HACKING WITH SWIFT

HACKING WITH watchOS

COMPLETE TUTORIAL COURSE

Learn to make watchO
apps with real-wo
Swift projects

FREE SAMPLE

Paul Hudson

Project 1

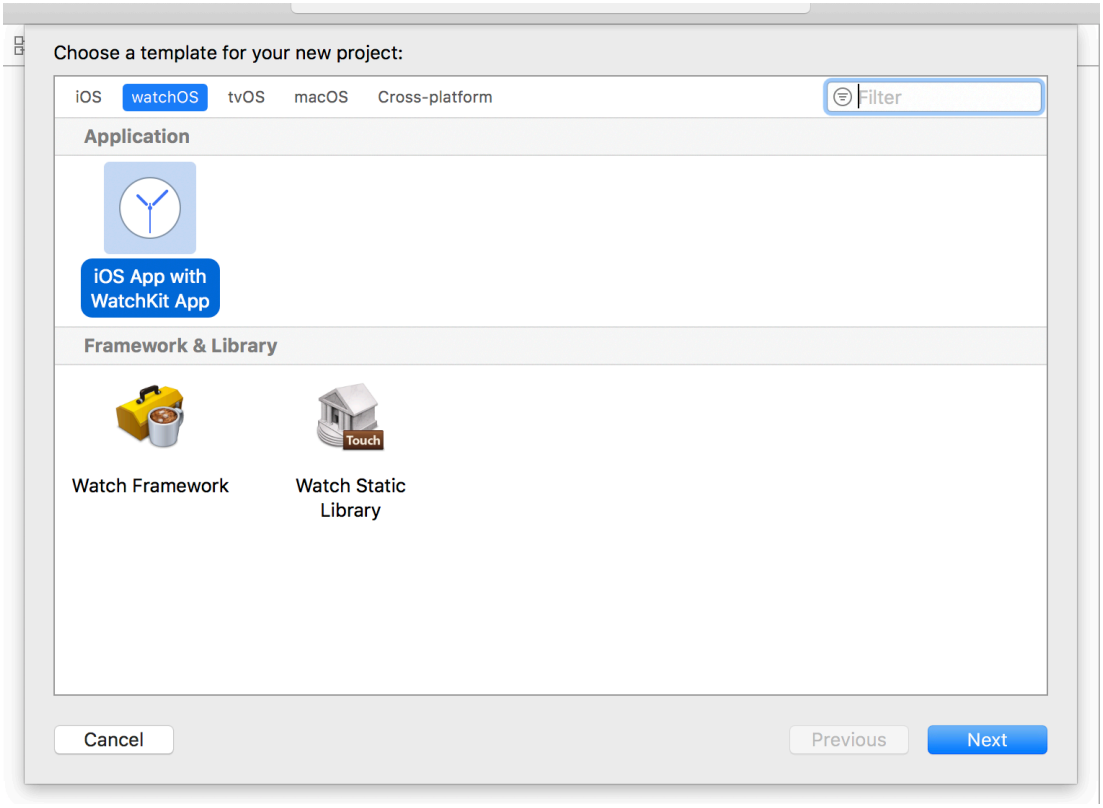
NoteDictate

Setting up

In this project you'll produce an application that lets users dictate notes directly to their watch, then browse through and read previous notes. In doing so you'll learn how to show tables of data for the user to choose from, how to move between different screens of information, and how to convert the user's voice into text so it can be read.

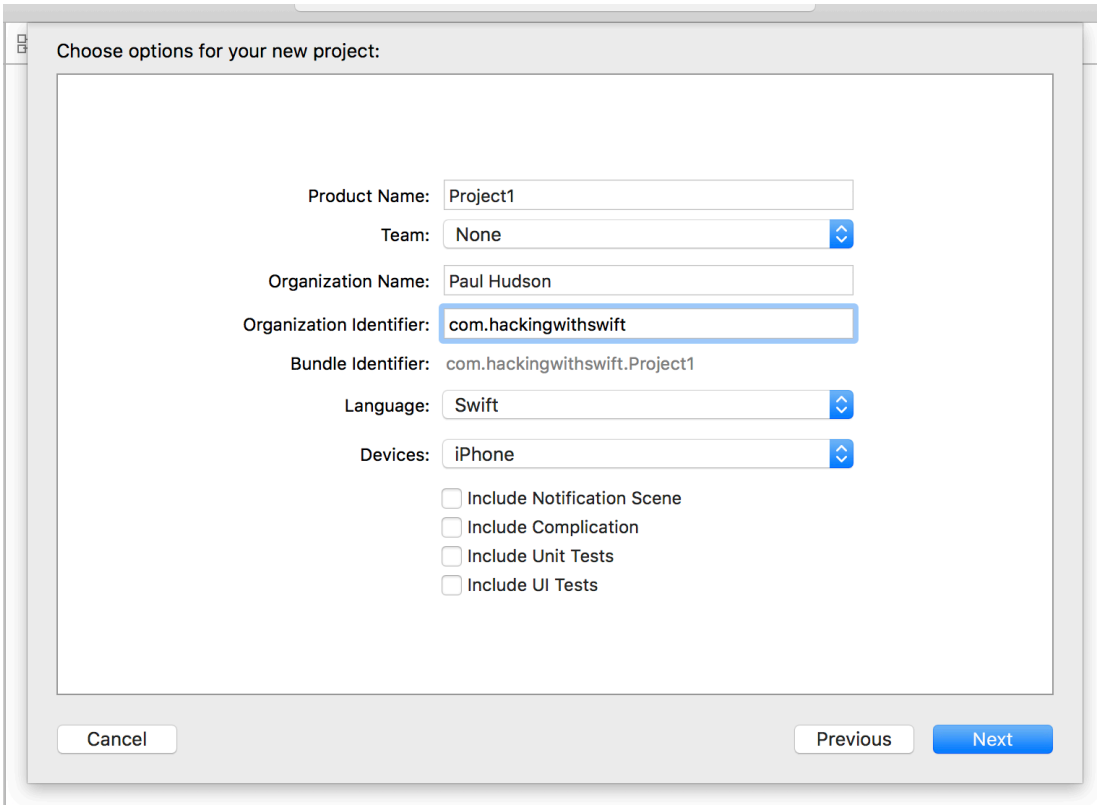
Yes, I realize that sounds quite a lot, but don't worry: this project is deliberately designed to take it slow so everyone can learn. The pace picks up in later projects, but if you hit problems you can always refer back here as a reminder.

