Apple Watch Programming Guide

Developer

Contents

Figures, Tables, and Listings

Objective-CSwift

Overview

Important: This is a preliminary document for an API or technology in development. Apple is supplying this information to help you plan for the adoption of the technologies and programming interfaces described herein for use on Apple-branded products. This information is subject to change, and software implemented according to this document should be tested with final operating system software and final documentation. Newer versions of this document may be provided with future betas of the API or technology.

Developing for Apple Watch

With Apple Watch, users can now access data in a way that is both distinctly personal and unobtrusive. Without having to pull an iPhone out of a pocket, users can get important information quickly just by glancing at their Apple Watch (Figure 1-1).

Figure 1-1 The Apple Watch Home screen

As a developer of third-party apps for Apple Watch, you support these brief interactions by providing only the most relevant information in the most straightforward way possible.

Apple Watch and Its Paired iPhone

Apple Watch requires the presence of an iPhone to run third-party apps. To create a third-party app, you need two separate bundles: a WatchKit app (that runs on Apple Watch) and a WatchKit extension (that runs on the user's iPhone). The *WatchKit app* contains only the storyboards and resource files associated with your app's user interface. The *WatchKit extension* contains the code for managing the WatchKit app's user interface and for responding to user interactions.

User interactions are what make the Apple Watch unique. First, you always provide users with a full-app experience, which they interact with by opening your app from the Home screen. The full interface, with its multiple screens of content, makes it easy for user to interact with your app's data.

Besides the full-app experience, you can optionally offer users a read-only interface, known as a *glance*, which displays timely and relevant information from your app. Finally, you can change the way local and remote notifications are displayed to your users by providing custom notification interfaces.

Because a WatchKit app extends the behavior of your existing iOS app, the WatchKit app and WatchKit extension are bundled together and packaged inside your iOS app bundle. During installation of your iOS app, the system prompts the user to install the WatchKit app when a paired Apple Watch is present.

The WatchKit App

A WatchKit app is a user launchable app that appears on the Apple Watch home screen. A WatchKit app is the user's main way of viewing and interacting with your data. It provides the means to view your data and optionally to manipulate or interact with that data. Depending on the data, a WatchKit app might present only a subset of the data presented by its containing iOS app.

A WatchKit app acts as the public face of your app but it works in tandem with your WatchKit extension, which is the brains of the operation. The WatchKit app contains only the storyboards and resource files associated with your app's user interface. The WatchKit extension contains the code for managing content, responding to user interactions, and updating your user interface. And because the extension runs on the user's iPhone, it can coordinate with your iOS app as needed to perform more sophisticated tasks.

To get started creating a WatchKit app, see App Essentials (page 28).

Glance Interfaces

A glance is a focused interface that you use to display your app's most important information. Glances are aptly named because they are intended to be looked at quickly by the user. Glances are nonscrolling; the entire glance interface must fit on a single screen. Glances are read-only and cannot contain buttons, switches, or other interactive controls. Tapping a glance launches your WatchKit app.

To create a glance, you do not have to create a separate executable. Instead, you create a specialized set of objects inside your existing WatchKit app and WatchKit extension. In fact, the classes and techniques you use to implement a glance are the same ones you use to create your WatchKit app.

To get started creating a glance interface, see Glance Essentials (page 59).

Custom Interfaces for Local and Remote Notifications

Apple Watch works with its paired iPhone to display local and remote notifications. Initially, Apple Watch uses a minimal interface to display incoming notifications. When the user's movement indicates a desire to see more information, the minimal interface changes to a more detailed interface displaying the contents of the notification. You can customize this detailed interface and add custom graphics or arrange the notification data differently from the default interface provided by the system.

Apple Watch provides automatic support for the actionable notifications introduced in iOS 8. Actionable notifications are a way to add buttons to your notification interface that reflect actions the user might take. For example, a notification for a meeting invite might include buttons to accept or reject the invitation. When your iOS app registers support for actionable notifications, Apple Watch automatically adds appropriate buttons to the notification interfaces on Apple Watch. All you need to do is handle the actions that the user selects. You do this in your WatchKit extension.

To get started creating a custom notification interface, see Notification Essentials (page 65).

Configuring Your Xcode Project

A WatchKit app requires an existing iOS app. In the Xcode project for your iOS app, you add a new WatchKit app target, which configures the bundles and initial resources for your WatchKit app and WatchKit extension. Those bundles are then delivered as part of your iOS app on the App Store.

The WatchKit app target provided by Xcode contains everything you need to get started creating your WatchKit app, glances, and custom notification interfaces. And iOS Simulator provides you with a runtime environment for testing the appearance and behavior of all of your interfaces.

Note: WatchKit development requires the iOS 8.2 SDK or later. To get the latest SDK, go to developer.apple.com.

Adding a WatchKit App to Your iOS Project

You must have an existing iOS app to create a WatchKit app. The WatchKit app is implemented as a separate target of your Xcode project and is built and packaged inside your iOS app's bundle.

To add a WatchKit app target to your existing iOS app project

1. In Xcode, open the project for your iOS app.
2. Select File > New > Target, and navigate to the Apple Watch section.
3. Select WatchKit App, then click Next.
4. If you plan to implement a glance or custom notification interface, select the appropriate checkboxes.

   For notification interfaces, it is recommended that you select the Include Notification Scene checkbox, even if you do not plan on implementing that interface right away. Selecting that checkbox adds an additional file to your project for debugging your notification interfaces. If you do not select that option, later you must create the file manually.
5. Click Finish.

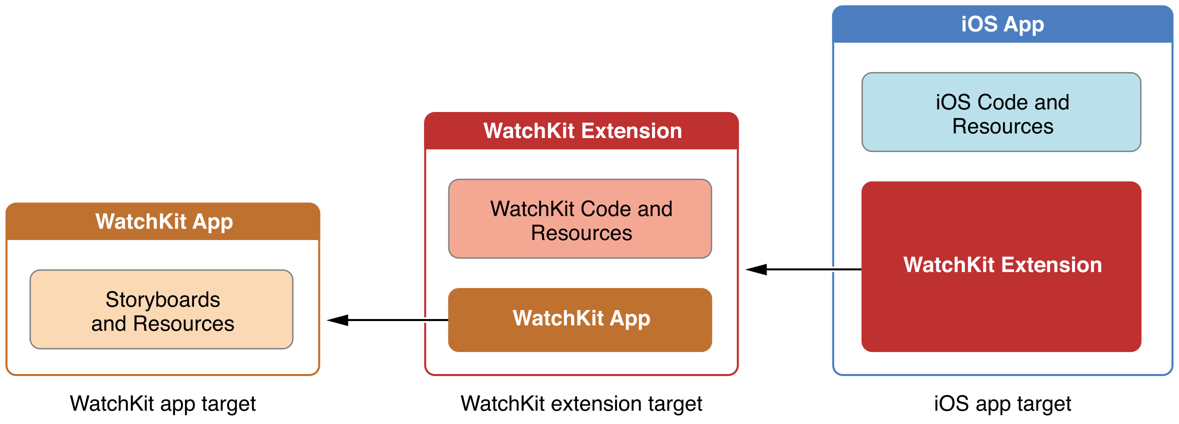
Xcode configures the targets for your WatchKit app and WatchKit extension and adds the needed files to your iOS project. The bundle IDs for both new targets are configured automatically, based on the bundle ID of your iOS app. The base IDs for all three bundles must match; if you change your iOS app's bundle ID, you must update the other bundle IDs accordingly.

App Target Structure

Adding a WatchKit App target to your Xcode project creates two new executables and updates your project's build dependencies. Building your iOS app builds all three executables (the iOS app, WatchKit extension, and WatchKit app) and packages them together. Xcode also creates a build scheme specifically for building and debugging only your WatchKit app.

Figure 2-1 illustrates the structure of your iOS app and WatchKit executables. The WatchKit app is packaged inside your WatchKit extension, which is in turn packaged inside your iOS app. When the user installs your iOS app on an iPhone, the system prompts the user to install your WatchKit app if there is a paired Apple Watch available. iOS handles the installation process automatically and requires no further work on your part.

Figure 2-1 The target structure for a WatchKit app



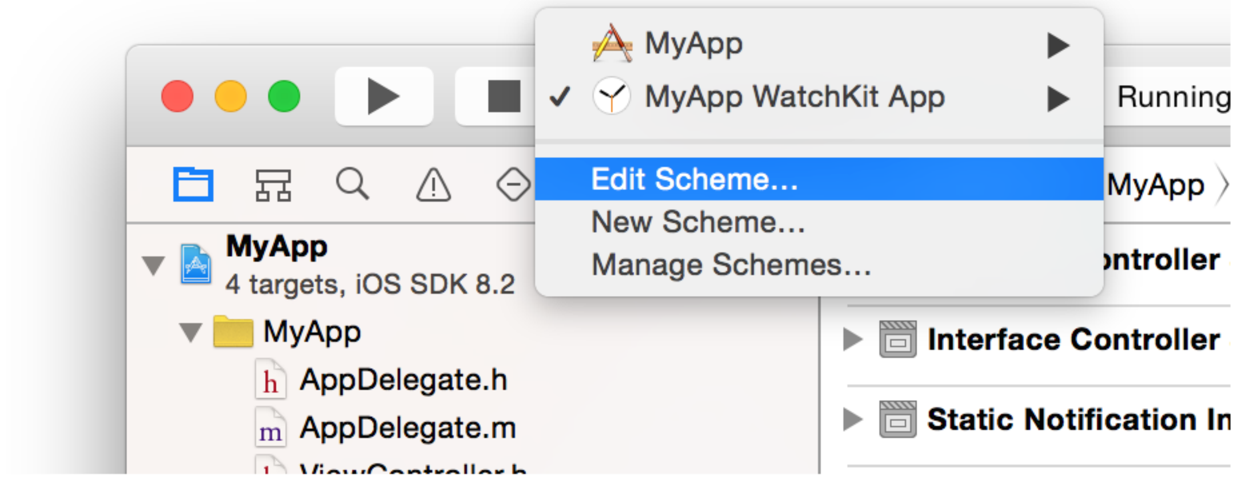
The Build, Run, and Debug Process

When you create your WatchKit app target, Xcode automatically configures a build scheme to run and debug your WatchKit app. Use that scheme to launch and run your app in iOS Simulator or on a device.

For apps that include glance or custom notification interfaces, you must configure additional build schemes to test those interfaces. Use the glance scheme to debug your glance interface in the simulator and use the notification scheme to test your dynamic and static notification interfaces.

To configure custom build schemes for glances and notifications

1. Select your existing WatchKit app scheme.
2. From the scheme menu, select Edit Scheme.

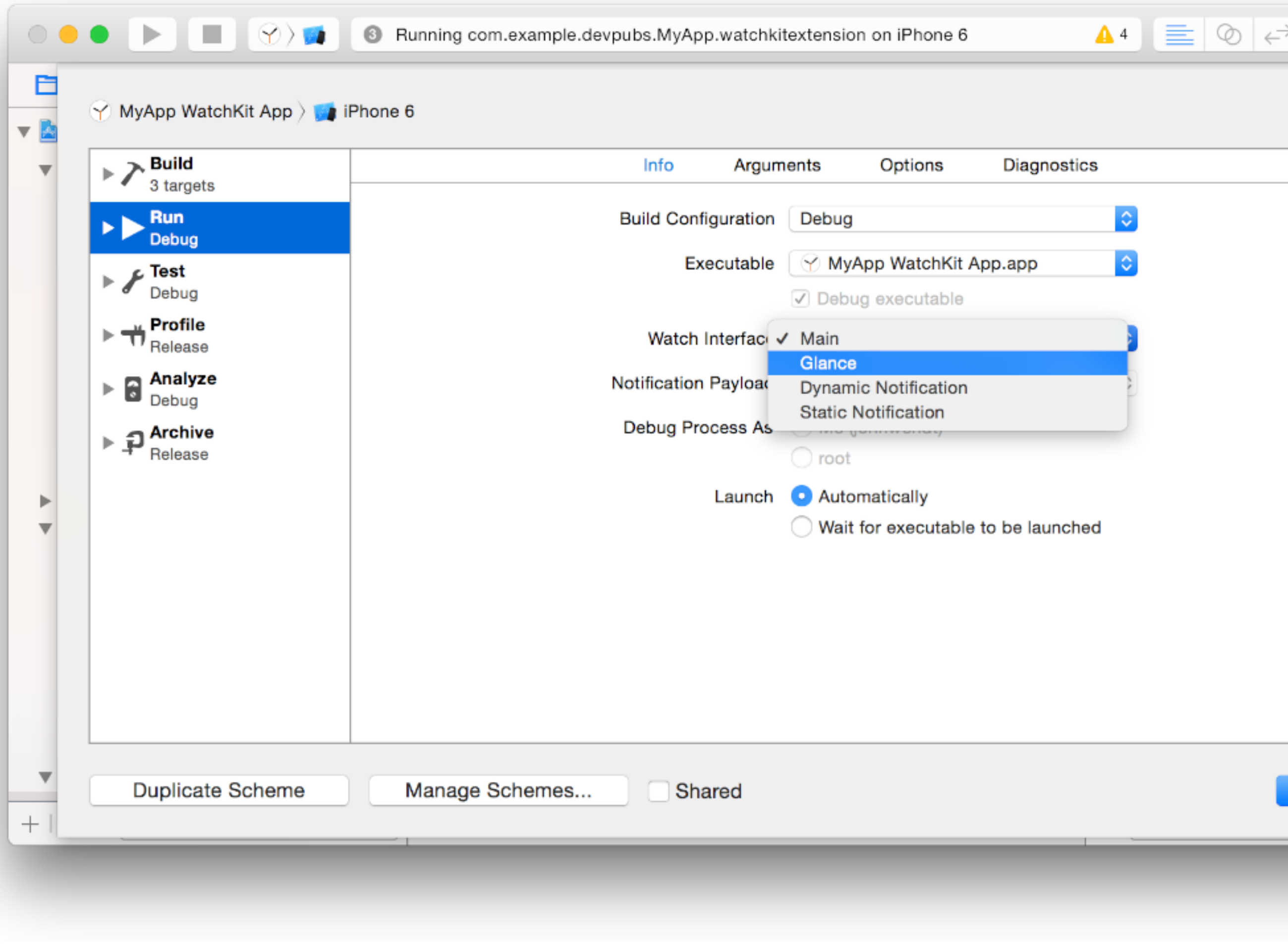


3. Duplicate your existing WatchKit app scheme, and give the new scheme an appropriate name.

   For example, give it a name like "Glance - My WatchKit app" to indicate that the scheme is specifically for running and debugging your glance.

4. Select Run in the left column of the scheme editor.

5. In the Info tab, select the appropriate executable for the new scheme.



Build Configuration: Debug
Executable: MyApp WatchKit App.app
☑ Debug executable
Watch Interface: ✓ Main, Glance, Dynamic Notification, Static Notification
Notification Payload
Debug Process As: root
Launch: ● Automatically ○ Wait for executable to be launched

6. Close the scheme editor to save your changes.

When you create a build scheme for your notification interfaces, specify a JSON file to use as the notification payload during testing. For more information about notification payloads, see Specifying a Notification Payload for Testing (page 15).

