Sample Code

Building a Localized Food-Ordering App

Format, style, and localize your app's text for use in multiple languages with string formatting, attributed strings, and automatic grammar agreement.

Download

iOS 15.0+ | iPadOS 15.0+ | Xcode 13.0+ | watchOS 8.0+

Overview

> Note
>
> This sample code project is associated with WWDC21 session 10109: What's New in Foundation.

The Caffé sample app presents a list of menu items — each of which are available in a variety of sizes — that users can order from a café. In presenting the various food items and helping to prepare an order, the app uses various Foundation APIs to localize and stylize the app's text:

- `FormatStyle`-based formatting customizes the display of currency values, dates and times, and lists of strings.
- Attributed strings allow the app to easily create styled text with Markdown for display in SwiftUI views. The app also uses localized attributed strings to build strings at runtime, even when the word order changes in different languages.
- Automatic grammar agreement handles localization situations when strings need to adjust at runtime to match grammatical gender or number in certain languages.

Use Formatters to Format Strings at Runtime

When the app launches, the user can choose one of several foods to add to their order. When the user chooses a food item, a new view shows the item's ingredients and the available sizes with corresponding prices.

The ingredient list shows an example of formatting a list of items, using the `formatted(_:)` method defined on the Swift `Sequence` type. It starts with the an array of `ingredients` defined by the `Food` type. In `FoodHeaderView`, the `ingredientText` variable takes the ingredient strings, maps each to a localized string, and then uses the `formatted(_:)` method to create a comma-separated list. By adding the `ListFormatStyle` list type `ListFormatStyle.ListType.and` as a format style parameter, the formatter places an "and" (or its localized equivalent) before last member of the list.

```
private var ingredientText: String {
    food.ingredients.map(\.localizedDescription).formatted(.list(type: .and))
}
```

In English, the ingredient text reads "Our pizza is made from: prosciutto, cheese, flour, and tomatoes." In Spanish, the list reads "Nuestro pizza está hecho de: prosciutto, queso, harina y tomates."

The app also uses string formatters to present the price of each item, as seen here:

```
func localizedPrice(_ size: FoodSize) -> String {
    price[size]!.formatted(.currency(code: "USD"))
}
```

As with the list of ingredients earlier, the `formatted(_:)` method applies directly to the type it formats. In this case, the formatted type is a `Decimal`; this type conforms to Swift's `BinaryInteger`, which defines the `formatted(_:)` method. A `FormatStyle` parameter indicates that the formatting should format the price as a currency, using U.S. dollars.

For more sophisticated formatting needs, some format styles support chaining modifier methods to customize a default style. The Caffé app includes a companion app for Apple Watch that shows the next date when the user is eligible to receive a free coffee. The `Date` presented in this view customizes the default `dateTime` format style to show only the weekday, hour, and minute:

```
var str = date.formatted(.dateTime
                              .locale(locale)
                              .minute()
                              .hour()
                              .weekday()
                              .attributed)
```

Use Attributed Strings to Style Text

The previous listing also uses the `attributed` modifier to return an `AttributedString`. Attributed strings contain text and metadata that applies to ranges of that text. In this case, the attributed string returned by the formatter uses the `dateField` attribute to mark which ranges of text correspond to which parts of the formatted date. This allows the app to find the weekday attribute in the attribute container and change it to an orange foreground color attribute. The SwiftUI view can then use this attribute when styling the watch display.

```
let weekday = AttributeContainer
    .dateField(.weekday)

let color = AttributeContainer
    .foregroundColor(.orange)

str.replaceAttributes(weekday, with: color)
```

`AttributedString` is strongly-typed, meaning that all attributes must have defined names and value types. `AttributedString` defines attributes for Foundation, SwiftUI, AppKit, and UIKit in its `AttributeScopes` type. For common inline attributes like emphasis and links, attributed strings support initialization from with Markdown syntax, either in source or in `.strings` files. The following entry from the Spanish localization's `Localizable.strings` file shows Markdown formatting for strong emphasis (`**`), regular emphasis (`_`), and links (`[ ]` for link text, followed by a URL in parentheses):

```
"**Thank you!**" = "**¡Gracias!**";
"_Please visit our [website](https://www.example.com)._" = "_Visita nuestro [sitio w
```

An app can also define custom attributes, as Caffé does with its `RainbowAttribute` type, an attribute that indicates a range of text to display in multiple colors. The Caffé app adds this attribute by:

1. Defining the `RainbowAttribute` as an extension of `CodableAttributedStringKey`, and providing the name and value type of the attribute.
2. Extending `AttributeScopes` to define a new `AttributeScope` called `CaffeApp Attributes`, whose one member is `rainbow`, of type `RainbowAttribute`. The app also extends `AttributeScopes` with `caffeApp`, a variable of the `CaffeAppAttributes` type, that allows access to the Caffé app's custom attributes with dynamic member lookup syntax.
3. Extending `AttributeDynamicLookup` to provide a subscript method that takes key paths of type `CaffeAppAttributes`. This allows code to use dot syntax when looking up the members of `CaffeAppAttributes`.

```
enum RainbowAttribute: CodableAttributedStringKey, MarkdownDecodableAttributedStrin
    enum Value: String, Codable, Hashable {
        case plain
        case fun
        case extreme
    }

    static var name: String = "rainbow"
}

extension AttributeScopes {
    struct CaffeAppAttributes: AttributeScope {
        let rainbow: RainbowAttribute
    }

    var caffeApp: CaffeAppAttributes.Type { CaffeAppAttributes.self }
}

extension AttributeDynamicLookup {
    subscript<T: AttributedStringKey>(dynamicMember keyPath: KeyPath<AttributeScopes
        self[T.self]
    }
}
```

The implementation of `RainbowText` uses these attributes by creating an `AttributedString` and calling a private `annotateRainbowColors(from:)` method to apply its color attributes. To create an `AttributedString` that uses custom attribute scopes, Caffé uses the `init(localized:options:table:bundle:locale:comment:including:)` initializer, passing the key path to the custom attribute name as the `including:` parameter:

```
init(_ localizedKey: String.LocalizationValue) {
    attributedString = RainbowText.annotateRainbowColors(
        from: AttributedString(localized: localizedKey, including: \.caffeApp))
}
```

To apply a custom attribute in a string, a caller uses the Markdown extension syntax, as seen in the following example, which applies two different values of the rainbow attribute:

```
RainbowText("^[Fast](rainbow: 'fun') & ^[Delicious](rainbow: 'extreme') Food")
    .font(.slogan)
    .frame(maxWidth: 260, alignment: .leading)
```

Simplify Localization by Performing Grammar Agreement Automatically

Some languages' grammar require that nouns, adjectives, articles, and other parts of speech agree in number or gender with other parts of a sentence. Localized attributed strings can perform this agreement by using a template string to format the values at runtime.

In Caffé, each food's detail view has a button indicating how many of each item the user has selected to add to their order. The app fills in this button text with the number, size, and food item to add to the order:

```
Button(
    "Add ^[\(quantity) \(foodSizeSelection.localizedName) \(food.localizedName)](inf
    action: orderButtonTapped
)
```

The syntax `^[text](inflect:true)` tells the generated attributed string to *inflect* the string, meaning to perform automatic grammar agreement on the range of text within the square braces. This process takes into account the value of any numeric substitutions and grammatical gender of string substitutions. In English, this causes the food name to pluralize when `quantity` is not equal to `1`.

In Spanish, the localized string in the `.strings` file uses the parameter reordering syntax to place the noun before the adjective, like the following:

```
"Añadir ^[%1$lld %3$@ %2$@](inflect: true) a tu pedido";
```

When the automatic grammar engine inflects the generated string for Spanish, it pluralizes the food name, as it does in English. In Spanish, it also adjusts the adjective (`foodSizeSelection.localizedName`) to match the number of `quantity` and the grammatical gender of `food.localizedName`. For example, one small salad becomes "1 ensalada pequeña" in Spanish, while two small salads is "2 ensaladas pequeñas". In both cases, the grammar engine changes the adjective "pequeño" to match the feminine gender of "ensalada".

In some languages, an app may need to provide part-of-speech information to the inflection engine. This happens in English, where the words "sandwich" and "juice" are both a noun and a verb. In Spanish, the food size terms "grande" and "enorme" can be used as both adjectives and nouns. The inflection engine logs a warning when it encounters this type of ambiguity. To clarify intent, the inflection engine accepts a grammar markup that wraps the substitution with the syntax `^[…](morphology: {…})` and provides part-of-speech information. The following entry from the English strings file shows an example of this disambiguation:

```
"Add ^[%lld %@ %@](inflect: true) to your order" = "Add ^[%lld %@ ^[%@](morphology:
```