Let's get started: launch Xcode, and choose "Create a new project" from the welcome screen. You'll be asked to choose a template for the new project, so please choose watchOS > iOS App with WatchKit App.



For Product Name enter Project1, then make sure you have Swift selected for language. This screen contains four different checkboxes that affect what kind of template you're given. None of the projects in this book use the bottom two, so they should always be unchecked. For this project, I'd like you to uncheck all four.

One of the fields you'll be asked for is "Organization Identifier", which is a unique identifier usually made up of your personal website domain name in reverse. For example, I would use **com.hackingwithswift** if I were making an app. You'll need to put something valid in there if you're deploying to devices, but otherwise you can just use **com.example**.



Now click Next again and you'll be asked where you want to save the project – your desktop is fine. Once that's done, you'll be presented with the example project that Xcode made for you.

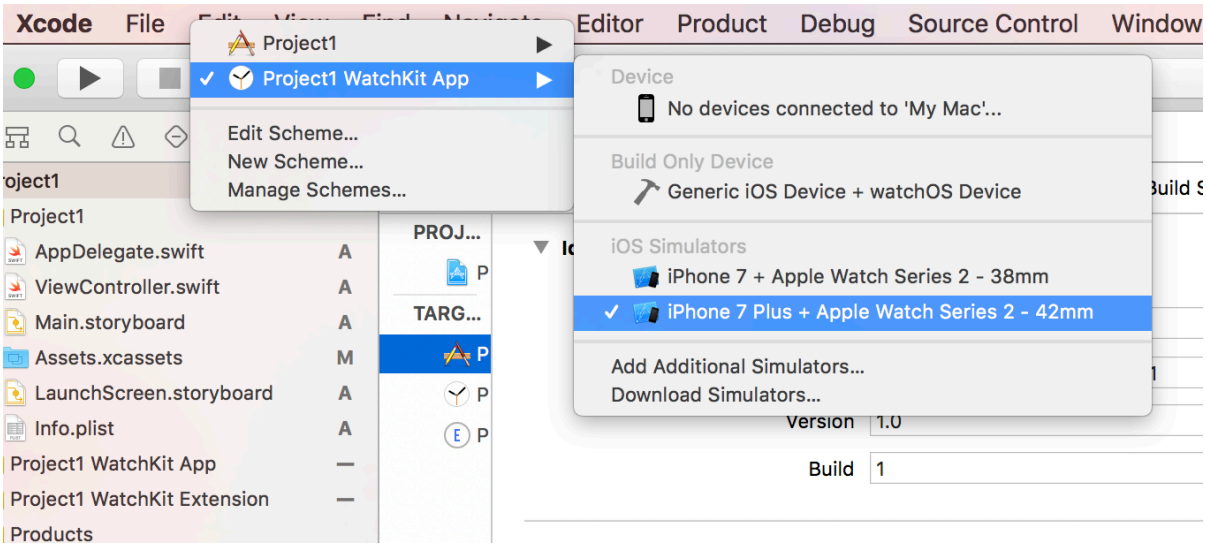
The first thing we need to do is make sure you have everything set up correctly, and that means running the project as-is.

This is important: Because watchOS apps ship inside iOS apps, your template gives you both a watchOS app and an iOS app. When you click the Play triangle button at the top-left of the Xcode window it will launch the iOS app by default. That's often what you want, but for most of this book we want to choose something else.

In the top-left corner of Xcode you'll see a play button and a pause button. Immediately to the right of that you'll see "Project1 > iPhone 7 Plus", which is the active build scheme – the thing

that will get built and run when you click the play button.

What I'd like you to do is click the Project1 part of that button, and choose Project1 WatchKit App > iPhone 7 Plus + Apple Watch Series 2 - 42mm.



What you've done is tell Xcode you want to build and run the watchOS app, rather than the iOS app it's bundled inside. So, go ahead and press the play button now to build and run your app. It will take a little while at first because Xcode will launch the iOS Simulator and the Watch Simulator, both of which have to boot up their respective operating systems before your app can run.

Eventually you'll see our app, such as it is. Our "app" just shows an empty window – it does nothing at all, at least not yet. You'll see the time in the top-right corner, which is automatically inserted by watchOS, cutting your available screen space even further.

You'll be starting and stopping projects a lot as you learn, so there are three basic tips you need to know:

- You can run your project by pressing Cmd+R. This is equivalent to clicking the play button.
- You can stop a running project by pressing Cmd+. when Xcode is selected.
- If you have made changes to a running project, just press Cmd+R again. Xcode will prompt you to stop the current run before starting another. Make sure you check the "Do not show this message again" box to avoid being bothered in the future.