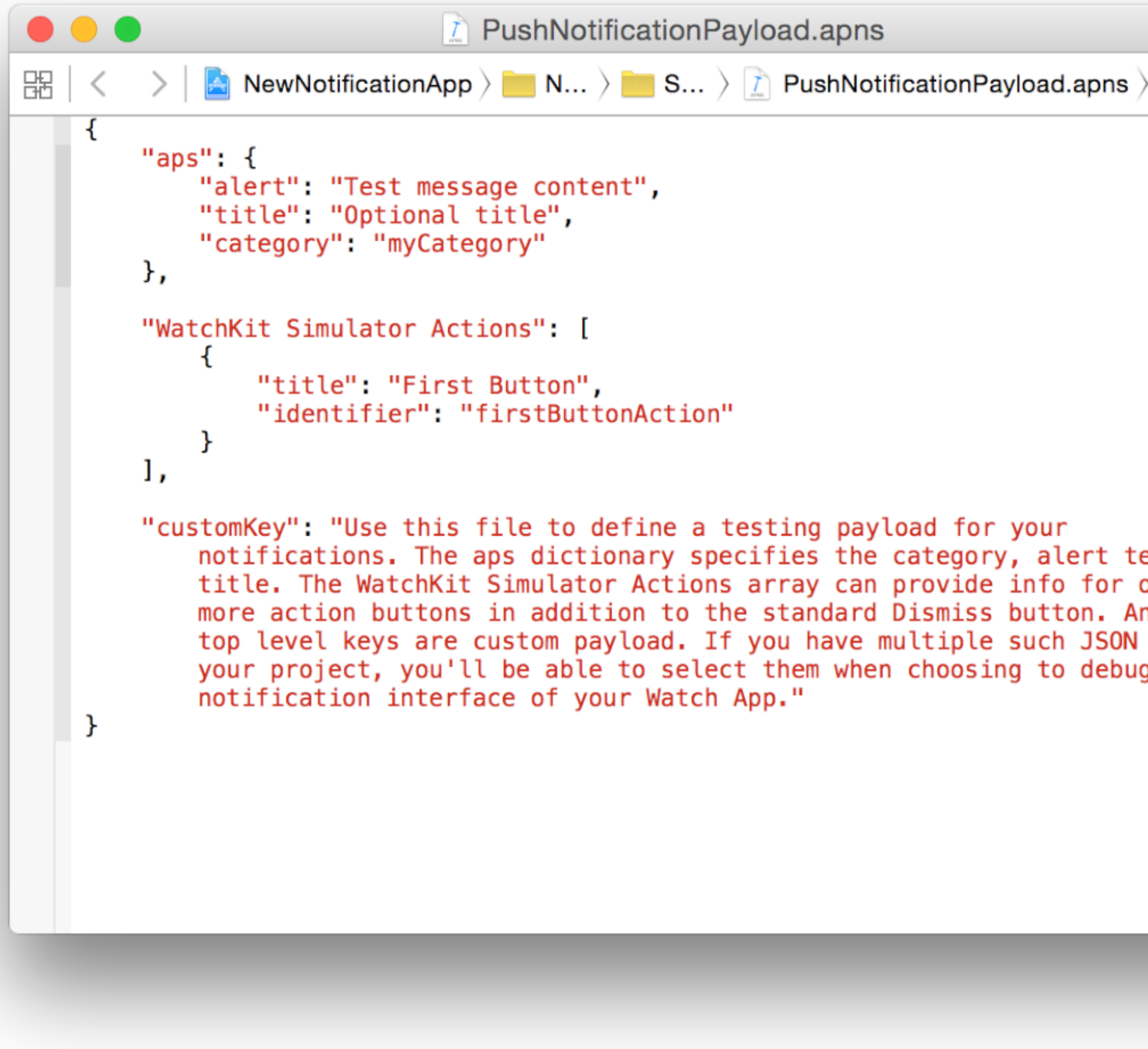
Specifying a Notification Payload for Testing

When debugging custom notification interfaces in iOS Simulator, you can specify the JSON payload data you want delivered to your interface during testing. Use the scheme editor to specify which payload you want to use when running your notification interface. The payload itself is a file that you create with a `.apns` filename extension.

Note: If you selected the Include Notification Scene option when creating your WatchKit app target, Xcode provides you with an initial `PushNotificationPayload.apns` file for specifying your test data. (The file is located in the Supporting Files directory of your WatchKit extension.) You can also create payload files manually later.

The `PushNotificationPayload.apns` file contains most of the keys you need to simulate a remote notification, and you can add more keys as needed. Figure 2-2 shows the default JSON file that comes with your project.

Figure 2-2 A simulated remote notification payload


PushNotificationPayload.apns
NewNotificationApp 〉 N... 〉 S... 〉 PushNotificationPayload.apns

```
{
    "aps": {
        "alert": "Test message content",
        "title": "Optional title",
        "category": "myCategory"
    },

    "WatchKit Simulator Actions": [
        {
            "title": "First Button",
            "identifier": "firstButtonAction"
        }
    ],

    "customKey": "Use this file to define a testing payload for your
        notifications. The aps dictionary specifies the category, alert te
        title. The WatchKit Simulator Actions array can provide info for o
        more action buttons in addition to the standard Dismiss button. An
        top level keys are custom payload. If you have multiple such JSON
        your project, you'll be able to select them when choosing to debug
        notification interface of your Watch App."
}
```

Most of the JSON data is packaged into a dictionary and delivered to your code at runtime. Because iOS Simulator does not have access to your iOS app's registered actions, you may also use payload files to specify the action buttons to display in your interface. The WatchKit Simulator Actions key contains an array of dictionaries, each of which represents an action button to add to your interface. Each dictionary contains the following keys:

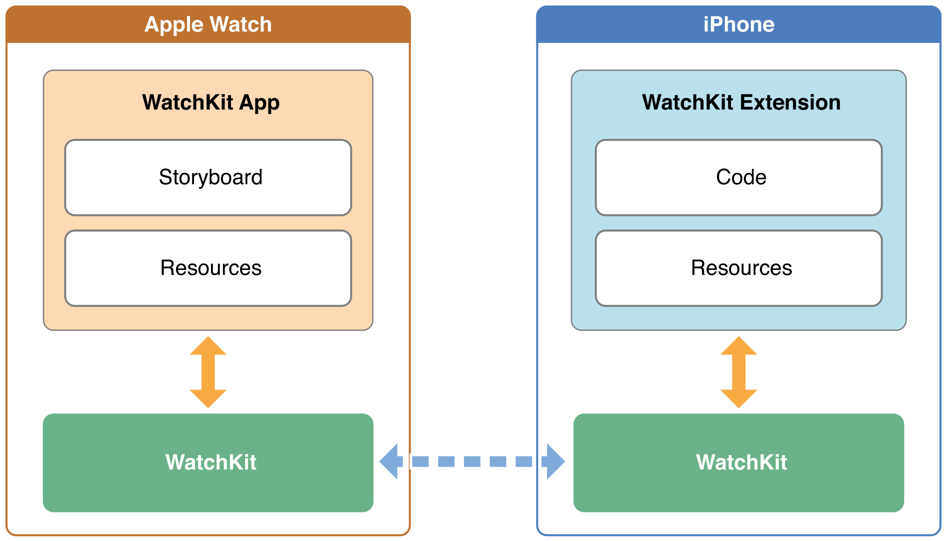
- `title`—The value of this key is the title of the action button. This key is required.
- `identifier`—The value of this key is the string to pass to your interface controller's `application:handleActionWithIdentifier:forLocalNotification:completionHandler:` or `application:handleActionWithIdentifier:forRemoteNotification:completionHandler:` method. This key is required.
- `destructive`—The value of this key is the value `1` or `0`, where `1` causes the resulting button to be rendered in a way that indicates it performs a destructive action. The value `0` causes the button to be rendered normally. This key is optional.

To test your notification interface with the JSON payload, configure the build scheme with the appropriate payload file. When you select a notification interface executable, Xcode adds a menu for choosing one of your payload files. You can create different build schemes for different notification payloads, or you can update the payload file for an existing build scheme before testing.

WatchKit App Architecture

Your WatchKit app and WatchKit extension work in tandem to implement your app's interface. When the user interacts with your app on Apple Watch, your WatchKit app chooses the appropriate scene from your storyboards to handle that interaction. For example, if the user views your app's glance, it chooses your glance scene. After choosing the scene, WatchKit tells the paired iPhone to launch your WatchKit extension and create the objects needed to manage that scene. When the scene is fully configured, it is displayed on Apple Watch. This transfer of information between the WatchKit app and WatchKit extension happens transparently behind the scenes.

Figure 3-1 Communication between a WatchKit app and WatchKit extension



Managing Scenes: The Interface Controller

Each scene is managed by a single *interface controller* object, which is an instance of the `WKInterfaceController` class. An interface controller in WatchKit serves the same purpose as a view controller in iOS: It presents and manages content on the screen and responds to user interactions with that content. Unlike a view controller, an interface controller does not manage the actual views of your interface. Those views are managed for you behind the scenes by WatchKit.

WatchKit apps typically contain multiple interface controllers, with each one displaying a different type of information. Because only one interface controller at a time is displayed onscreen, an app presents new interface controllers in response to user actions. Apps can present interface controllers modally. The navigation style of an app also determines how interface controllers are presented. For information on how to present new interface controllers, see Interface Navigation (page 33).

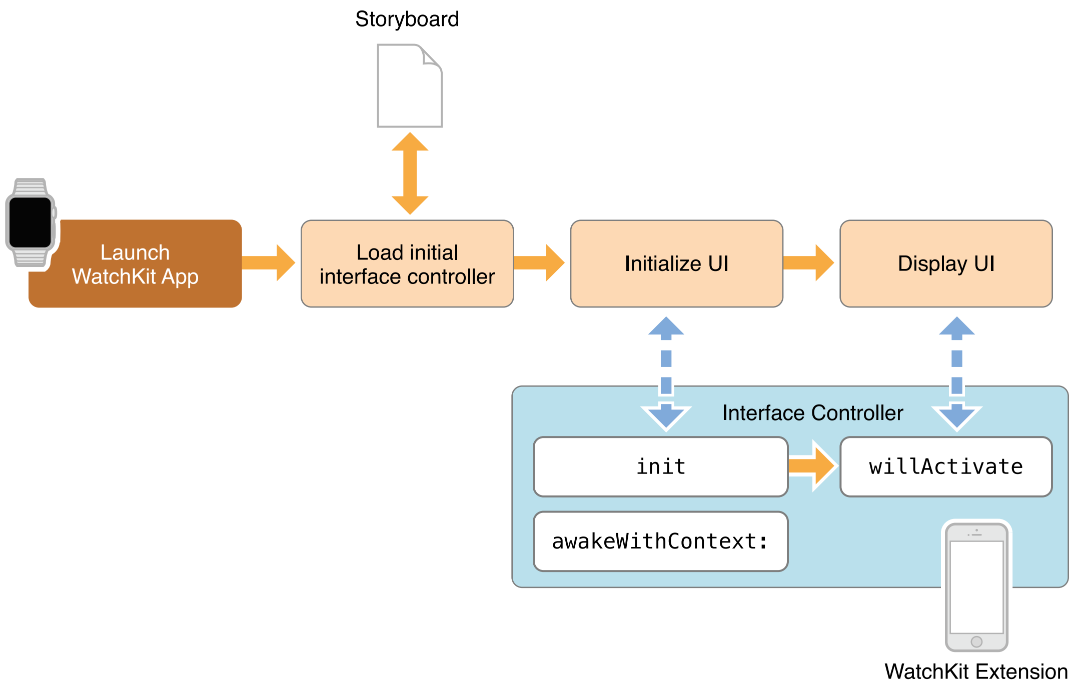
Note: Glance and custom notification interfaces use specialized interface controllers that are separate from your app's other interface controllers. For information about implementing a glance, see Glance Essentials (page 59). For information about the implementing custom notification interfaces, see Notification Essentials (page 65).

WatchKit App Life Cycle

User interactions with Apple Watch drive the launching of your app and its life cycle. The user can launch your app from the Home screen, interact with your glance, or view notifications using your custom UI. Each of these interactions launches your WatchKit app and the corresponding WatchKit extension. Your WatchKit app and WatchKit pass information back and forth until the user stops interacting with your app, at which point iOS suspends the extension until the next user interaction.

At launch time, WatchKit automatically loads the appropriate scene for the current interaction. If the user views your app's glance, WatchKit loads the glance scene from your storyboard; if the user launched your app directly, WatchKit loads the initial scene for your app. After it loads the scene, WatchKit asks the WatchKit extension to create the corresponding interface controller object, which you use to prepare the scene for display to the user. Figure 3-2 shows the steps for this sequence. You'll learn more about how interface controllers work in App Essentials (page 28).

Figure 3-2 Launching a WatchKit app



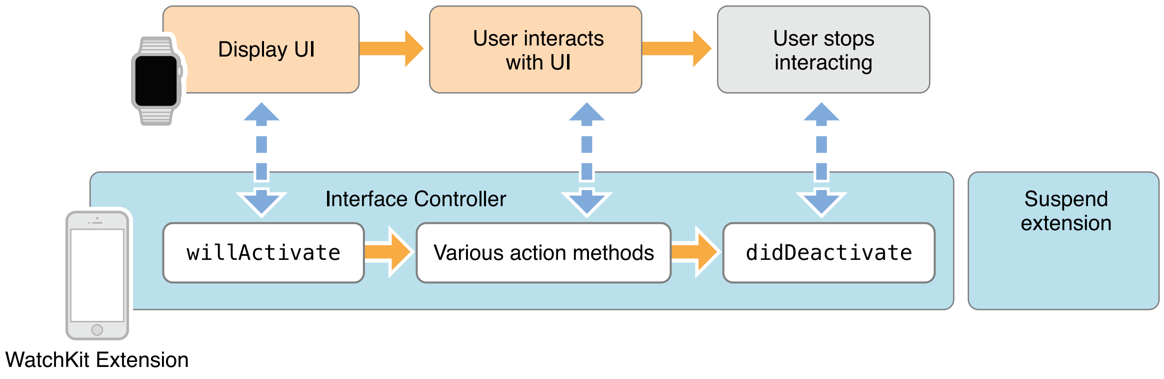
Use your interface controller's `init` and `awakeWithContext:` methods to load any required data, set the values for any interface objects, and prepare your interface to be displayed. Do not use the `willActivate` to initialize your interface controller. The `willActivate` method is called shortly before your interface is displayed onscreen, so you should use that method only to make last-minute changes. For example, you might also use that method to start animations or start other tasks that should only happen while your interface is onscreen.

While your interface controller is onscreen, user interactions are handled by your interface controller's custom action methods. As the user interacts with tables, buttons, switches, sliders, and other controls, WatchKit calls your action methods so that you can respond. You use those action methods to update your interface or perform other relevant tasks. To perform tasks at other times, use an `NSTimer` object to run code at the time you designate.

> **Note:** Glance interfaces do not support action methods. Tapping your app’s glance interface always launches the app.

Your WatchKit extension remains running only while the user is interacting with your app on Apple Watch. Interactions with Apple Watch are meant to be brief, so interface controllers should be lightweight and never perform long-running tasks. When the user exits your app explicitly or stops interacting with Apple watch, iOS deactivates the current interface controller and suspends your extension, as shown in Figure 3-3.

