutable things go, `null` looks quite immutable. Everybody who wants to change the `FrontLight` color must first provide a `FrontLight`. But what if you want the default value to be non-null? We'll look at this next time.

[1] A cache with a bad policy is another name for a memory leak. One way to avoid this problem is to put weak references in the cache, so that the objects remain alive only as long as somebody is observing them. Scavenge dead slots in the dictionary periodically to clean out the tombstones. For example, you might scavenge after a certain number of objects have been added to the cache.

Raymond Chen

Follow