Obviously this is a pretty dull app as it stands, so let's dive right into making something better…

Designing our interface

watchOS makes extensive use of tables, just like iOS, macOS, and tvOS. These are vertically scrolling lists of choices for the user to select from. On iOS you can see these in the Settings app, the Mail app, in most Twitter apps, and more. On watchOS you can also find a table in the Settings app, as well as the Reminders app, the Phone app, and more.

If you've used table views on other Apple platforms, the watchOS approach will surprise you with its simplicity: you tell it up front precisely how many rows you want and what's in them, and that's it. All rows in your table get loaded immediately, even though they aren't visible, so you don't need to worry about the table rows once they are created.

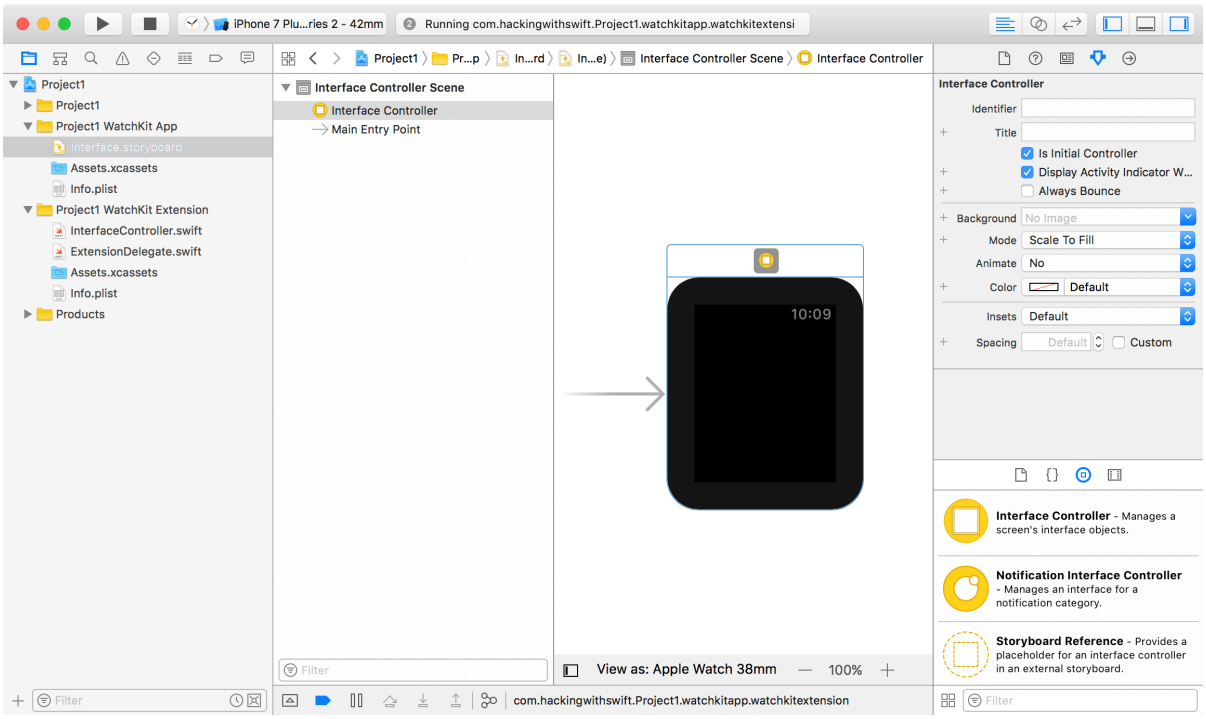
Keep in mind what I said already about resource constraints: there isn't a lot of screen space, and creating all the rows up front has a cost attached to it. As a result of those two combined, Apple recommends you don't show more than 20 rows in your table at a time – any more just wouldn't be a pleasant experience.

On the left of your Xcode window should be the project navigator, showing you the files inside your project. If you don't see it, press Cmd+1 to show it now.

You'll see three groups inside there: "Project1" contains the code for your iOS app, "Project1 WatchKit App" contains the user interface for your watchOS app, and "Project1 WatchKit Extension" contains the code that drives the user interface.

For now, I'd like you to open the "Project1 WatchKit App" group, and inside there you'll see a file called Interface.storyboard. Please select that now, and you'll see Xcode switch into a new layout commonly called Interface Builder, or IB for short. This is a graphical user interface designer that lets you visually design how you want your app to look.