Figure 3-3 The life cycle of an interface controller



Tasks to Perform at Different Stages of an App’s Life

At different stages of your app’s life, iOS calls the methods of your `WKInterfaceController` objects to give you a chance to respond. Table 3-1 lists the key methods that you should almost always implement in your interface controllers, along with the kinds of tasks you perform in them.

Table 3-1 Key methods of `WKInterfaceController`

Method	Tasks to perform
`init`	This method is your first chance to initialize your interface controller.
`awakeWithContext:`	This method lets you configure the interface controller using any available context data. Use it to load data and update labels, images, tables, and other interface objects in your storyboard scene. The context data is data you provide to assist in the configuration of the new interface controller. For example, when pushing a new interface controller in a hierarchical interface, you specify a context object that contains the next level of data to display. Providing a context object is recommended but not required.

Method	Tasks to perform
`willActivate`	This method lets you know that your interface will soon be visible to the user. Use this method only to make small changes to your interface. For example, you might use this method to update a label based on new data. The bulk of your interface initialization should still occur in the `init` and `awakeWithContext:` methods.
`didDeactivate`	Use the `didDeactivate` method to clean up your interface and put it into a quiescent state. For example, use this method to invalidate timers and stop animations. You cannot set values for any interface objects from this method. From the time this method is called to the time the `willActivate` method is called again, any attempts to set values for interface objects are ignored.

Debugging Your Activation Code in iOS Simulator

During testing, you can lock and unlock the simulator to verify that your activation and deactivation code is working as expected. When you use the Hardware > Lock command to lock the simulator, WatchKit calls the `didDeactivate` method of the current interface controller. When you subsequently unlock the simulator, WatchKit calls the `willActivate` method of the interface controller.

Sharing Data with Your Containing iOS App

If your iOS app and WatchKit extension rely on the same data, use a shared app group to store that data. An app group is a secure container that multiple processes can access. Because your WatchKit extension and iOS app run in separate sandbox environments, they normally do not share files or communicate directly with one another. An app group lets the two processes share files or user defaults.

You set up a shared app group from the Capabilities tab of your iOS app and WatchKit extension. Enabling the App Groups capability adds an entitlements file (as needed) to each target and adds the `com.apple.security.application-groups` entitlement to that file. To share data, both targets must have the same app group selected.

At runtime, you share files between processes by reading and writing those files in the shared container directory. To access the container directory, use the `containerURLForSecurityApplicationGroupIdentifier:` method of `NSFileManager` to retrieve the base URL for the directory. Use the provided URL to enumerate the directory contents, or create new URLs for files in the directory.

To share preferences data between apps, create an `NSUserDefaults` object using the identifier of the shared group. The `initWithSuiteName:` method of `NSUserDefaults` creates an object that allows access to the shared user defaults data. Both processes can access this data and write changes to it.

Communicating Directly with Your Containing iOS App

Apps that work closely with their containing iOS app can use the `openParentApplication:reply:` method to send requests to that app and receive a response. WatchKit extensions do not support background execution modes; they run only while the user is interacting with the corresponding app on Apple Watch. Their containing iOS app has fewer restrictions and can be configured to run in the background or to gather information on behalf of the WatchKit extension. Activities that might require extra time to complete, such as fetching a user's location, should therefore be performed by the iOS app and communicated back to the WatchKit extension.

When you call the `openParentApplication:reply:` method, iOS launches or wakes the containing iOS app in the background and calls the `application:handleWatchKitExtensionRequest:reply:` method of its app delegate. The app delegate performs the request using the provided dictionary and then returns a reply to the WatchKit extension.

Leveraging iOS Technologies

WatchKit extensions have access to the same technologies found in iOS apps but because they are extensions, the use of some technologies may be restricted and the use of others is not recommended. Here are some guidelines for deciding when to use a particular technology:

- **Avoid using technologies that request user permission, like Core Location.** Using the technology from your WatchKit extension could involve displaying an unexpected prompt on the user's iPhone the first time you make the request. Worse, it could happen at a time when the iPhone is in the user's pocket and not visible.
- **Do not use background execution modes for a technology.** WatchKit extensions run only while the user interacts with the corresponding WatchKit app and are therefore considered foreground extensions. As a result, WatchKit extensions cannot execute using the background modes supported by some technologies.
- **Avoid performing long-running tasks with a technology.** A WatchKit extension is suspended soon after the user stops interacting with the corresponding WatchKit app. Because WatchKit app interactions are typically brief, the extension might already be suspended by the time the requested data arrives.

The best solution for performing any long-running tasks is to let your iOS app perform the task instead. For example, instead of starting location services in your WatchKit extension, start it in your iOS app. Your iOS app can gather the needed data and put it in a shared app group so that your extension can access it later. Use the `openParentApplication:reply:` method to initiate tasks and receive a reply, or use a shared group container to communicate details between your iOS app and WatchKit extension. For information about how to handle communication between your iOS app and WatchKit extension, see Communicating Directly with Your Containing iOS App (page 24).

Handoff Support

Apple Watch supports the creation of activities that can be completed on other devices using Handoff. You can use the `updateUserActivity:userInfo:webpageURL:` method of `WKInterfaceController` to create activities and advertise them to other devices.

With the exception of your app's glance, Apple Watch does not handle activities generated by other devices. In your glance interface controller, you can use an activity dictionary to specify information that might be useful to your main app. If the user taps your glance to launch your app, WatchKit delivers that activity dictionary to your app's main interface controller. That interface controller can use the contents of the dictionary to update the app's UI accordingly. For more information about how to pass information from your glance to your app, see Customizing App Launch From Your Glance (page 62).

Remote Control Events and Now Playing Information

Apple Watch uses the remote control events system to manage the playback of audio or video on a user's paired iPhone. The transport controls of the Now Playing glance generate remote control events for the app that is currently playing content. An iOS app that registers handlers with the commands of the `MPRemoteCommandCenter` object receives these events automatically when it is the "Now Playing" app. You do not need to do extra work in your WatchKit extension to support remote control events coming from Apple Watch.

Note: For feedback commands to like, dislike, or bookmark an item, Apple Watch uses the `localizedShortTitle` instead of the `localizedTitle` string of the `MPFeedbackCommand` object.

The Now Playing glance automatically displays any "Now Playing" information provided by the currently playing iOS app. An iOS app provides this information using the `MPNowPlayingInfoCenter` object. As your app plays its content, it should update the values of the `nowPlayingInfo` dictionary. Apple Watch automatically retrieves this information and displays it. In addition, tapping the track title in the Now Playing glance launches the app's WatchKit app if one is available.

For information on how to implement support for remote control events and now playing information in your iOS app, see Remote Control Events.

WatchKit Apps

UI Essentials

The starting point for implementing your app is to define your storyboard scenes. Each scene defines a portion of your app's user interface. You can customize scenes for different Apple Watch sizes, and you can configure different aspects of your interface.

Assembling Your Storyboard Scenes

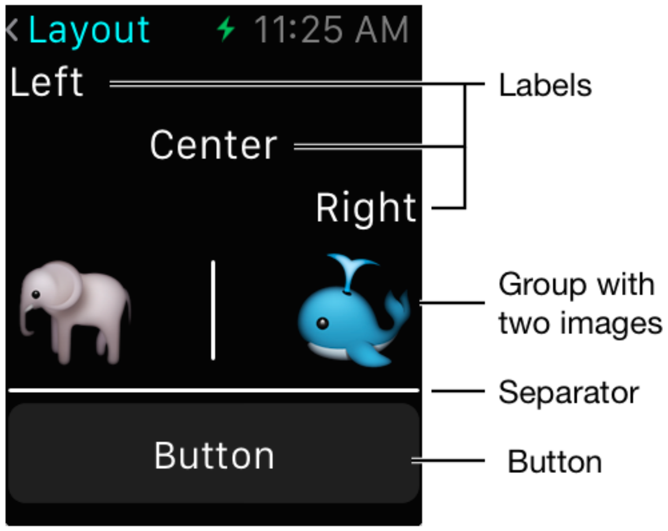
WatchKit apps do not use the same layout model as iOS apps. When assembling the scenes of your WatchKit app interface, you do not create view hierarchies by placing elements arbitrarily in the available space. Instead, as you add elements to your scene, Xcode arranges the items for you, stacking them vertically on different lines. At runtime, Apple Watch takes those elements and lays them out for you based on the available space.

Although Xcode handles the overall layout of your interface, WatchKit provides ways to fine tune the position of items within a scene. The size and position of most items can be configured using the Attributes inspector. Changing the position of an item changes lets you set the horizontal and vertical alignment of that item at its current location in the element stack. The sizing options let you specify a fixed width for an item or give it the ability to resize itself within the available space.

Group objects offer another important tool for arranging other elements in your interface. Group elements are a container for other elements, giving you the option to arrange elements in the group horizontally as well as vertically. You can nest groups within other groups and use each group's spacing and inset values to alter the size and position of items. Groups have no default visual representation but they can be configured with a background color or image if you want.

Figure 5-1 shows how you can arrange different elements in your storyboard file. The first three elements are labels, which have different alignments within the interface controller's bounds. Below the labels is a group object containing two images arranged horizontally. The interface also contains a separator and a button stacked vertically underneath the group object.

Figure 5-1 Interface objects in Xcode



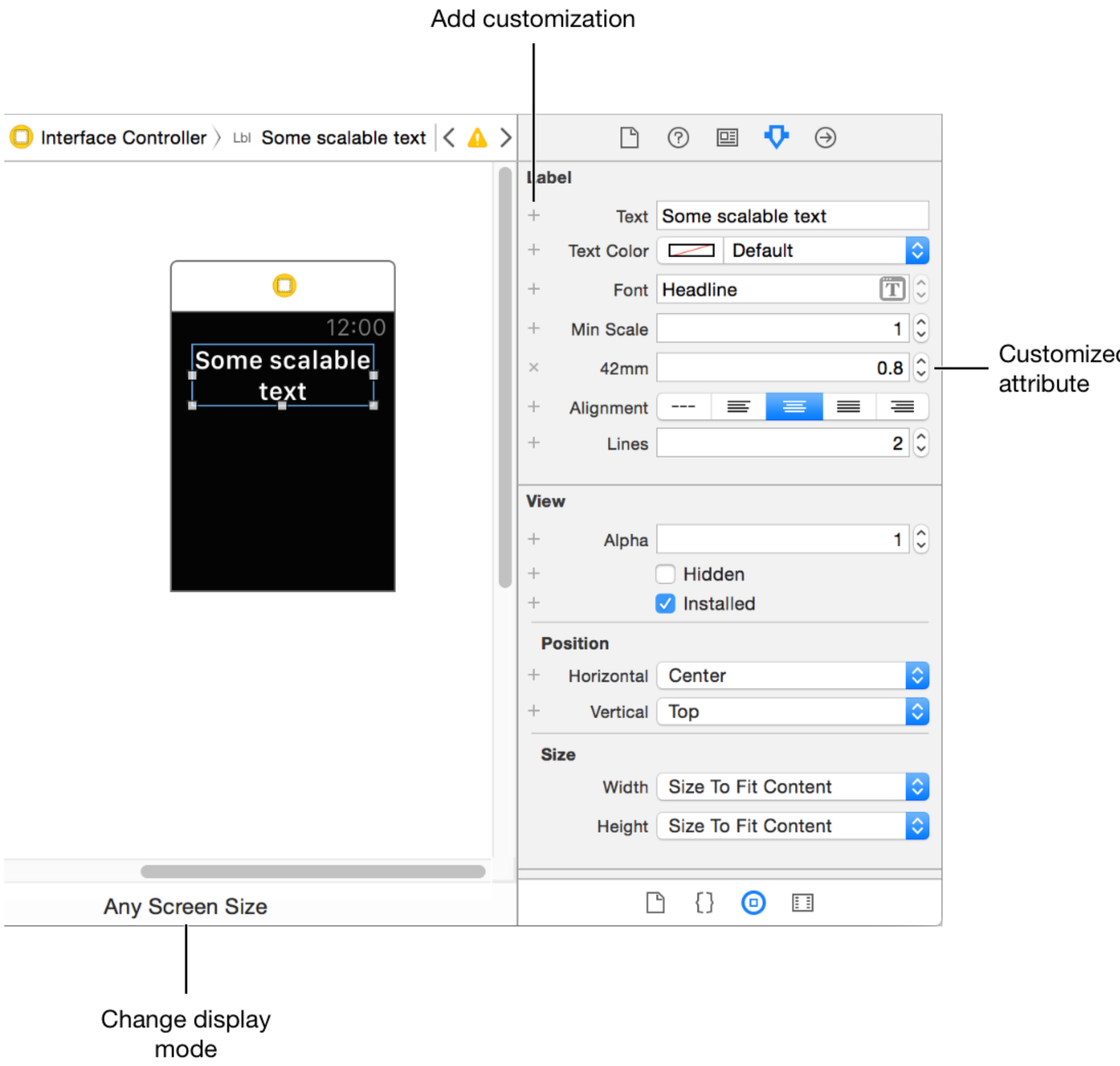
When creating your interfaces in Xcode, let objects resize themselves to fit the available space whenever possible. App interfaces should be able to run both display sizes of Apple Watch. Letting the system resize objects to fit the available space minimizes the amount of custom code you have to write for each device.

Accommodating Different Display Sizes

Xcode supports customizing your interface for the different sizes of Apple Watch. The changes you make in the storyboard editor by default apply to all sizes of Apple Watch, but you can customize your storyboard scenes as needed for different devices. For example, you might make minor adjustments to the spacing and layout of items or specify different images for different device sizes.

To customize an item for a specific device size, use the plus buttons (+) in the Attributes inspector to override the value of an attribute for a given device. Clicking a plus button adds a new device-specific entry for the attribute. Changes you make to that version of the attribute affect only the selected device. Figure 5-2 shows how text scaling is handled differently for Apple Watch 42mm.

Figure 5-2 Customizing attributes for different devices



Label

Text	Some scalable text
Text Color	Default
Font	Headline
Min Scale	1
42mm	0.8
Alignment	
Lines	2

View

Alpha	1
☐ Hidden	
✓ Installed	

Position

Horizontal	Center
Vertical	Top

Size

Width	Size To Fit Content
Height	Size To Fit Content

Users should not notice significant differences in your app's interface on different sizes of Apple Watch, so minimize the customizations you make for different device sizes. Whenever possible, limit interface changes to layout-related behaviors such as spacing and margins. Although it is possible to remove interface objects altogether from your interface in different layouts, doing so is not recommended. Try to use the exact same set of interface objects on all sizes of Apple Watch.

To see customizations applied to different device sizes, use the control at the bottom of the storyboard editor to toggle between the device sizes. The storyboard editor displays the Any device size by default. Changes applied in the Any display mode apply to all sizes of Apple Watch. If you change the display mode to a specific device size, the changes you make while in that mode apply only to the current device.

Updating Your Interface at Runtime

At runtime, an interface controller can make the following modifications to the objects in its corresponding storyboard scene:

- Set or update data values.
- Change the visual appearance of objects that support such modifications.
- Change the size of an object.
- Change the transparency of an object.
- Show or hide an object.

You cannot add new objects to your interface or change the order of the objects that are already there. Although you cannot remove objects, you can hide them, which removes them from the layout temporarily. When an item is hidden, other objects fill in the space previously occupied by the item. To hide an object without filling in the space, set the item's alpha value to 0. For more information about hiding objects in a scene, see Hiding Interface Objects (page 41).

