Custom Formatters

When formatting data into a user-readable format we tend to use quick one-off solutions. This is a shame because Foundation comes with NSFormatter, which is perfectly suited for this task and can be easily reused throughout your code base. Heck, if you’re on a Mac, AppKit classes have built-in support for NSFormatter, making your life a lot easier.

Built-in Formatters

Foundation comes with the abstract NSFormatter class and two concrete subclasses: NSNumberFormatter and NSDateFormatter. We’re going to skip these and jump right into the deep end, implementing our own subclass.

If you need a more subtle introduction, I recommend reading this [NSHipster post](http://nshipster.com/nsformatter/).

Introduction

NSFormatter by itself doesn’t do anything, except throw errors. I have yet to find a programmer who wants this, but if such a thing tickles your fancy, go for it!

Because we don’t like errors, we’ll implement an NSFormatter subclass that can transform instances of UIColor to a human-readable name. For example, the following code will return the string “Blue”:

KPAColorFormatter \*colorFormatter = [[KPAColorFormatter alloc] init]; [colorFormatter stringForObjectValue:[UIColor blueColor]] // Blue

Two methods are required when implementing a NSFormatter subclass:stringForObjectValue: and getObjectValue:forString:errorDescription:. We’ll start of with the first because that’s the one you’ll use most often. The second is, as far as I know, most often used in OS X and actually not very useful. More on that later.

The Initializer

Hold your horses, as first we need to do some setup. There is no pre-defined mapping from colors to their names, so we need to define this. For the sake of simplicity, this will be our initializer:

- (id)init; { return [self initWithColors:@{ [UIColor redColor]: @"Red", [UIColor blueColor]: @"Blue", [UIColor greenColor]: @"Green" }]; }

Our ‘known’ colors are a dictionary keyed by a UIColor with the English name as values. I’ll leave the implementation of the initWithColors: method to your imagination. Or, if you’re that person who looks at the answers on the last page of the puzzle book, go ahead and take a look at the [Github repo](https://github.com/klaaspieter/KPAColorFormatter).

Formatting Object Values

The first thing we need to do in stringForObjectValue: is verify that the value is of the expected class. We can only format UIColors so this is the start of our method:

- (NSString \*)stringForObjectValue:(id)value; { if (![value isKindOfClass:[UIColor class]]) { return nil; } // To be continued... }

After we’ve verified that the value is what we expect it to be, we can do the real magic. Recall that our formatter has a dictionary of color names keyed by their color. To make it work, all we need to do is look up the name using the color value as key:

- (NSString \*)stringForObjectValue:(id)value; { // Previously on KPAColorFormatter return [self.colors objectForKey:value]; }

This is the simplest implementation possible. A more advanced (and useful) formatter would also be able to look up color names that don’t exist in our dictionary by finding the closest known color. I’ll leave that as an exercise to the reader. Or, if you don’t work out much, take a look at the [Github repo](https://github.com/klaaspieter/KPAColorFormatter).

Reverse Formatting

Any formatter should also support reverse formatting from a string back to an instance of the class. This is done usinggetObjectValue:forString:errorDescription:. The reason for this is that on OS X, formatters are often used in combination with NSCells.

NSCell’s have a objectValue property. By default, NSCell will use the objectValue’s description, but it can optionally use a formatter. In the case of NSTextFieldCell, a user can also enter a string value and you, as the programmer, now can expectobjectValue to be an instance of UIColor that represents that string value. In our case, the user could enter “Blue” and we should return, by reference, a [UIColor blueColor] instance.

There are two parts to implementing reverse formatting: the part where the formatter can successfully transform the string value into an instance of UIColor, and one where it cannot. Let’s start with the happy path:

- (BOOL)getObjectValue:(out \_\_autoreleasing id \*)obj forString:(NSString \*)string errorDescription:(out NSString \*\_\_autoreleasing \*)error; { \_\_block UIColor \*matchingColor = nil; [self.colors enumerateKeysAndObjectsUsingBlock:^(UIColor \*color, NSString \*name, BOOL \*stop) { if([name isEqualToString:string]) { matchingColor = color; \*stop = YES; } }]; if (matchingColor) { \*obj = matchingColor; return YES; } // Snip

There is some optimization that can be done here, but let’s not do that prematurely. This enumerates through every object in our colors dictionary and when a name is found it will return the color instance associated with it by reference. We also returnYES to notify the caller that we were able to turn the string back into an object.

Now the error path:

if (matchingColor) { // snap } else if (error) { \*error = [NSString stringWithFormat:@"No known color for name: %@", string]; } return NO;

If we can’t find a matching color we check if the caller is interested in errors, and if so, return it by reference. The check for error here is important. If you don’t do this you *will* crash. We also return NO to notify the caller that conversion was not successful.

Localization

That’s it! We have a fully functional NSFormatter subclass, for English speakers, living in the United States.

That’s about [319 million people](http://www.wolframalpha.com/input/?i=population+of+the+united+states) compared to [7.13 billion](http://www.wolframalpha.com/input/?i=population+of+the+world) in the entire world. In other words [96 percent](http://www.wolframalpha.com/input/?i=ratio+of+population+of+the+United+States+vs+population+of+the+world) of your potential users aren’t impressed, yet. Of course I hear you say: most of those don’t own iPhones or Macs and it’s really a much smaller number. Where’s the fun in that? Party pooper!

If you take a look at NSNumberFormatter and NSDateFormatter you’ll see that both have a locale property taking an instance of NSLocale. Let’s extend our formatter to support this and return a translated name based on the locale.