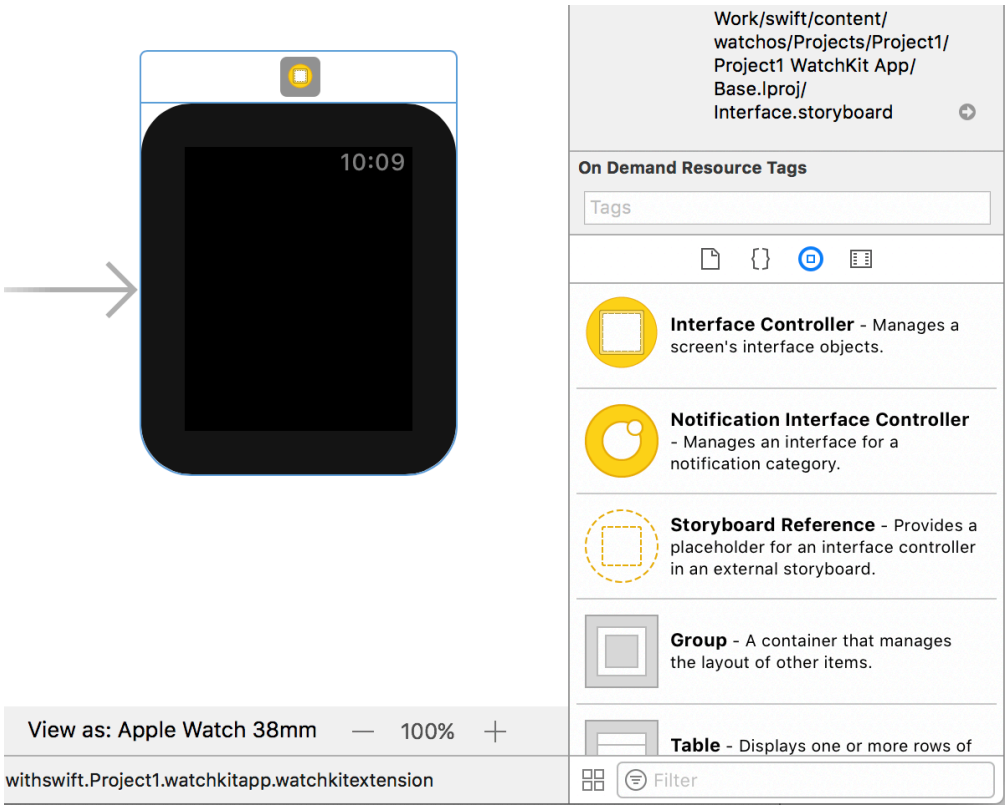
Now, on iOS, macOS, and tvOS, Interface Builder is optional: you can design your interfaces visually or do it all in code, depending on how you like to work. But on watchOS you don't have a choice: you either use IB, or you don't make apps for watchOS. This makes your life significantly easier, because there's only One Right Way to do things. It also makes *my* life easier, because there's only one thing to teach – hurray for simplicity!



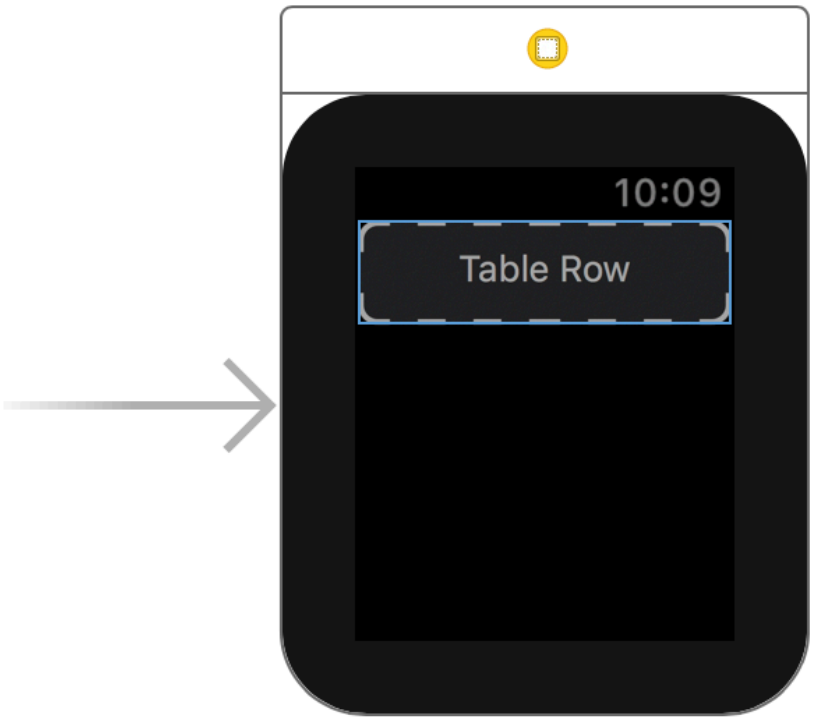
When you selected Interface.storyboard, you should see a large black square with rounded edges, mimicking the Apple Watch display. It's called a "storyboard" because you can add more screens and position them around each other, just like a movie storyboard.

For now, we're going to add only two things: a table that will show a list of notes the user has already dictated, and a button to let them add more.

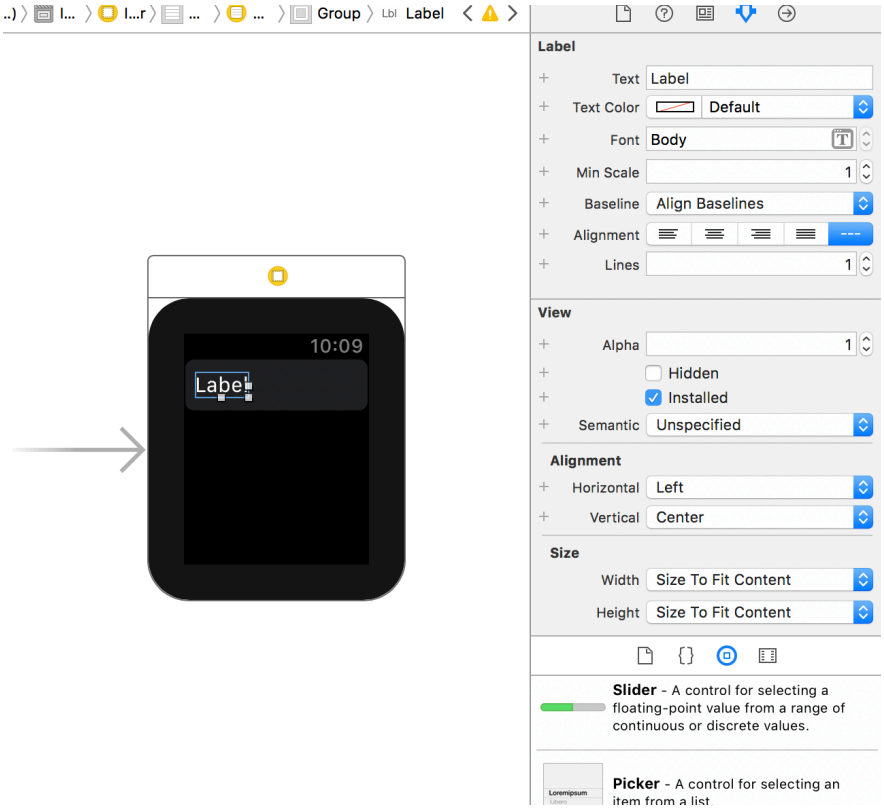
In the bottom-right of your Xcode window you should see the Object library – it starts with a yellow circle that says "Interface Controller" next to it. If you don't see that, press Ctrl+Alt +Cmd+3. Notice it has a "Filter" text box at the bottom, which is helpful to let you find controls quickly – just type a few letters of what you're looking for in there.