Setting Your App's Key Color

Every WatchKit app has an associated key color, which is applied to the following UI elements:

- The title string in the status bar
- The app name in short-look notifications

An app's key color is stored in the Global Tint property of an app's storyboard. To access this property, select your storyboard and display the File inspector. Select one of several preexisting colors from the pop-up menu, or use the color picker to specify a custom color.

Internationalizing Your Interface

The storyboards that come with your WatchKit app have base internationalization enabled by default. This feature causes any strings in your storyboard to be added to your project's `Localizable.strings` files automatically. Simply translate the strings in those files for each target language, and include them with your shipping app. When you create a storyboard scene at runtime, Xcode inserts the strings associated with the appropriate localization.

Arrange your interface so that labels and controls that contain text have room to expand. Instead of placing multiple buttons on the same line, arrange them vertically so that each one has plenty of room to display its title.

For text and images that you specify programmatically, use the same internationalization techniques you use for iOS and OS X apps:

- Use the `NSLocalizedString` macros to load strings from resource files.
- Use an `NSNumberFormatter` object to format numerical values.
- Use an `NSDateFormatter` object to format dates.

When used in your WatchKit extension, an `NSLocale` object returns the locale information configured on the user's Apple Watch. Use that class to get the user's preferred languages and other language and locale-related information.

For more information about internationalizing your app, see *Internationalization and Localization Guide* .

Interface Navigation

For WatchKit apps with more than one screen of content, you must choose a technique for navigating between those screens. WatchKit apps support two navigation styles, which are mutually exclusive:

- **Page based.** This style is suited for apps with simple data models where the data on each page is not closely related to the data on any other page. A page-based interface contains two or more independent interface controllers, only one of which is displayed at any given time. At runtime, the user navigates between interface controllers by swiping left or right on the screen. A dot indicator control at the bottom of the screen indicates the user's current position among the pages.
- **Hierarchical.** This style is suited for apps with more complex data models or apps whose data is more hierarchical. A hierarchical interface always starts with a single root interface controller. In that interface controller, you provide controls that, when tapped, *push* new interface controllers onto the screen.

Although you cannot mix page-based and hierarchical navigation styles in your app, you can supplement these base navigation styles with modal presentations. Modal presentations are a way to interrupt the current user workflow to request input or display information. You can present interface controllers modally from both page-based and hierarchical apps. The modal presentation itself can consist of a single screen or multiple screens arranged in a page-based layout.

Implementing a Page-Based Interface

You configure a page-based interface in your app's storyboard by creating a next-page segue from one interface controller to the next.

To create a next-page segue between interface controllers

1. In your storyboard, add interface controllers for each of the pages in your interface.
2. Control-click your app's main interface controller, and drag the segue line to another interface controller scene.

   The second interface controller should highlight, indicating that a segue is possible.
3. Release the mouse button.
4. Select "next page" from the relationship segue panel.

5. Using the same technique, create segues from each interface controller to the next.

   The order in which you create your segues defines the order of the pages in your interface.

The segues you create in your storyboard file define the page-based interface that is loaded when your app is launched. You can change the set of pages you want to display by calling the `reloadRootControllersWithNames:contexts:` method early in the launch cycle. For example, you might call that method in the `init` method of your main interface controller to force WatchKit to load a different set of pages.

All interface controllers in a page-based interface are created and initialized before the interface is displayed, but only one interface controller at a time is displayed. Normally, WatchKit displays the first interface controller in the sequence initially. To change the initially displayed interface controller, call the `becomeCurrentPage` method from its `init` or `awakeWithContext:` method.

As the user navigates from page to page, WatchKit activates and deactivates interface controllers accordingly. During a transition, the currently visible interface controller's `didDeactivate` method is called, followed by a call to the `willActivate` method of the interface controller that is about to be displayed. Use the `willActivate` method to update the contents of your interface controller to reflect any last minute changes.

Implementing a Hierarchical Interface

In a hierarchical interface, you tell WatchKit when to transition to a new screen using segues or by calling the `pushControllerWithName:context:` method of the current interface controller. In your storyboard, you create push segues from a button, group, or table row in your interface to another interface controller in your storyboard. If you prefer to initiate the push transition programmatically, call the `pushControllerWithName:context:` method from any of your interface controller's action methods.

When pushing a new interface controller onto the screen, it is recommended that you pass a data object in the `context` parameter of the `pushControllerWithName:context:` method. This context object is how you pass state information to the new interface controller before it appears onscreen. Use this object to tell the new interface controller what data to display.

A pushed interface controller displays a chevron in the upper-left corner of the screen to indicate that the user can navigate backward. When the user taps the upper-left corner of the screen or performs a left-edge swipe, WatchKit dismisses the topmost interface controller automatically. You can also dismiss the interface controller programmatically by calling its `popController` method. You cannot dismiss your app's main interface controller.

Presenting Interface Controllers Modally

A modal interface is a way to interrupt the current navigation flow temporarily to prompt the user or display information. You can present a modal interface from any interface controller, regardless of the navigation style used by your app. To display an interface controller modally, do one of the following:

- Create a modal segue in your storyboard file.
- Call the `presentControllerWithName:context:` method to present a single interface controller modally.
- Call the `presentControllerWithNames:contexts:` method to present two or more interface controllers modally using a page-based layout.

When creating a modal segue, connect the segue to the interface controller you want to display. When using a segue to present multiple interface controllers, first use the next-page segue to connect the modal interface controllers together, in the same way that you connect them together for a page-based interface. Your modal segue should connect to the first interface controller in the group. If you connect to an interface controller in the middle of the group, the interface controllers that precede it in the group are not displayed.

The top-left corner of a modal interface displays the interface controller's title string. When the user taps that string, WatchKit dismisses the modal interface. Set the title string to reflect the meaning of dismissing the modal interface. For example, when displaying information, you might set the string to `Done` or `Close`. If you do not specify a title for your interface controller, WatchKit displays the string `Cancel` by default.

Interface Objects

Objective-CSwift

You manipulate your WatchKit app's UI using interface objects. An interface object is an instance of the `WKInterfaceObject` class, or more specifically one of its subclasses. The WatchKit framework provides interface objects for most (but not all) of the visual elements you can add to your UI in your storyboard files. Interface objects are not views. They are proxy objects that communicate wirelessly with the actual views used to implement your UI on Apple Watch.

Note: Communication between an interface object and the corresponding view on Apple Watch is one way, with information flowing from your WatchKit extension to Apple Watch. In other words, you set values on an interface object but you cannot get the current values of its attributes. There are performance and latency implications for retrieving data from Apple Watch, making changes, and writing those changes back to the device. So it is recommended that you maintain information about the configuration of your interface in your WatchKit extension.

Creating an Interface Object

You create interface objects indirectly by adding declared properties to your interface controller and connecting those properties to the corresponding elements in your storyboard file. You never allocate and initialize interface objects yourself. During the initialization of your interface controller object, WatchKit creates the interface objects for any connected outlets automatically.

When adding declared properties for interface objects, set the class to the appropriate type and include the `IBOutlet` keyword in your declaration. For example, the declaration for a label is as follows:

```
@interface MyHelloWorldController()
@property (weak, nonatomic) IBOutlet WKInterfaceLabel* label;
@end
```

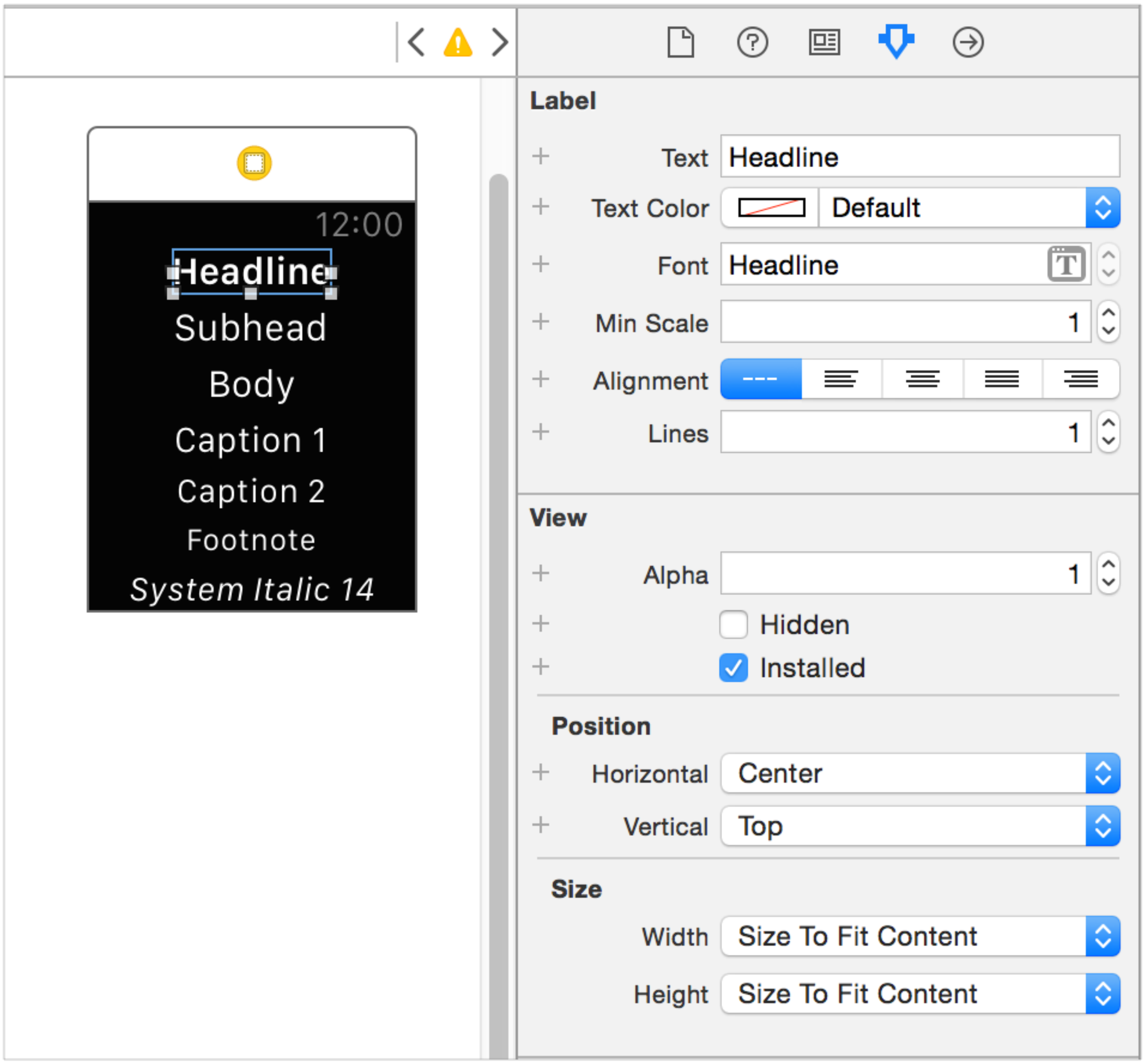
```
class MySwiftInterfaceController {
    @IBOutlet weak var label: WKInterfaceLabel!
}
```

Connect each declared property in your interface controller to the corresponding item in your storyboard. A quick way to create property declarations and connect them to an item is to use the assistant editor in Xcode. After displaying the assistant editor, control-drag from an element in your storyboard to the interface definition of your class to create an outlet. (In Swift, drag to your class definition.) After prompting you for the outlet's name, Xcode creates the property declaration in your class and connects it to the storyboard element.

Configuring Your Interface at Design Time

At design time, use Xcode to configure the appearance of the visual elements in your storyboards. For many layout-related attributes, design time is the only time you can configure the attribute. For example, you can change a label's text, color, and font using a `WKInterfaceLabel` object, but you cannot change the number of lines or the height of each line. Those attributes must be configured in Xcode, as shown in Figure 7-1.

Figure 7-1 Configuring a label object



Label

Text: Headline
Text Color: Default
Font: Headline
Min Scale: 1
Alignment
Lines: 1

View

Alpha: 1
☐ Hidden
☑ Installed

Position

Horizontal: Center
Vertical: Top

Size

Width: Size To Fit Content
Height: Size To Fit Content

For more information about how to configure interface objects, see the classes for your interface objects in *WatchKit Framework Reference*.

Changing Your Interface at Runtime

In the code of your WatchKit extension, you update your app's UI by calling methods of any referenced interface objects. An interface controller may change the configuration of its interface objects only while it is active, which includes initialization time. In your `init`, `awakeWithContext:`, and `willActivate` methods, call methods to assign data values to labels, images, and other objects in your user interface. You might also update them from your interface controller's action methods.

At initialization time, it is important to let WatchKit initialize your interface controller class before doing anything else. The initialization methods of `WKInterfaceController` and its subclasses are where WatchKit creates your app's interface objects. So any initialization code you write for your interface controllers must call the `super` implementation first. Listing 7-1 shows an example of an `init` method for an interface controller that contains an outlet (called `label`) for a `WKInterfaceLabel` object.

Listing 7-1 Initializing an interface controller

```
- (instancetype)init {
    // Always call super first.
    self = [super init];
    if (self){
        // It is now safe to access interface objects.
        [self.label setText:@“Hello New World”];
    }
    return self;
}
```

```
override init {
    // Initialize variables here.
    super.init

    // It is now safe to access interface objects.
    label.setText("Hello New World")
}
```

To improve performance and battery life, the WatchKit framework optimizes any attempts to set values on your app's interface objects. Whenever you set the values for one or more interface objects in the same run loop iteration, the new values are coalesced and transmitted to Apple Watch in a single batch to improve efficiency. Coalescing changes means that only that last change to a given property of an object is sent to the device. More importantly, setting the same property to the same value generates a log message to help you track down the duplicate calls.

For information about the methods you use to configure your interface objects, see the corresponding class descriptions in *WatchKit Framework Reference* .

Responding to User Interactions

Use buttons, switches, and other interactive controls to initiate changes in your app. When a button is tapped or the value of another control changes, WatchKit calls the associated action method in your interface controller. Each type of interface object has a required format for its action method, which are listed in Table 7-1. Change the name of the action methods to something appropriate for your app.

Table 7-1 Action methods for interface objects

Object	Objective-C	Swift
Button	`- (IBAction)buttonAction`	`@IBAction func buttonAction()`
Switch	`- (IBAction)switchAction:(BOOL)on`	`@IBAction func switchAction(value: Bool)`
Slider	`- (IBAction)sliderAction:(float)value`	`@IBAction func sliderAction(value: Float)`
Menu Item	`- (IBAction)menuItemAction`	`@IBAction func menuItemAction()`

Interfaces can use segues or the `table:didSelectRowAtIndex:` method of the interface controller to respond to taps in a table row. Use a segue to display another interface controller. Prior to performing the segue, WatchKit calls the `contextForSegueWithIdentifier:inTable:rowIndex:` or `contextsForSegueWithIdentifier:inTable:rowIndex:` method of your interface controller so that you can specify the context objects to use when displaying the interface controller. If you use the `table:didSelectRowAtIndex:` method instead of a segue, you can perform whatever actions are appropriate for tapping on the row.

After your interface controller is initialized and onscreen, WatchKit calls the methods of your interface controller only when the user interacts with your interface. If you want to update your user interface without user intervention, you must configure an `NSTimer` object and use its handler to perform any needed tasks.