Translating

The first thing we need to do is translate the color strings. Messing with genstring, and \*.lprojs is outside of the scope of this article. I mean, you have to get back to work at some point right? If not, [there](https://developer.apple.com/internationalization/) [are](http://nshipster.com/nslocalizedstring/) [a](https://developer.apple.com/library/ios/documentation/MacOSX/Conceptual/BPInternational/BPInternational.html#//apple_ref/doc/uid/10000171i) [bunch](http://www.tethras.com/apple) [of](http://www.ibabbleon.com/iphone_app_localization.html) [articles](http://www.getlocalization.com/library/get-localization-mac/) [about](http://www.daveoncode.com/2010/05/15/iphone-applications-localization/) [it](http://useyourloaf.com/blog/2010/12/15/localize-iphone-application-name.html). Read them all? Awesome! No more work, time to go home.

Localized Formatting

Back to implementing localization. After we have access to the translated strings, we need to update stringForObjectValue: so it knows where to get the translation. Those who have worked with NSLocalizedString before probably already went ahead and replaced every string with NSLocalizedString. Wrong!

We are dealing with a dynamic locale here and NSLocalizedString will only find the translation for the current default language. Now in 99 percent of the cases, this is probably what you want, hence the default behavior, but we need to look up the language dynamically using the locale set on our formatter.

The new implementation of stringForObjectValue:

- (NSString \*)stringForObjectValue:(id)value; { // Previously on... don't you hate these? I just watched that 20 seconds ago! NSString \*languageCode = [self.locale objectForKey:NSLocaleLanguageCode]; NSURL \*bundleURL = [[NSBundle bundleForClass:self.class] URLForResource:languageCode withExtension:@"lproj"]; NSBundle \*languageBundle = [NSBundle bundleWithURL:bundleURL]; return [languageBundle localizedStringForKey:name value:name table:nil]; }

This leaves room for some refactoring, but bear with me. It’s easier to read if all the code is in the same place.

We first find the language code for the locale, after which we look up the NSBundlefor that language. We then ask the bundle to give us the translation of the English name. If it can’t be found, the name: argument (our English name) will be returned as a fallback value. In case you’re interested, this is exactly what NSLocalizedStringdoes except that we’re dynamically looking up the bundle.

Localized Reverse Formatting

That leaves us with reverse formatting from a translated color name back to color instances. Honestly, I don’t think it’s worth it. Our current implementation works perfect for probably 99 percent of the cases. In the other one percent where you are using this formatter on a Mac, on an NSCell, and you’re allowing your users to enter color names which you’re going to try and parse, you’ll need something a lot more complicated than a simple NSFormatter subclass. Also, you probably shouldn’t be letting your users enter colors using text. [NSColorPanel](https://developer.apple.com/library/mac/documentation/Cocoa/Reference/ApplicationKit/Classes/NSColorPanel_Class/) is a much better solution.

Attributed Strings

Now that our formatter does what we want it to do, let’s implement something completely useless. You know, because we can.

Formatters also have support for attributed strings. I find that whether or not you want to use that support really depends on your specific application and its user interface. Therefore, you will probably want to make that part of your formatter configurable.

For ours, I want to set the text color to the color that we’re formatting. This is what that looks like:

- (NSAttributedString \*)attributedStringForObjectValue:(id)value withDefaultAttributes:(NSDictionary \*)defaultAttributes; { NSString \*string = [self stringForObjectValue:value]; if (!string) { return nil; } NSMutableDictionary \*attributes = [NSMutableDictionary dictionaryWithDictionary:defaultAttributes]; attributes[NSForegroundColorAttributeName] = value; return [[NSAttributedString alloc] initWithString:string attributes:attributes]; }

First we format the string like normal, after which we check that formatting was successful. Then we merge the default attributes with our foreground color attribute. Finally, we return the attributed string. Simple, right?

Convenience

Because initializing built-in formatters [is slow](https://twitter.com/ID_AA_Carmack/status/28939697453), it has become common practice to also expose a convenience class method on your formatter. The formatter should use same defaults and the current locale. This is the implementation for our formatter:

+ (NSString \*)localizedStringFromColor:(UIColor \*)color; { static dispatch\_once\_t onceToken; dispatch\_once(&onceToken, ^{ KPAColorFormatterReusableInstance = [[KPAColorFormatter alloc] init]; }); return [KPAColorFormatterReusableInstance stringForObjectValue:color]; }

Unless your formatter is doing really crazy formatting, like NSNumberFormatter andNSDateFormatter, you probably don’t need this for performance reasons. But it’s still good to do because it makes using your formatter easier.

Wrapping Up

Our color formatter can now format a UIColor instance into a human-readable name and the other way around. There is a lot more that NSFormatter can do that we haven’t covered yet. Especially on the Mac where, because of its integration withNSCell, you can use more advanced stuff, like specifying and validating a string representation while the user is editing.

There is also more we can do to make our formatter more customizable. For example, my implementation will attempt to look up the closest color name if there is no direct match. Sometimes you may not want this so our formatter should have a Boolean property to control this behavior. Similarly, our attributed string formatting might not be what you want either and should also be more customizable.

That said, we ended up with a pretty solid formatter. All the code (with an OS X example) is available on [Github](https://github.com/klaaspieter/KPAColorFormatter). My implementation is also available on [CocoaPods](http://cocoapods.org/). If you desperately need this functionality in your app, add pod "KPAColorFormatter"to your Podfile and have fun experimenting!