Look for "Table", and drag one into the black space on the canvas. You'll see "Table Row" appear in the Watch screen, which is where we can design what our table rows should look like. The table automatically takes up only the room it needs, so if it has three rows it will expand to fit them. Even better, watchOS automatically makes the whole screen scroll if your interfaces are too long, so if we added 20 items the user could just scroll down to see the others.



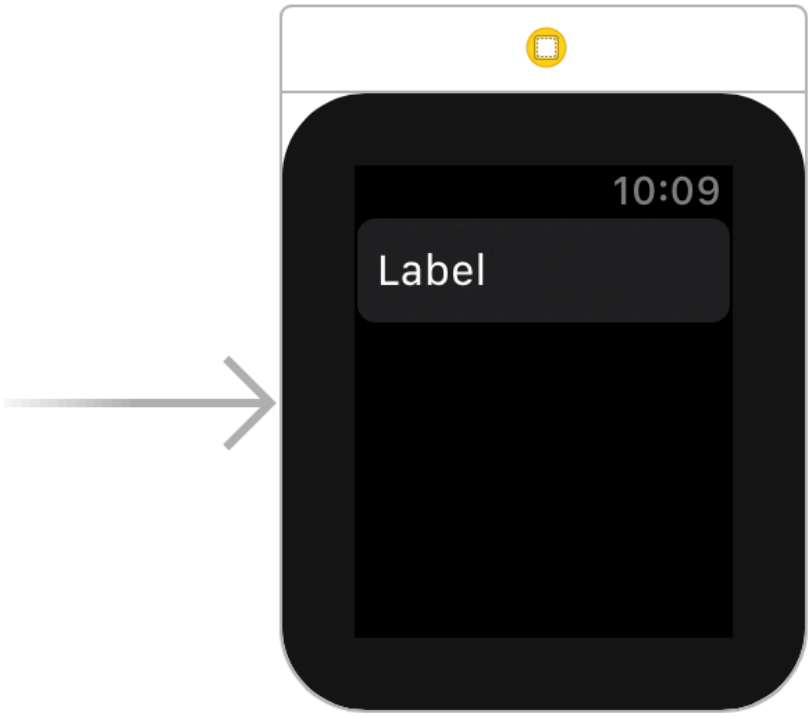
You'll see that working soon enough, but for now let's finish with the table. We're just going to show some text in each row, so we need add a text label to that row. This is done using a special "Label" component from the object library, so find one of them and drag it *into* where it says "Table Row".



Like the table, our new label automatically takes up only the space it needs. If you look carefully, the table row has a dark gray background color against the main screen's black color. If you look even *more* carefully, you'll see that the label is aligned to the top left of the row, which doesn't look great.

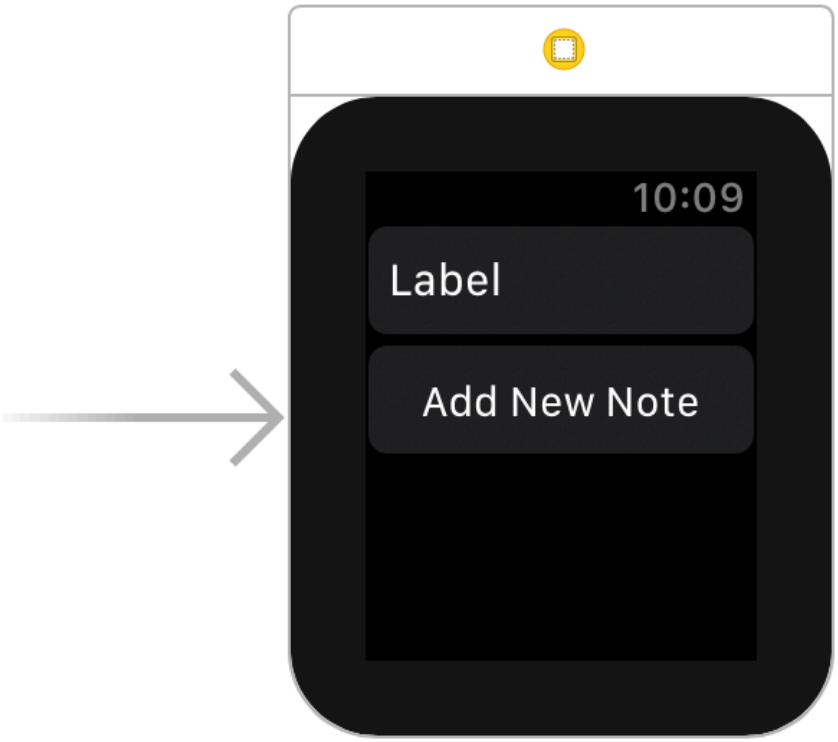
To fix this, we can ask IB to adjust the vertical alignment of our label so that its sits centered inside its container. This won't make the row height change, just make the label sit more attractively inside it. To do this, select the label then press Alt+Cmd+4 to active the attributes inspector – this is a large set of tools on the right-hand of your Xcode window, above the object library.