For tasks that might take more than a second or two, consider handing those tasks off to your parent iOS app for execution. Long-running tasks such as network access and location monitoring are best handled by the parent app, which can then communicate that information back to your WatchKit extension through a shared group container directory. For information about handing off tasks to your parent app, see Communicating Directly with Your Containing iOS App (page 24).

Hiding Interface Objects

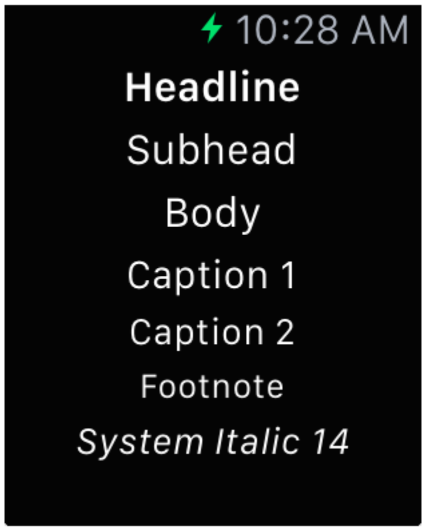
Hiding objects lets you use the same interface controller to display different types of content. Each scene in your storyboard file must contain all of the interface objects needed to display its content at runtime. If you customize your interface based on the data you have available, you can hide objects that you do not need. When you hide an object, you effectively remove it from your interface. During layout, hidden items are treated as if they were removed entirely from the layout altogether. To hide an object, call its `setHidden:` method and pass the value `YES`.

Text and Labels

To display text in your WatchKit app, use label objects. Labels support formatted text that can be changed programmatically at runtime.

To add a label to your interface controller, drag it into the corresponding storyboard scene. From there, configure the label's initial text string and format. WatchKit supports both standard fonts and custom fonts that you specify yourself. Figure 8-1 shows the standard font styles available for you to use.

Figure 8-1 Standard font styles for labels



For more information about configuring label objects in Xcode and in your code, see *WKInterfaceLabel Class Reference* .

Using Fonts

WatchKit apps, glance interfaces, and notification interfaces use the system font for displaying text. The interface controllers in your main app may also use custom fonts to display text. (Glance and notification interfaces cannot use custom fonts.) To use custom fonts, you must install those fonts by doing the following:

- Include the custom font file in both your WatchKit app and your WatchKit extension bundle.

- Add the `UIAppFonts` key to your WatchKit app's `Info.plist` file, and use it to specify the fonts you added to the bundle. For more information about this key, see *Information Property List Key Reference*.

Important: You must include the font in your WatchKit extension so that you can create strings with that font at runtime. The font information is included with the attributed string when it is sent to Apple Watch, and the copy of the font in your WatchKit app bundle is then used to render the string there.

To format text using a custom font, create an attributed string using the font information and then use that string to set the text of your label, as shown in Listing 8-1. The font name and size are encoded with the attributed string, which is then used to update the label on the user's Apple Watch. If the font name you specify is neither the system font nor one of your custom installed fonts, WatchKit uses the system font.

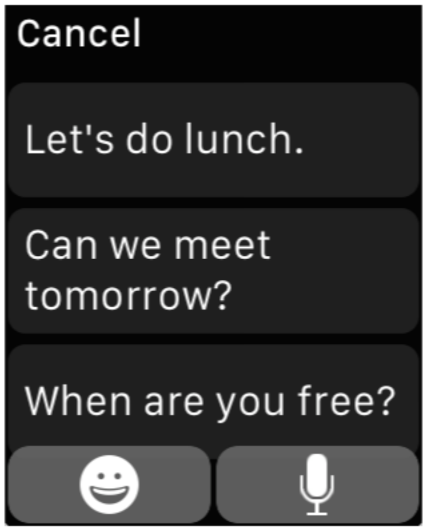
Listing 8-1 Using a custom font in a label string

```
// Configure an attributed string with custom font information

let menloFont = UIFont(name: "Menlo", size: 12.0)!
var fontAttrs = [NSFontAttributeName : menloFont]
var attrString = NSAttributedString(string: "My Text", attributes: fontAttrs)

// Set the text on the label object
self.label.setAttributedText(attrString)
```

Managing Text Input

WatchKit provides a standard modal interface for retrieving text input from the user. When presented, the interface allows the user to enter text via dictation or to select from a standard set of phrases or emoji, as shown in Figure 8-2.

Figure 8-2 Gathering text input from the user



To present this interface, call the `presentTextInputControllerWithSuggestions:allowedInputMode:completion:` method of the currently active interface controller. When presenting the interface, you specify the types of input you support and a block to execute with the results. You also specify an initial set of phrases to display in the interface. The user can select from the available phrases or use the controls to input a different phrase.

Listing 8-2 shows how to configure the text input controller and process the results. After you specify the initial phrases and input modes, the controller runs asynchronously. When the user selects an item or cancels input, your block is executed on the main thread. Use that block to retrieve the text or emoji image that was selected by the user and update your app.

Listing 8-2 Presenting the text input controller

```
NSArray* initialPhrases = @[@"Let's do lunch.", @"Can we meet tomorrow?", @"When
are you free?"];
[self presentTextInputControllerWithSuggestions:initialPhrases
      allowedInputMode:WKTextInputModeAllowAnimatedEmoji
      completion:^(NSArray *results) {
        if (results && results.count > 0) {
            id aResult = [results objectAtIndex:0];
            // Use the string or image.
        }
        else {
           // Nothing was selected.
        }
    }];
```

Internationalizing Your Text Code

WatchKit apps can use the same technologies for internationalization that iOS apps use.

- Use Xcode's base internationalization support for storyboards and xib files. Base internationalization lets you have only one set of storyboard files, which supports all localizations. The localized strings for the storyboard are stored separately in language-specific strings files.
- Use the `NSLocalizedString` family of macros to retrieve localized strings programmatically.
- Use the `NSNumberFormatter` class to format numerical values using the user's region and locale settings.
- Use the `NSDateFormatter` class to format dates using the user's region and locale settings.

When internationalizing your app, your main concern should be arranging your interface so that labels (and other controls with text) have room to expand. For example, rather than using a group to arrange three buttons horizontally, arrange the buttons vertically to give each one room to grow horizontally when its text becomes longer.

For more information about internationalizing your app, see *Internationalization and Localization Guide* .

Images

WatchKit provides the following ways to incorporate images into your content:

- The `WKInterfaceImage` class displays a single image or a sequence of images as standalone content.
- The `WKInterfaceGroup`, `WKInterfaceButton`, and `WKInterfaceController` classes allow you to specify an image as the background for other content.

Specifying Your Image Assets

Here are guidelines to follow when creating your image assets:

- Use the PNG format for images whenever possible.
- Always create images that are sized appropriately for your interface. For images whose size you cannot control, use the `setWidth:` and `setHeight:` methods of the interface object to ensure that the image is displayed at a proper size.
- Use image assets to manage your images. Image assets let you specify provide different versions of an image for each device size.

Using Named Images to Improve Performance

There are several ways to change the current image of an interface object:

- Use the `setImageNamed:` or `setBackgroundImageNamed:` methods to assign an image that is already in the WatchKit app bundle or that is currently in the on-device cache.

  Specifying images by name is preferred because only the name string is transferred to Apple Watch, which takes less time and uses less power than sending the entire image. WatchKit searches your WatchKit app bundle for an image file with the name you specified. If it does not find an image file in the bundle, it searches the device-side image caches for an image with the given name.
- Use the `setImage:`, `setImageData:`, `setBackgroundImage:`, or `setBackgroundImageData:` methods to transfer image data wirelessly from your WatchKit extension to your WatchKit app.

Any time you create a `UIImage` object in your extension, that image object exists on the user's iPhone and must be sent to Apple Watch before it can be used. Even using the `imageNamed:` method of `UIImage` loads the image from your WatchKit extension's bundle, not from your WatchKit app. If you try to assign that image to one of your interface objects, the image data is transferred wirelessly to Apple Watch.

Caching Images on the Device

For images you create in your WatchKit extension but use frequently, cache those images on the device and refer to them by name. You must cache images before you attempt to use them, by calling the `addCachedImage:name:` or `addCachedImageWithData:name:` methods of `WKInterfaceDevice`.

To use a cached image in your interface, do the following:

- For `WKInterfaceImage` objects, call the `setImageNamed:` method, specifying the name of the cached image.
- For `WKInterfaceGroup` and `WKInterfaceButton` objects, call the `setBackgroundImageNamed:` method, specifying the name of the cached image.

Important: When caching animated images, use the `animatedImageWithImages:duration:` method to create a single `UIImage` object with all of the animation frames and cache that image. Do not cache the images for the individual frames separately.

The Apple Watch image caches are limited in size, with each app receiving approximately 5 MB of cache space. Caches are persistent and can be used between launches of the WatchKit app. When your app's caches fill up, you must remove existing images from the cache before attempting to add new ones. Use the `removeCachedImageWithName:` method to remove a single image, or use the `removeAllCachedImages` method to clear the caches completely.

Tables

Objective-CSwift
Use tables to display lists of data whose content changes dynamically. WatchKit supports single-column tables using the `WKInterfaceTable` class. Displaying data in a table requires defining the layout for your data in advance and writing code to fill the table with the actual data at runtime. Specifically, you need to do the following in your Xcode project:

- In your storyboard file:
  - Add a table object to your interface controller scene. Create an outlet for that table in your interface controller.
  - Configure one or more row controllers for your table as described in Configuring Row Types (page 48).
- In your code:
  - Define a row controller class for each row controller you defined; see Configuring Row Types (page 48).
  - At initialization time, add rows to the table as described in Configuring the Table's Contents at Runtime (page 51).
  - Respond to interactions with table rows as described in Handling Row Selections (page 52).

For each table, you can define multiple row controller types, each with a different appearance. At runtime, you specify which row types you need and in what order they should be arranged in the table. For additional information about how to configure a table, see *WKInterfaceTable Class Reference* .

Configuring Row Controllers

A row controller is a template for displaying a single row of data in your table. When you add a table to your interface controller scene, Xcode automatically creates an initial row controller, but you can add more. For example, you might use different row controllers for content rows, headers, and footers in your table.

To add row controllers to a table

1. Select the table object in your storyboard file.
2. Open the Attributes inspector.

3. Use the Rows attribute to change the number of available row controllers.

Each row controller contains a single group element initially. To that group, you add the labels, images, and other objects that you want to include in the row. The content for labels and images in a row controller is irrelevant at design time. At runtime, you replace the content of each item when you configure the row.

Each row controller is backed by a custom class that you use to access the row's contents. Most row controller classes contain only properties for accessing the row's interface objects—few contain any code. However, if you add buttons or other interactive controls to a row, your row class can also include action methods for responding to user interactions with those controls.

To define a custom class for your row controller

1. Add a new Cocoa Touch class to your WatchKit extension.
2. Make your new class a subclass of `NSObject`.
3. Add declared properties for each label, image, or control that you plan to access at runtime. Use the following format for declared properties, changing the class to match the class of the corresponding interface object:

```
@property (weak, nonatomic) IBOutlet WKInterfaceLabel* label; //
Objective-C
```

Listing 10-1 shows a sample row controller class definition. In this example, the class contains outlets for an image and a label.

Listing 10-1 A sample class for managing a row

```
@interface MainRowType : NSObject
@property (weak, nonatomic) IBOutlet WKInterfaceLabel* rowDescription;
@property (weak, nonatomic) IBOutlet WKInterfaceImage* rowIcon;
@end
```

```
class MainRowType: NSObject {
    @IBOutlet weak var rowDescription: WKInterfaceLabel!
    @IBOutlet weak var rowIcon: WKInterfaceImage!
}
```

You finish the configuration of your row controller in your storyboard file by setting its class and connecting up any outlets. You must also assign an identifier string to the row. You use that string at runtime when creating the row.

To configure a row controller in your storyboard

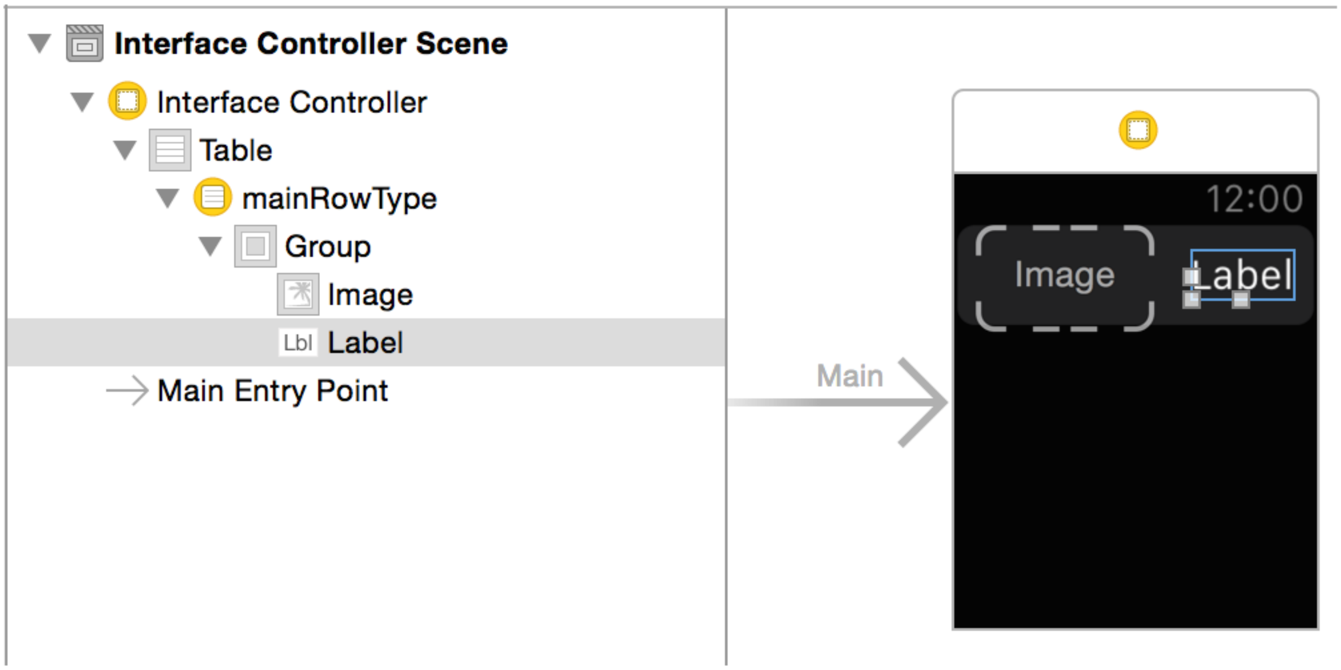
1. In your storyboard file, select the row controller object.
2. Set the row controller's Identifier attribute to a unique value for the table.

   You use the identifier later on when creating the table's rows. The value must be unique among the row types of the table but the actual value is at your discretion. Set this value in the Attributes inspector.
3. Set the class of the row controller to your custom class. Set this value in the Identity inspector.
4. Connect the labels and other elements to the outlets in your custom class.

   Connecting items in your storyboard file to your outlets binds the two together. WatchKit needs this information to create a row's interface objects at runtime.

Figure 10-1 shows an example of a row controller configured with the identifier `mainRowType` and the class `MainRowType`, which is defined in Listing 10-1 (page 49). The `rowDescription` and `rowIcon` outlets in that class are connected to the image and label objects in the row.

