Sample Code

Providing Multiple Complications

Present multiple complications for a single complication family using descriptors.

Download

iOS 13.0+ | iPadOS 13.0+ | watchOS 7.1+ | Xcode 12.5+

Overview

The Coffee Tracker app records a user's caffeine intake over the course of a 24-hour period. Each time the user adds a drink, the app recalculates the current caffeine levels and the equivalent cups of coffee consumed. It then updates the complication timeline and estimates the decrease in the user's caffeine level over the next 24 hours.

This sample provides three different complications.

- The Caffeine Dose complication displays the estimated amount of caffeine remaining in the user's body based on the logged drinks. The caffeine dose decreases over time based on a 5-hour half-life.
- The Total Cups complication displays the equivalent number of 8-ounce cups of regular coffee that the user drank based on the logged drinks. For example, if the user logs a double espresso, the Total Cups increases by 1.3 cups.
- The Both Caffeine and Cups complication displays both the caffeine dose and the total equivalent cups drank.

For most complication families, the app provides both the Caffeine Dose and the Total Cups complication; however, some families have enough space to display the Both Caffeine and Cups complication instead. For example, for the `CLKComplicationFamily.graphicCircular` family, Coffee Tracker provides a Caffeine Dose and a Total Cups complication, while it only provides the Both Caffeine and Cups complication for the `CLKComplicationFamily.graphicBezel` family.

Configure the Sample Code Project

To add the complication to an active watch face, start by building and running the sample code project in the simulator, and follow these steps:

1. Click the Digital Crown to exit the app and return to the watch face.
2. Using the trackpad, firmly press the watch face to put the face in edit mode, and then tap Customize.
3. Swipe left until the configuration screen highlights the complications. Select the complication to modify.
4. Scroll to the Coffee Tracker complication, and then click the Digital Crown again to save your changes.
5. Tap the Coffee Tracker complication to go back to the app.

For more information on setting up watch faces, see Change the watch face on your Apple Watch.

Define the Complications Supported by the App

The Coffee Tracker app declares constants that contain an identifier and a display name for each type of complication that the app supports. The system uses the identifiers to define the different types of complications. It shows the display names when the user scrolls through the complications while configuring a watch face.

```
// A structure that contains the identifier and display name for a specific
// complication.
struct Complication {
    let identifier: String
    let displayName: String

    static let caffeine = Complication(identifier: "Coffee_Tracker_Caffeine_Dose", d
    static let cups = Complication(identifier: "Coffee_Tracker_Number_Of_Cups", disp
    static let both = Complication(identifier: "Coffee_Tracker_Both", displayName: '
}
```

Then, the complication data source's `getComplicationDescriptors(handler:)` method creates a descriptor for each identifier.

It starts by creating the descriptor for the Caffeine Dose complications. Here, the `supportedFamilies` parameter specifies that this complication is available for most (but not all) of the complication families.

```
// Create the descriptor for complications that show the current caffeine dose.
let caffeine = CLKComplicationDescriptor(
    identifier: Complication.caffeine.identifier,
    displayName: Complication.caffeine.displayName,
    supportedFamilies: [.modularSmall,
                        .utilitarianSmall,
                        .utilitarianSmallFlat,
                        .utilitarianLarge,
                        .circularSmall,
                        .extraLarge,
                        .graphicCorner,
                        .graphicCircular,
                        .graphicRectangular,
                        .graphicExtraLarge])
```

Next, it creates the Total Cups descriptor. This descriptor supports the same subset of families.

```
// Create the descriptor for complications that show the equivalent number
// of 8-ounce cups drank today.
let cups = CLKComplicationDescriptor(
    identifier: Complication.cups.identifier,
    displayName: Complication.cups.displayName,
    supportedFamilies: [.modularSmall,
                        .utilitarianSmall,
                        .utilitarianSmallFlat,
                        .utilitarianLarge,
                        .circularSmall,
                        .extraLarge,
                        .graphicCorner,
                        .graphicCircular,
                        .graphicRectangular,
                        .graphicExtraLarge])
```

Then, it creates the Both Caffeine and Cups descriptor. This descriptor supports the families that aren't already supported by the Caffeine Dose and Total Cups descriptors.

```
// Create the descriptor for complications that show both the current caffeine
// dose and the total number of cups drank.
let both = CLKComplicationDescriptor(identifier: Complication.both.identifier,
                                     displayName: Complication.both.displayName,
                                     supportedFamilies: [.modularLarge, .graphicBeze
```

Finally, the app passes the descriptors back to the `getComplication Descriptors(handler:)` method's handler. While each descriptor doesn't need to cover all possible families, every family is covered by at least one descriptor.

```
handler([both, caffeine, cups])
```

The complications appear in the same order as the descriptor array. When the user configures a complication, the picker shows the first three items from the array that support the complication's family. If there are more than three, the picker displays a More button to provide access to the additional complications.

Create Timeline Entries for Each Type of Complication

When the system calls the complication data source's `getCurrentTimelineEntry(for:withHandler:)`, `getTimelineEntries(for:after:limit:withHandler:)`, or `getLocalizableSampleTemplate(for:withHandler:)` method to request new timeline entries, it passes a `CLKComplication` object that contains the complication's `family` and `identifier`.

Coffee Tracker then creates a template based on the family and identifier pair by creating a switch statement for the pair.

```
switch (complication.family, complication.identifier) {
```

For most families, the app checks the complication to determine which template it should provide.

```
case (.modularSmall, Complication.cups.identifier):
    return createCupsModularSmallTemplate(forDate: date)
case (.modularSmall, _):
    return createCaffeineModularSmallTemplate(forDate: date)
```

> **Note**
>
> If an app creates templates based on the complication's `identifier` property, it must check for the `CLKDefaultComplicationIdentifier` default identifier. Coffee Tracker first explicitly checks for the Total Cups identifier and then uses the Caffeine Dose templates for any other identifiers.

If the family only supports a single complication, the app just returns a template based on the family.

```
case (.modularLarge, _):
    return createBothModularLargeTemplate(forDate: date)
```

Finally, the app creates a template that displays the requested data for the specified family. For example, the following code creates a modular small template that displays the current caffeine dose.

```
private func createCaffeineModularSmallTemplate(forDate date: Date) -> CLKComplicati
    // Create the data providers.
    let mgCaffeineProvider = CLKSimpleTextProvider(text: data.mgCaffeineString(atDat
    let mgUnitProvider = CLKSimpleTextProvider(text: "mg Caffeine", shortText: "mg")

    // Create the template using the providers.
    return CLKComplicationTemplateModularSmallStackText(line1TextProvider: mgCaffeir
                                                        line2TextProvider: mgUnitPro
}
```

See Also

Sample Code

{} Creating and updating a complication's timeline

Create complications that batch-load a timeline of future entries and run periodic background sessions to update the timeline.