The attributes inspector lets you change a number of built-in properties for all of WatchKit's components. In the case of labels, that means things like what text it contains, what color it is, what font is used, and so on. Right now, though, we're interested in the vertical alignment property, so scroll down until you see the "Alignment" header and change Vertical from Top to Center.

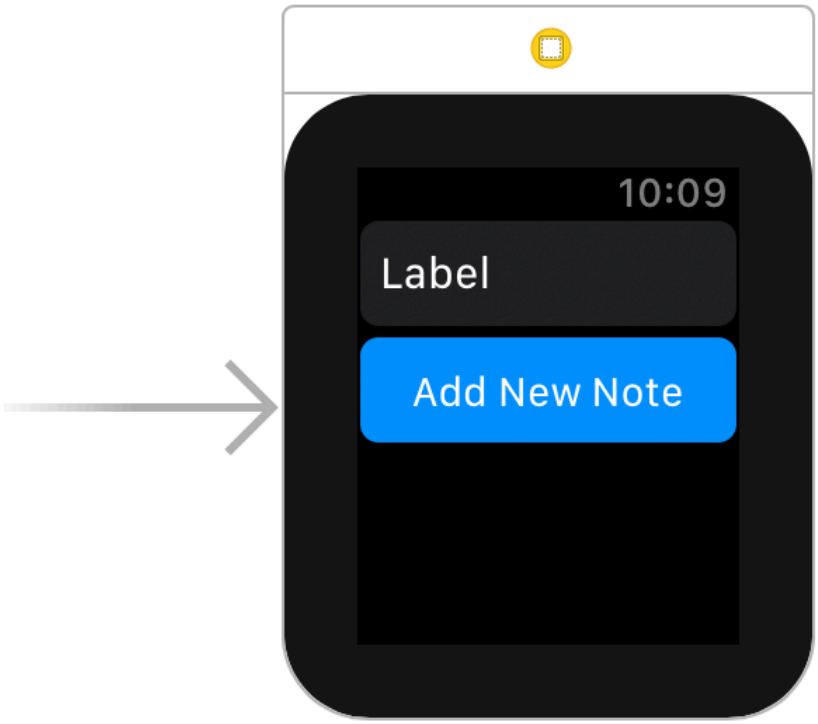


Alongside the table we also need a button to let the user add new notes. This is much easier: WatchKit has a dedicated button component that does everything we need. Find it in the object library, then drop it below the table.

By default, buttons occupy the full width of the screen, have a dark gray background color, and the title “Button”. You should still have the attributes inspector open, so when the button is selected you can adjust its properties.



First, change the Title property from “Button” to “Add New Note”. Now look for the Color property that’s *under* Background property – that’s the one controlling the background color of the button. It should have a white rectangle with a red line through, which means “default color”, or dark gray for buttons. If you click that white rectangle a color picker should appear, so go ahead and choose something a bit more interesting than dark gray – I went for a mid-blue shade often called Dodger Blue.

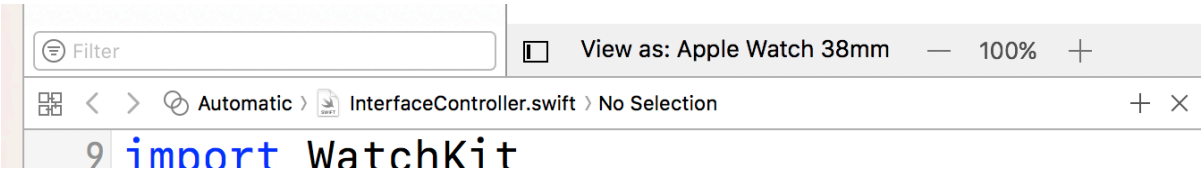


Before we're done we need to tell Interface Builder how the various parts of our user interface relate to the code we're going to write. For example, we're going to need to refer in code to that table we just made, and we also need to tell Xcode what should happen when our "Add New Note" button is clicked.

In IB this is done in quite a peculiar way, but don't worry: I'll walk you through it step by step. With your storyboard still selected, I'd like you to go to the View menu and choose Assistant Editor > Show Assistant Editor. If you prefer shortcuts, you can also press Alt+Cmd+Return to activate it, or even click the small overlapping circles icon in the top-right of your Xcode window.