Figure 10-1 Examining a row controller in Xcode



Configuring the Table's Contents at Runtime

At runtime, you add rows to a table and configure the contents of each row programmatically. Add and configure rows as part of your interface controller's initialization process.

To create and configure the rows of a table

1. Determine the number and type of rows you want, based on the data you want to display.
2. Use the `setRowTypes:` or `setNumberOfRows:withRowType:` method to create the rows.

   Both methods create the rows in your interface and instantiate each row's corresponding class in your WatchKit extension. The instantiated classes are stored in the table and can be accessed with the `rowControllerAtIndex:` method.
3. Iterate over the rows using the `rowControllerAtIndex:` method.
4. Use the row controller objects to configure the row contents.

The `setRowTypes:` and `setNumberOfRows:withRowType:` methods instantiate the classes associated with the corresponding row controllers. Immediately after calling one of those methods, you retrieve the newly created row controller objects and use them to configure the rows. Listing 10-2 uses some provided data to configure a label and image for a row. The data is provided by an array of custom data objects of type `MyDataObject`. (The `MyDataObject` class exposes a string and image as properties and its implementation is not shown here.) The rows themselves are instances of the custom `MainRowType` class, which is defined in Listing 10-1 (page 49).

Listing 10-2 Creating and configuring the rows of a table

```
- (void)configureTableWithData:(NSArray*)dataObjects {
    [self.table setNumberOfRows:[dataObjects count] withRowType:@"mainRowType"];
    for (NSInteger i = 0; i < self.table.numberOfRows; i++) {
        MainRowType* theRow = [self.table rowControllerAtIndex:i];
        MyDataObject* dataObj = [dataObjects objectAtIndex:i];

        [theRow.rowDescription setText:dataObj.text];
        [theRow.rowIcon setImage:dataObj.image];
    }
}
```

When configuring tables, you can improve performance by limiting the number of rows you create initially. But because table rows must all be created up front, creating large numbers of rows can adversely affect the performance of your app. The precise number of rows depends on the complexity of your data and how long it takes you to create each one, but consider keeping the total number of rows to 20 or fewer. For tables that require more rows, consider loading only a subset of rows initially and then provide the user with controls to load more rows. An even better solution is to display only the most important subset of rows. For example, you might use location data to limit the number of rows to those that are most relevant to the user's current location.

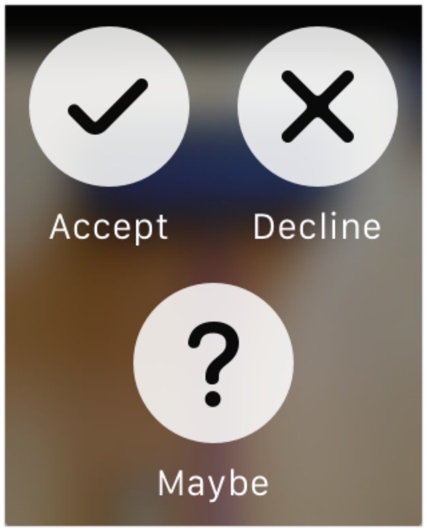
Handling Row Selections

An interface controller is responsible for handling row selections in any tables it owns. When the user taps the row of a table, WatchKit selects the row and calls the `table:didSelectRowAtIndex:` method of the interface controller. Use that method to perform relevant actions for your app. For example, you might display a new interface controller or update the row's content. If you do not want a table row to be selectable, deselect the Selectable option for the corresponding row controller in your storyboard.

Context Menus

Objective-CSwift
The Retina display with Force Touch found on Apple Watch provides a new way to interact with content. Instead of just tapping items on the screen, pressing the screen with a small amount of force activates the context menu (if any) associated with the current interface controller. Context menus are optional. You use them to display actions related to the current screen. WatchKit displays the menu over your content, as shown in Figure 11-1.

Figure 11-1 A context menu with three items



A context menu can display up to four actions. Each action is represented by a title string and an image. Tapping an action's image dismisses the menu and executes the action method associated with that menu item. Tapping anywhere else dismisses the menu without any further action.

Designing Your Menu Items

Each menu item consists of a tappable area and a title. The tappable area contains a circular background, on top of which sits an image that you provide. The image must be a template image, where the alpha channel defines the shape to draw on top of the background. Opaque portions of the image appear as black, and fully or partially transparent portions let the background color show through.

The template images you provide should be smaller than the circular background on which they sit. For more information about the size of menu images and guidelines for how to create them, see *Apple Watch Human Interface Guidelines* .

Adding a Context Menu to an Interface Controller

You configure an interface controller's context menu at design time, but you can also add and remove menu items at runtime. At design time, edit your storyboard to include the menu items that you always want to be present on the menu for a given interface controller. When you initialize your interface controller later, you can add menu items to supplement the ones you created in your storyboard. Menu items you add programmatically can also be removed. The total number of menu items in a menu cannot exceed four, regardless of whether you included them in your storyboard or added them programmatically.

To add a context menu to an interface controller

1. Open your storyboard file.
2. Drag a menu object from the library and add it to your interface controller scene.

   The initial menu contains a single menu item.

3. Drag up to three more items from the library to your menu.

   You can also use the Attributes inspector of the menu to set the number of items. The items you add cannot be removed programmatically later.

4. For each item, use the Attributes inspector to specify the menu's title and image. Both are required.
5. Connect each menu item to an action method in your interface controller class.

   Menu action methods have the following format:

   ```
   - (IBAction)doMenuItemAction
   ```

6. Save your storyboard file.

To add menu items at runtime, call the `addMenuItemWithImage:title:action:` or `addMenuItemWithImageNamed:title:action:` method of your interface controller object. The items you specify are added to the ones in your storyboard file. Items added programmatically remain attached to the menu until you explicitly remove them or until your interface controller is deallocated.

Handling Taps in a Menu Item

When the user taps a menu item, WatchKit dismisses the menu and calls the associated action method. You define action methods in your interface controller using the following syntax:

```
- (IBAction)doMenuItemAction {
    // Handle menu action.
}
```

```
@IBAction func doMenuAction() {
    // Handle menu action.
}
```

If any state information is required to perform the action, it is your responsibility to store and maintain that information in your interface controller object. For example, if an action relies on the currently selected row of a table, your interface controller must include a variable to track the most recently selected row. And, if you want to request more information from the user after tapping a menu action, your action method must present a modal interface controller.

Settings

Preferences and settings are data values that change infrequently and that you use to configure your app's behavior or appearance. If your WatchKit app uses preferences for its configuration, you can add a WatchKit–specific settings bundle to your project to present those settings to the user. This settings bundle lives inside your containing iOS app, and the settings themselves are displayed by the Apple Watch app on the user's iPhone.

A WatchKit settings bundle works in the same way that an iOS settings bundle works. The settings bundle defines the controls you want displayed by the system and the name of the preference that each control modifies. The Apple Watch app on the user's iPhone takes your settings bundle information and uses it to display the actual controls to the user. When the user changes the value of a control, the system updates the underlying preference value.

For general information about how settings bundles work, see *Preferences and Settings Programming Guide* .

Creating Your WatchKit Settings Bundle

To add a WatchKit settings bundle to your Xcode project, do the following:

1. Select File > New > File.
2. In the Apple Watch section, select WatchKit Settings Bundle and click Next.
3. Create the settings bundle with the name `Settings-Watch.bundle` and add it to your iOS app target.

   Naming the bundle `Settings-Watch.bundle` is required to distinguish it from your iOS app's settings bundle (if any).

The initial contents of the WatchKit settings bundle are the same as for an iOS app's settings bundle and are shown in Listing 12-1.

Listing 12-1 Contents of a WatchKit settings bundle

```
Settings-Watch.bundle/
   Root.plist
   en.lproj/
      Root.strings
```

For information on how to configure the contents of your settings bundle, see Implementing an iOS Settings Bundle. For detailed information about the keys you can include in a Settings bundle, see *Settings Application Schema Reference* .

Enabling Access to Preference Values for Your WatchKit Extension

WatchKit settings must be stored in a shared group container that is accessible to both your iOS app and your WatchKit extension. Because the WatchKit settings bundle resides in your iOS app, the system writes preference values to the iOS app's container by default. To make settings accessible to your WatchKit extension, you must make the following configuration changes to your project:

- ✓ Enable the App Groups capability for both your iOS app and WatchKit extension.

  Select the same group identifier for both.

- ✓ Add the `ApplicationGroupContainerIdentifier` key to the `Root.plist` file of your WatchKit settings bundle.

  Set the value of this key to the same identifier you used when configuring the App Groups capability. You do not need to include this key in the property lists for any child panes.

Accessing Settings at Runtime

To access preferences stored in a group container, create your `NSUserDefaults` object using the `initWithSuiteName:` method. Specify the string you used for your group container identifier when calling that method. You can then use the user defaults object to access preference values. Listing 12-1 shows an example that accesses a custom group.

Listing 12-2 Accessing preferences in a shared group container

```
NSUserDefaults *defaults = [[NSUserDefaults alloc]
initWithSuiteName:@"group.example.MyWatchKitApp"];
BOOL enabled = [defaults boolForKey:@"enabled_preference"];
```

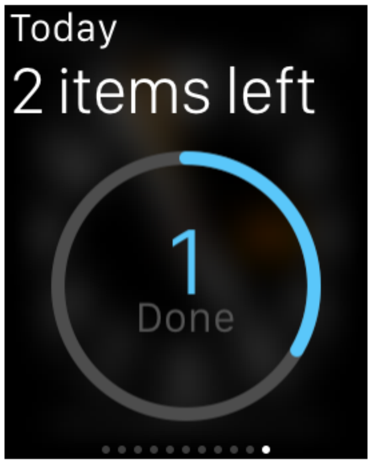
For more information on how to access preferences values, see *NSUserDefaults Class Reference* .

Glances

Glance Essentials

A glance is a supplemental way for the user to view important information from your app. Not all apps need a glance. A glance provides immediately relevant information in a timely manner. For example, the glance for a calendar app might show information about the user's next meeting, whereas the glance for an airline app might display gate information for an upcoming flight. Figure 13-1 shows the glance for the Lister sample app, which displays the number of completed items and the number of remaining items on the user's to-do list.

Figure 13-1 A glance interface for the Lister sample app



Glances are delivered as part of your WatchKit app and WatchKit extension. Your glance's interface resides in your WatchKit app's existing storyboard file, and that interface is managed by a custom `WKInterfaceController` object. However, the only job of your glance interface controller is to set the contents of the glance. Glances do not support interactivity—tapping on a glance automatically launches your WatchKit app.

The Glance Life Cycle

The life cycle of a glance interface controller is the same as for other interface controllers except that glance interface controllers are initialized early so that the glance be displayed quickly to the user. Because a nontrivial amount of time can elapse between the initialization and display of a glance, include checks in your `willActivate` method to make sure the information being displayed is up to date.

For information about the life cycle of interface controllers, see WatchKit Extension Life Cycle (page 20).

Glance Interface Guidelines

Xcode provides fixed layouts for arranging the contents of your glance. After choosing a layout that works for your content, use the following guidelines to fill in that content:

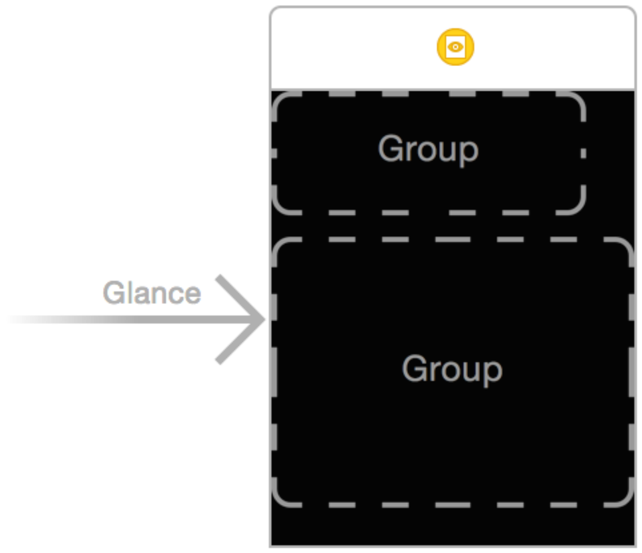
- **Design your glance to convey information quickly.** Do not display a wall of text. Make appropriate use of graphics, colors, and animation to convey information.
- **Focus on the most important data.** A glance is not a replacement for your WatchKit app. Just as your WatchKit app is a trimmed down version of its containing iOS app, a glance is a trimmed down version of your WatchKit app.
- **Do not include interactive controls in your glance interface.** Interactive controls include buttons, switches, sliders, and menus.
- **Avoid tables and maps in your glance interface.** Although they are not prohibited, the limited space makes tables and maps less useful.
- **Be timely with the information you display.** Use all available resources, including time and location to provide information that matters to the user. And remember to update your glance to account for changes that occur between the time your interface controller is initialized and the time it is displayed to the user.
- **Use the system font for all text.** To use custom fonts in your glance, you must render that text into an image and display the image.

Because an app has only one glance interface controller, that one controller must be able to display the data you want.

Managing Your Glance Interface

When adding a WatchKit app target to your Xcode project, you can specify whether you want a glance interface. You can also add a glance to your project later if you forget to add one initially. A glance interface controller has a slightly different appearance in your app's storyboard. Specifically, it has a glance entry point object attached to it, and it has a default layout, as shown in Figure 14-1.

Figure 14-1 An interface controller with the glance entry point object



To configure the contents of your glance at runtime, you use a custom `WKInterfaceController` subclass. You implement this subclass in the same way that you implement other interface controller classes in your WatchKit app.

Adding a Glance Interface to Your App

When creating the WatchKit App target for your app, select the Include Glance Scene option to create the corresponding files you need to implement your glance interface. Xcode provides you with a glance storyboard scene and a custom glance interface controller class. If you did not select this option when creating your target, configure your project manually.

To create a glance interface manually

1. In your project, create a new `WKInterfaceController` subclass.

   Create the new source file and add it to your WatchKit extension target. Give your subclass an appropriate name, like `GlanceInterfaceController`.

2. In your storyboard file, drag a glance interface controller to your storyboard.

   The scene for your new glance interface controller has the glance entry point object attached to it and looks similar to the scene shown in Figure 14-1 (page 61).

3. Select the glance interface controller in your storyboard and open the Identity inspector.
4. Set the class of your glance interface controller to the class you created in Step 1.

WatchKit apps may have only one glance interface. Do not add more than one glance interface controller to your app's storyboard.

Implementing and Updating a Glance Interface Controller

The implementation of a glance interface controller is relatively simple because its only job is to set the content of the labels and images in the glance.

- Use the `init` and `awakeWithContext:` methods to initialize your glance interface and to set the initial values for its labels and images.
- Use the `willActivate` to update the glance interface as needed before it is displayed onscreen.

To update the content of a glance after it is onscreen, use an `NSTimer` object to perform periodic updates. You do not need to update `WKInterfaceDate` and `WKInterfaceTimer` objects, which automatically update themselves.

Customizing App Launch from Your Glance

When the user taps a glance, Apple Watch launches the corresponding WatchKit app. Normally, launching the app displays its main interface controller. To customize the launch behavior of your app from a glance, do the following:

- In the glance interface controller:
  - Configure the glance normally in the `init` and `willActivate` methods.