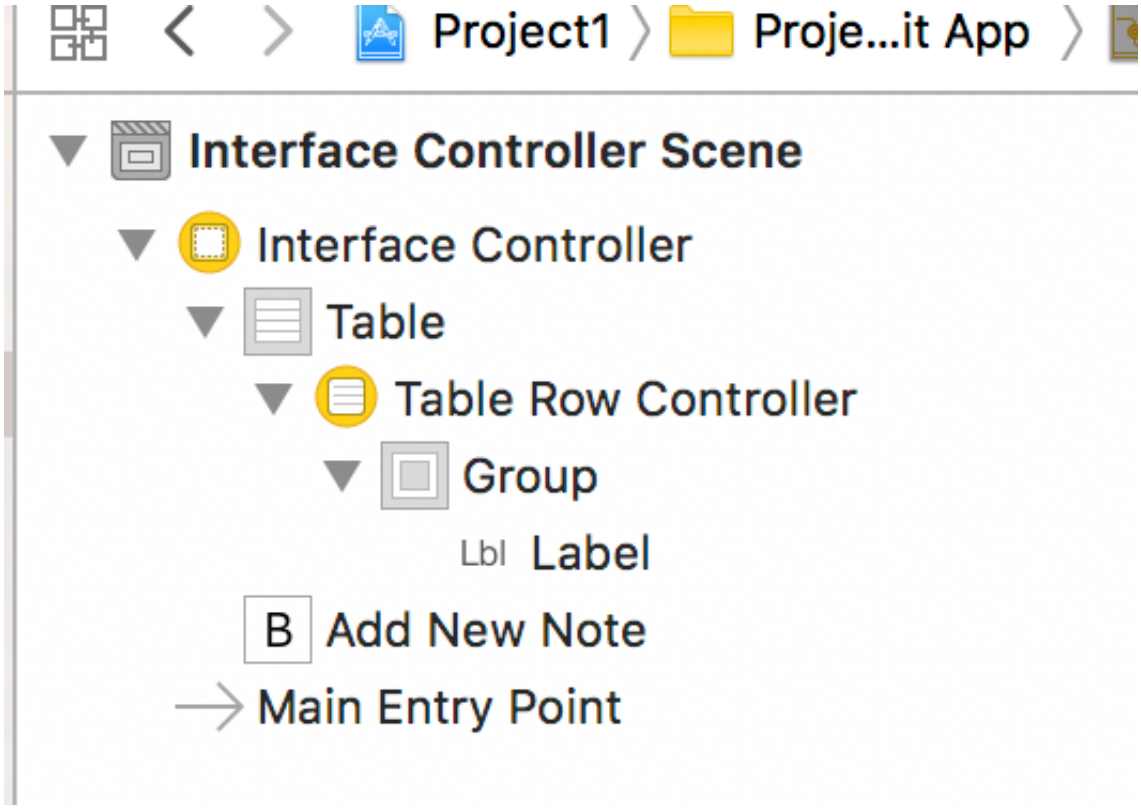
Regardless of how you activate it, the assistant editor splits your Xcode window in two: on the top you'll see the storyboard you were working on, and on the bottom you'll see the code that belongs to that storyboard. Some people like to have their assistant editor split vertically so one window is on the left and the other on the right, but I find the horizontal split easiest – feel free to experiment by going back to the View menu and choosing Assistant Editor > Assistant Editors On Right or Assistant Editors on Bottom.

What you *should* see in the bottom (or right) part of Xcode is the code for InterfaceController.swift. The assistant editor should use "automatic mode" by default, which means it automatically tries to figure out the most useful file to show to match whatever you're viewing in the storyboard. All being well, you'll see something like "Automatic > InterfaceController.swift", as seen in the screenshot below:



If you *don't* see InterfaceController.swift, or if you see some strange variant such as "InterfaceController.swift (Interface)" it means the assistant editor got it wrong and you need to select the file manually instead. To do that, click Automatic and select Manual instead, then choose Project1 > Project1 WatchKit Extension > InterfaceController.swift.

Whether you're using manual mode or automatic, the end result is that you should see your storyboard and the code for InterfaceController.swift at the same. Now for the strange part: inside Interface Builder you should see a pane containing a tree structure of what's in your user interface. If you don't already see it, try clicking on the table on your IB canvas then going to the Editor menu and choosing Show Document Outline. In the event you only see the option "Hide Document Outline" in that menu, it means the document outline is already visible.



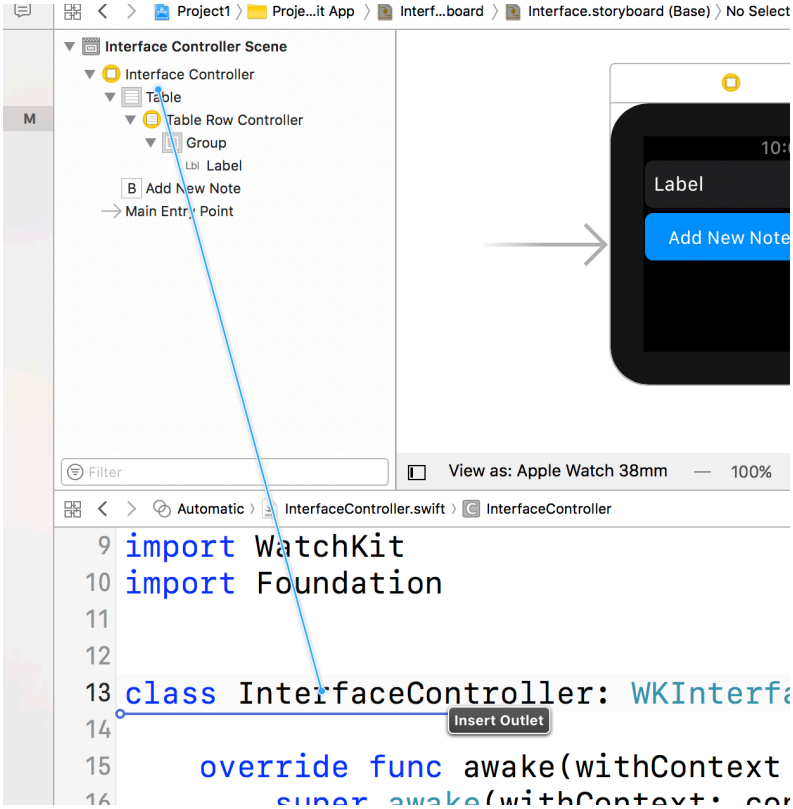
This document outline shows you exactly what's in each screen in your app, all neatly organized so you can see when controls are nested inside other controls. This is useful for the next step: we need to draw a line from controls in our UI right into the code of InterfaceController.swift.