    - Call the `updateUserActivity:userInfo:webpageURL:` method at some point and use the `userInfo` parameter to convey information about the state of the glance to your app. At launch time, your app can use that contextual data to display a different interface controller.
- In your app's main interface controller:
    - Implement the `handleUserActivity:` method. Use the provided `userInfo` dictionary to configure your UI appropriately.

Calling the `updateUserActivity:userInfo:webpageURL:` method tells WatchKit to call the main interface controller's `handleUserActivity:` method at launch time. In your implementation of the `handleUserActivity:` method, use the provided contextual data to configure your UI appropriately. For example, an app with a page-based interface might use the provided data to select which page to display initially.

Notifications

Notification Essentials

Apple Watch takes full advantage of the existing interactive notification support on iOS. If your iOS app supports notifications, Apple Watch displays those notifications at appropriate times. When one of your app's local or remote notifications arrives on the user's iPhone, iOS decides whether to display that notification on the iPhone or on the Apple Watch. For notifications sent to Apple Watch, the system lets the the user know subtly that a notification is available. If the user chooses to view the notification, the system displays an abbreviated version of the notification first, followed by a more detailed version. The user can dismiss the detailed notification, launch your WatchKit app, or act on the notification by tapping an available action button.

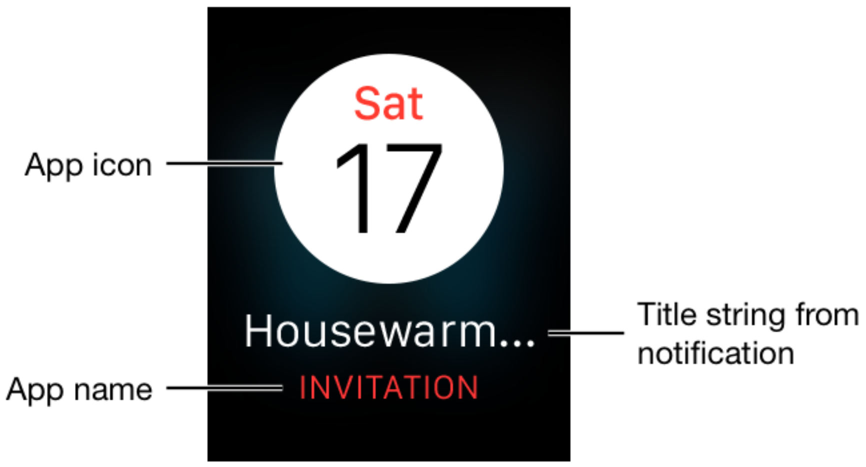
Apps are not required to do anything to support notifications. The system provides a default notification interface that displays the alert message from the notification. However, apps can customize the notification interface and include custom graphics, content, and branding.

Note: Apple Watch displays local and remote notifications only if the containing iOS supports them. For information about how to support local and remote notifications in your iOS app, see *Local and Remote Notification Programming Guide* .

The Short-Look Interface

When the user first looks at a notification, the system displays the short-look interface, an example of which is shown in Figure 15-1. The short-look interface is a nonscrolling screen that cannot be customized. The system uses a template to display the app name and icon along with the title string stored in the local notification or remote notification payload. If the user continues to look at the notification, the system transitions quickly from the short-look interface to the long-look interface.

Figure 15-1 A short-look interface



The title string used in the short look provides a brief indication of the intent of the notification. For local notifications, you specify this string using the `alertTitle` property of the `UILocalNotification` object. For remote notifications, add the `title` key to the `alert` dictionary inside the payload. For more information about adding a title string to your notifications, see *Local and Remote Notification Programming Guide* .

The Long-Look Interface

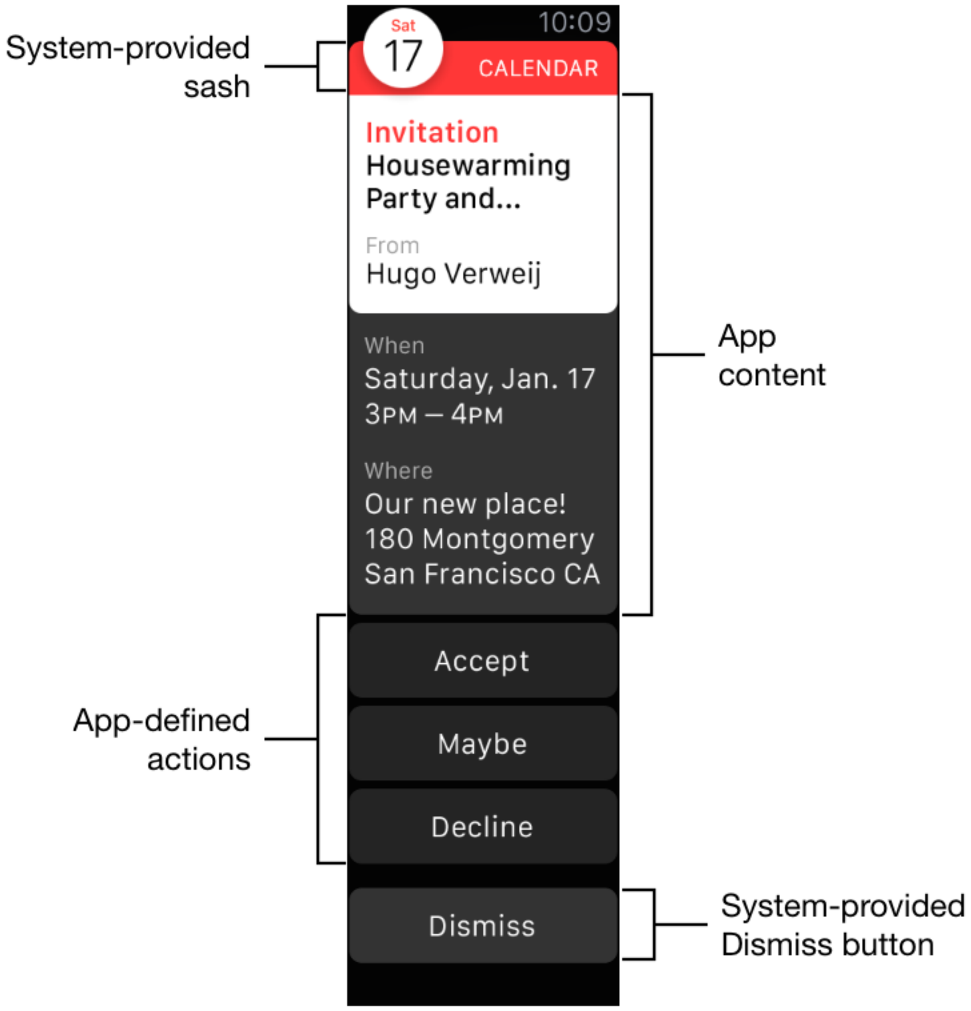
The long-look interface is a scrollable screen that displays the notification's content and any associated action buttons. If you do not provide a custom notification interface, Apple Watch displays a default interface that includes your app icon, the title string of the notification, and the alert message. If you provide a custom notification interface, Apple Watch displays your custom interface instead.

The long-look notification interface is divided into three areas:

- The sash is an overlay that contains the app icon and app name. The sash color is configurable.
- The content area contains the detailed information about the incoming notification. For information on how to customize the content in this area, see Custom Notification Interfaces (page 71).
- The bottom area contains a Dismiss button and any action buttons registered by the containing iOS app.

Figure 15-2 shows an example of a long-look notification containing several action buttons.

Figure 15-2 A long-look notification interface



Tapping the app icon launches your WatchKit app. Tapping one of the app-defined action buttons delivers the selected action to either your iOS app or your WatchKit app. Foreground actions are delivered to your WatchKit app and extension, and background actions are delivered to your iOS app. Tapping the Dismiss button closes the notification interface without any further actions. Tapping anywhere else does nothing.

For information about how to provide a custom long-look interface for your app, see Custom Notification Interfaces (page 71).

Adding Action Buttons to Notifications

Action buttons save time for the user by offering some standard responses for a notification. Apple Watch makes use of the interactive notifications registered by your iOS app to display action buttons. In iOS 8 and later, apps are required to register the types of notification-generated alerts they display using a `UIUserNotificationSettings` object. When registering that information, the app can also register a set of custom notification categories, which include the actions that can be performed for that category. Apple Watch uses this category information to add the corresponding action buttons to the long-look interface.

In Listing 15-1, a method registers the settings and categories for a sample iOS app. This method is implemented by the containing iOS app, not by the WatchKit extension, and is called by the iOS app delegate at launch time. The implementation is written in Swift and shows the creation and registration of an "invitation" category that contains actions to accept or decline a meeting invitation.

Listing 15-1 Registering actions in a containing iOS app (partial implementation)

```
func registerSettingsAndCategories() {
    var categories = NSMutableSet()

    var acceptAction = UIMutableUserNotificationAction()
    acceptAction.title = NSLocalizedString("Accept", comment: "Accept invitation")
    acceptAction.identifier = "accept"
    acceptAction.activationMode = UIUserNotificationActivationMode.Background
    acceptAction.authenticationRequired = false

    var declineAction = UIMutableUserNotificationAction()
    declineAction.title = NSLocalizedString("Decline", comment: "Decline invitation")
    declineAction.identifier = "decline"
    declineAction.activationMode = UIUserNotificationActivationMode.Background
    declineAction.authenticationRequired = false

    var inviteCategory = UIMutableUserNotificationCategory()
    inviteCategory.setActions([acceptAction, declineAction],
                  forContext: UIUserNotificationActionContext.Default)
    inviteCategory.identifier = "invitation"

    categories.addObject(inviteCategory)
    // Configure other actions and categories and add them to the set...

    var settings = UIUserNotificationSettings(forTypes: (.Alert | .Badge | .Sound),
                            categories: categories)

    UIApplication.sharedApplication().registerUserNotificationSettings(settings)
}
```

For information about how to configure categories and actions in your iOS app, see *Local and Remote Notification Programming Guide* .

Responding to Taps in Action Buttons

When the user taps an action button for a notification, the system uses the information in the registered `UIUserNotificationAction` object to determine how to process the action. Actions can be processed in the foreground or the background, with or without requiring user authentication. Foreground and background actions are processed differently:

- Foreground actions launch your WatchKit app and deliver the ID of the tapped button to the `handleActionWithIdentifier:forRemoteNotification:` or `handleActionWithIdentifier:forLocalNotification:` method of your main interface controller.
- Background actions launch the containing iOS app in the background so that it can process the action. Information about the selected action is delivered to the `application:handleActionWithIdentifier:forRemoteNotification:completionHandler:` or `application:handleActionWithIdentifier:forLocalNotification:completionHandler:` method of the app delegate.

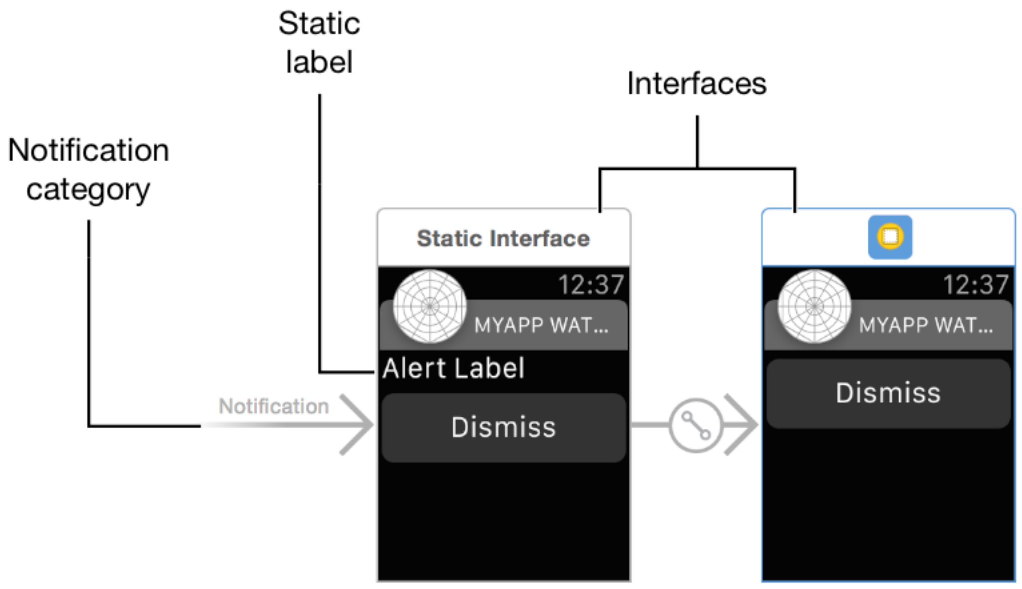
For foreground actions, it is important to note that your `WKUserNotificationInterfaceController` subclass does not process the action. Selecting a foreground action launches your app and loads the interface controller for your app's main entry point. It is this initial interface controller that is responsible for processing any actions. That interface controller must implement the `handleActionWithIdentifier:forRemoteNotification:` and `handleActionWithIdentifier:forLocalNotification:` methods (as appropriate) to process actions.

Managing a Custom Long Look Interface

The custom long-look notification interface consists of two separate interfaces: one static and one dynamic. The static interface is required and is a simple way to display the notification's alert message and any static images and text that you configure at design time. The dynamic interface is optional and gives you a way to customize the display of your notification's content.

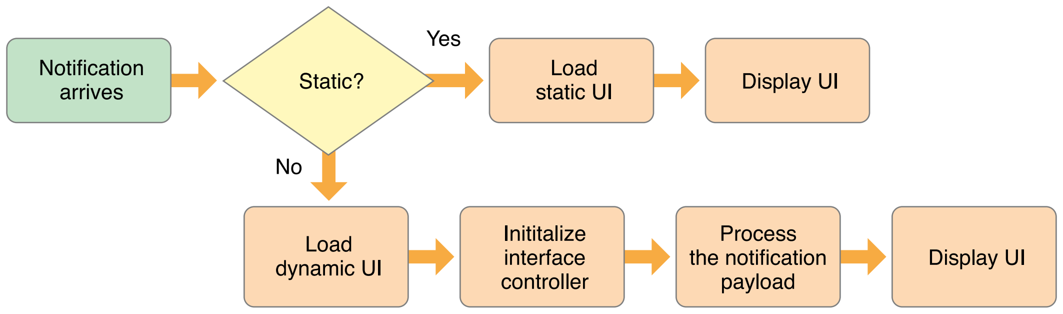
When you add a new notification interface controller to your storyboard file, Xcode creates only the static interface initially. You can add a dynamic interface by enabling the Has Dynamic Interface property of the notification category object in your storyboard. (Add a dynamic interface only when you require customizations that go beyond what is possible with the static interface.) Figure 16-1 shows both the unmodified static and dynamic interface scenes from a storyboard file. The static and dynamic scenes are associated with the same notification type, which you configure using the notification category object attached to the static scene.

Figure 16-1 Static and dynamic notification interfaces



When a notification of the correct type arrives, WatchKit chooses whether to display your static or dynamic interface based on several factors. WatchKit automatically displays the static interface when a dynamic interface is not available, there is not enough power to warrant displaying the dynamic interface, or you explicitly tell WatchKit not to display the dynamic interface. In all other cases, WatchKit displays your dynamic interface. After making the choice, WatchKit loads the appropriate storyboard resources and prepares the interface as shown in Figure 16-2. The loading process for the dynamic interface is mostly the same as for your app's other interface controllers, with the exception of processing the notification payload, which is specific to notification interface controllers.

Figure 16-2 Preparing the notification interface



Adding a Custom Notification Interface to Your App

When creating the WatchKit App target for your app, select the Include Notification Scene option to create the corresponding files you need to implement one of your notification interfaces. Xcode provides you with an empty storyboard scene and a custom subclass to use for your notification interface controller. If you did not enable this option when creating your target or if you need to create additional notification interfaces, create the notification interface manually.

To create a new notification interface, drag a notification interface controller object to your storyboard file. The new interface contains only the static interface controller initially. To add a dynamic interface, you must perform some additional configuration steps.

Configuring a dynamic notification interface controller

1. In your project, create a new `WKUserNotificationInterfaceController` subclass.

   Create the new source file and add it to your WatchKit extension target. Give your subclass an appropriate name that distinguishes it from your other notification interface controllers.

2. Enable the Has Dynamic Interface attribute of the notification category. This step adds the dynamic scene to your storyboard file.
3. Set the class of your dynamic notification interface controller to the class you created in Step 1.

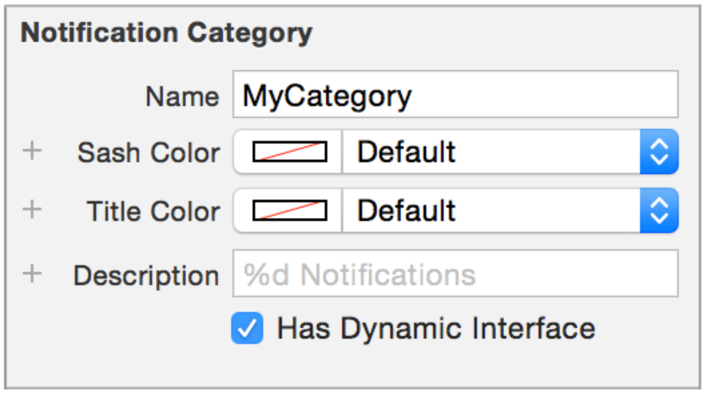
Apps may have multiple notification interfaces, which you differentiate using categories. In your storyboard file, use the Notification Category object to specify the category name associated with each scene. WatchKit uses the category value to determine which scene to load at runtime. If an incoming notification does not have a category, WatchKit loads the scene whose category name is set to `default`.

Configuring the Category of a Custom Interface

Each notification interface must have an assigned notification category that tells Apple Watch when to use it. Incoming notifications can include a `category` key in their payload, the value of which is a string you define. Apple Watch uses that category to decide which of your notification scenes to display. If an incoming notification does not include a category string, Apple Watch displays the notification interface configured with the `default` category.

To assign a notification type to your notification interface, select the Notification Category object in your storyboard and go to the Attributes inspector, shown in Figure 16-3. Enter the category name in the Name field of the inspector. You can also set the sash color and title text color for your custom interface in the inspector.

Figure 16-3 Configuring the notification type information



Notification Category

Name: MyCategory
Sash Color: Default
Title Color: Default
Description: %d Notifications
✓ Has Dynamic Interface

When generating remote notifications, your server specifies the notification type by including the `category` key in the `aps` dictionary of the payload. For local notifications, you specify this value in the `category` property of the `UILocalNotification` object.

Note: The category string also defines which action buttons (if any) are appended to the end of your notification interface. For more information on supporting custom actions, see Adding Action Buttons to Notifications (page 69).

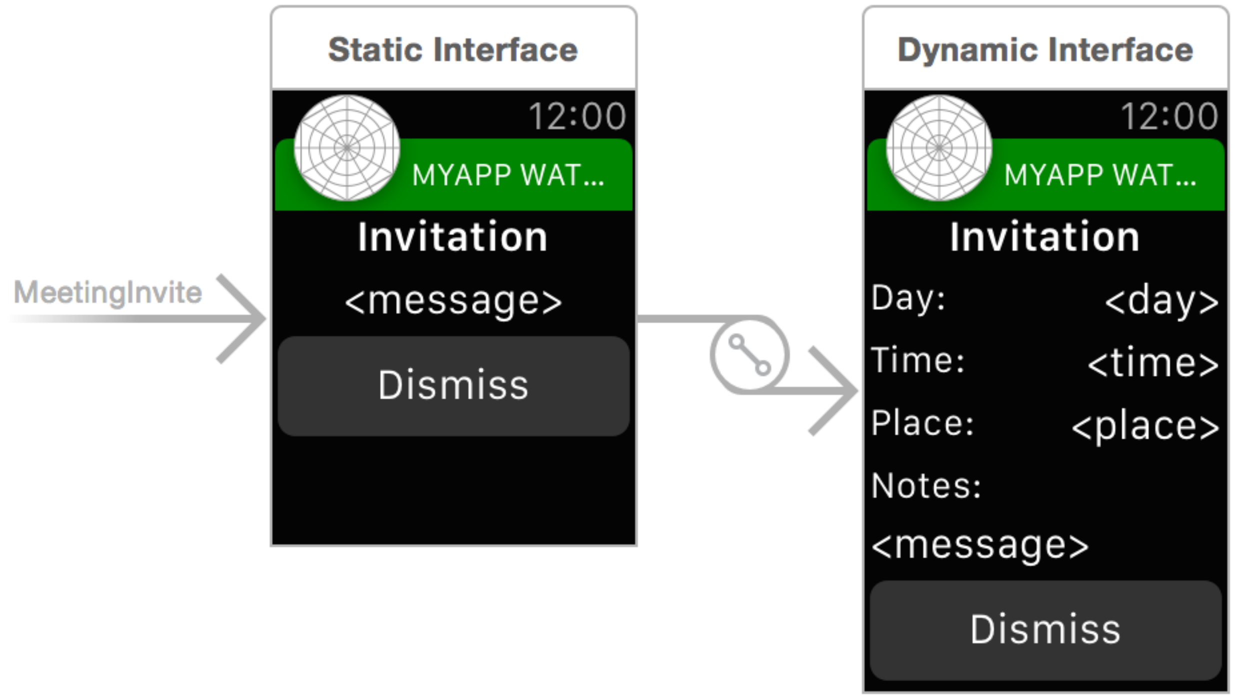
Configuring a Static Notification Interface

Use the static notification interface to define a simple version of your custom notification interface. The purpose of a static interface is to provide a fallback interface in the event that your WatchKit extension is unable to configure the dynamic interface in a timely manner. The rules for creating a static interface are as follows:

- All images must reside in the WatchKit app bundle.
- The interface must not include controls, tables, maps, or other interactive elements.
- The interface's `notificationAlertLabel` outlet must be connected to a label. The label's contents are set to the notification's alert message. The text for all other labels does not change.

Figure 16-4 shows the configuration of the static and dynamic scenes for a custom notification interface in a calendar app. The notification arrow points to the static scene, which contains a custom icon and two labels. In the static interface, the label containing the string `<message>` is the one associated with the `notificationAlertLabel` outlet and therefore receives the notification's alert message at runtime.

Figure 16-4 Static and dynamic scenes for a single notification type



Configuring the Dynamic Notification Interface

A dynamic notification interface lets you provide a more enriched notification experience for the user. With a dynamic interface, you can display more than just the alert message. You can incorporate additional information, configure more than one label, display dynamically generated content, and so on.

To implement a dynamic notification interface, you must create a custom `WKUserNotificationInterfaceController` subclass. How you implement that subclass determines what information is displayed in the notification interface.

Designing Your Dynamic Interface

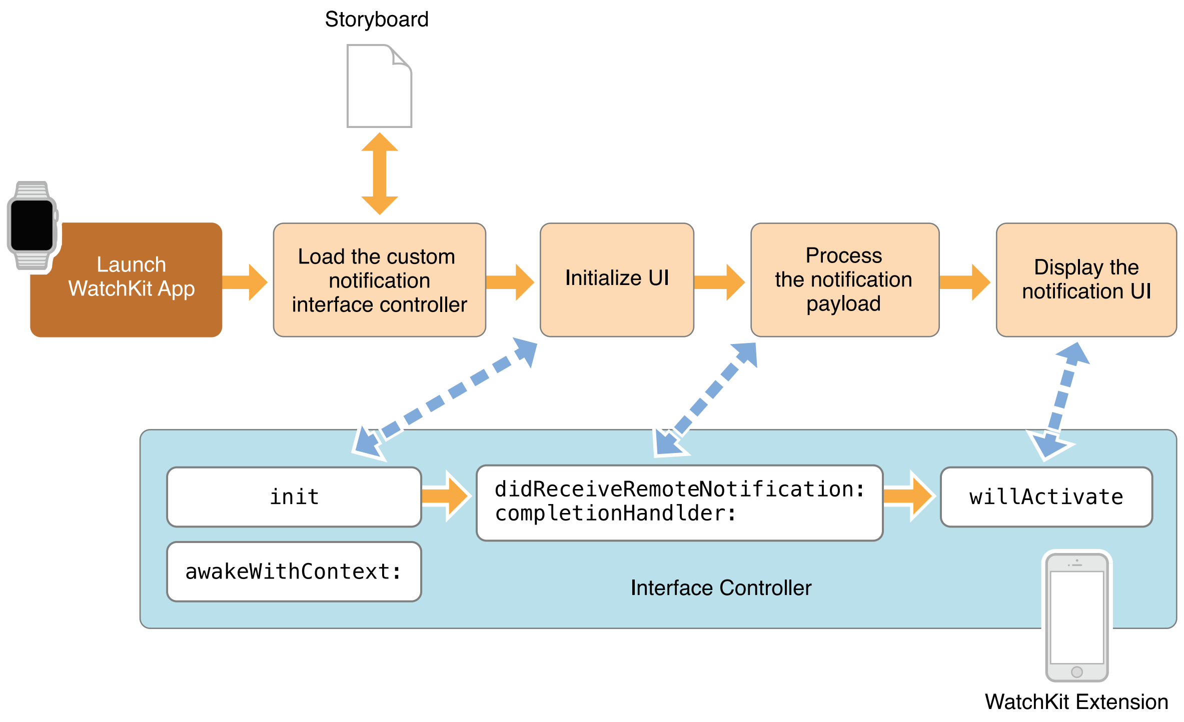
Configure your dynamic interface as you would any other interface controller scene. Include outlets in your subclass to refer to labels, images, and other objects in the scene, and use those outlets to configure the contents of the scene at runtime. Tapping your notification interface launches the app, so notification interfaces should not contain interactive controls.

- Use labels, images, groups, and separators for most of your interface.
- Include tables and maps only as needed in your interface.
- Do not include buttons, switches, or other interactive controls.

Configuring Your Dynamic Interface at Runtime

When a notification of the appropriate type arrives, WatchKit displays the appropriate scene from your storyboard and asks your WatchKit extension to instantiate the corresponding `WKUserNotificationInterfaceController` subclass. Figure 16-5 shows the steps that WatchKit takes to prepare your interface. After initializing the notification interface controller, WatchKit delivers the payload data to it using either the `didReceiveRemoteNotification:withCompletion:` or the `didReceiveLocalNotification:withCompletion:` method. You use the payload data to configure the rest of your notification interface and then call the provided completion handler block to let WatchKit know that your interface is ready.

Figure 16-5 Configuring the dynamic notification interface


Storyboard
Launch WatchKit App
Load the custom notification interface controller
Initialize UI
Process the notification payload
Display the notification UI
init
awakeWithContext:
didReceiveRemoteNotification: completionHandlder:
willActivate
Interface Controller
WatchKit Extension

Always use the `didReceiveRemoteNotification:withCompletion:` and `didReceiveLocalNotification:withCompletion:` methods to configure your notification interface. When implementing either method, execute the provided completion handler as soon as you have configured the interface. If you wait too long, Apple Watch abandons the attempt to display your dynamic interface and displays the static interface instead.

Listing 16-1 shows a sample implementation of the `didReceiveRemoteNotification:withCompletion:` method. This method is implemented by a fictional calendar app that sends remote notifications for new meeting invitations. The method extracts data from the remote notification's payload and uses that data to set values for labels in the notification interface. For brevity, the example assumes that the server always includes an appropriate value for each key. But your own code should perform whatever error checking is necessary to ensure the payload data is valid. After configuring the labels, the method calls the completion handler to let WatchKit know that the custom interface is ready to be displayed.

Listing 16-1 Configuring the custom interface from a remote notification

```
// Standard remote notification payload keys.
NSString* apsKeyString = @"aps";
NSString* titleKeyString = @"title";

// Payload keys that are specific to the app.
NSString* customDataKey = @"cal";
NSString* invitationDateKey = @"date";
NSString* invitationLocationKey = @"loc";
NSString* invitationNotesKey = @"note";

- (void)didReceiveRemoteNotification:(NSDictionary *)remoteNotification
withCompletion:(void(^)(WKUserNotificationInterfaceType interface)) completionHandler
 {
    // Get the aps dictionary from the payload.
    NSDictionary* apsDict = [remoteNotification objectForKey:apsKeyString];

    // Retrieve the title of the invitation.
    NSString* titleString = [apsDict objectForKey:titleKeyString];
    [self.titleLabel setText:titleString];

    // Extract the date and time from the custom section of the payload.
    // The date/time information is stored as the number of seconds since 1970.
    NSDictionary* customDataDict = [remoteNotification objectForKey:customDataKey];
    NSNumber* dateValue = [customDataDict objectForKey:invitationDateKey];
    NSDate* inviteDate = [NSDate dateWithTimeIntervalSince1970:[dateValue
doubleValue]];

    // Format the date and time strings.
    NSDateFormatter *dateFormatter = [[NSDateFormatter alloc] init];

    // Call a custom method to get the localized date format string for the user.
    // The default date format string is "EEE, MMM d".
    dateFormatter.dateFormat = [self dateFormatForCurrentUser];
    NSString *formattedDateString = [dateFormatter stringFromDate:inviteDate];

    // Call a custom method to get the localized time format string for the user.
    // The default time format string is "h:mm a".
    dateFormatter.dateFormat = [self timeFormatForCurrentUser];
    NSString *formattedTimeString = [dateFormatter stringFromDate:inviteDate];

    // Set the date and time in the corresponding labels.
    [self.dateLabel setText:formattedDateString];
    [self.timeLabel setText:formattedTimeString];

    // Set the location of the meeting.
    NSString* locationString = [customDataDict objectForKey:invitationLocationKey];
    [self.locationLabel setText:locationString];

    // Set the invitation's notes (if any).
    NSString* notesString = [customDataDict objectForKey:invitationNotesKey];
    [self.notesLabel setText:notesString];

    // Tell WatchKit to display the custom interface.
    completionHandler(WKUserNotificationInterfaceTypeCustom);
}
```

When calling the completion handler block, if you want WatchKit to display your static interface instead, specify the `WKUserNotificationInterfaceTypeDefault` constant.

Note: Notification interfaces support only the use of the system font for labels and other text. If you need to display text in a custom font, render that text into an image and display the image.

Testing Your Custom Interface

When you are ready to test your dynamic interface in the simulator, create a custom build scheme for running your notification interface if you have not done so already. When configuring the interface, specify the JSON data file containing the test data you want delivered to your interface. Xcode provides custom JSON files for specifying this data.

For more information about setting up the build schemes and configuring your payload data, see The Build, Run, Debug Process (page 13).

Document Revision History

This table describes the changes to *Apple Watch Programming Guide* .

Date	Notes
2015-03-09	New document that describes how to use the WatchKit framework to write apps for Apple Watch.