What we're doing is telling Xcode "this piece of UI should match up to this property in my class," so it knows how the two work together. You literally have to drag a line from the UI into your code, but because the UI has lots of nested items – the label is inside a group, which is inside a table row container, etc – it's easy to click the wrong thing by accident.

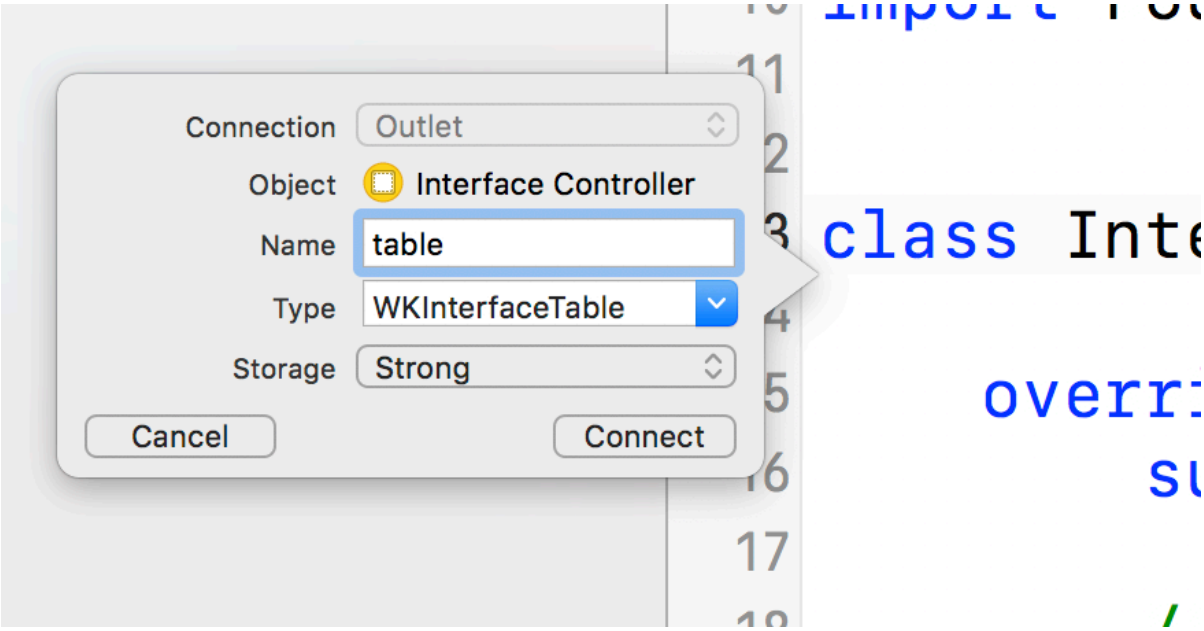
Instead we're going to be drawing lines using the document outline. So, find "Table" in the document outline, and click on it. Now hold down the Ctrl key then click and drag from "Table" down into InterfaceController.swift. A blue line should be drawn from "Table", following your mouse pointer around as you move. This is called a "Ctrl-drag" and is an important skill for IB.

What you need to do is drag just below the line that starts `class InterfaceController`. As you do that, a horizontal blue line should appear, with a

tooltip saying “Insert Outlet”.



Now release your mouse button, and a balloon should appear asking you to fill in some fields. For Name please enter “table”, the Type property should read “WKInterfaceTable” (if it doesn’t, you dragged from the wrong UI element!), and finally make sure Storage is set to “strong”.



When those fields are filled in please click the Connect button, and you'll see a new line of code appear: `@IBOutlet var table: WKInterfaceTable!`. Let's break that down:

- The `@IBOutlet` part means "this is important to IB," but that's all – it has no real meaning in code other than to signify this attribute belongs to IB.
- The `var` part means "this is a variable", which means it stores a value that might change.
- The `table` part is the name of the variable, for us to use in code.
- The `WKInterfaceTable` part is the type of the variable: a table from WatchKit. Most WatchKit things have WK in front of them

Now, the more observant among you will have noticed that the variable type is actually `WKInterfaceTable!` rather than `WKInterfaceTable`. There's a subtle difference but an important one, so let me take a brief detour to explain:

- A `WKInterfaceTable` variable must contain a table object, and can never *not* contain one. So you can't destroy the table and say "sorry, variable, but you're empty now." It must *always* have a table, so it's safe to use.
- A `WKInterfaceTable!` variable – with the added exclamation mark – might hold a table or it might not. It can have an empty value, known as `nil` in Swift.
- Swift has a third variety, written with a question mark like this:

`WKInterfaceTable?`. This might hold a table or it might not.

Now most people are able to understand the difference between "this must definitely always have a value" and "this might have a value sometimes, but other times it might not." The confusion comes with the last two: `!` and `?` both allow variables to have a value or be nil, so why do we have two?

The difference comes in the way you *use* the variables. When a variable is declared using a question mark Swift won't allow you to use it unless you always check that it has a value. Trying to use a variable that doesn't have a value will cause your app to crash, so if you use a `WKInterfaceTable?` for example Swift will always force you to check it has a value before you use it.

When you use an exclamation mark – `WKInterfaceTable!` – Swift *won't* force you to check it, and instead relies on you to know what you're doing.

Again, trying to use a variable that has an empty (`nil`) value is a programmer error, and your app *will* crash. That leads to another question: why ever use `!` rather than `?` if the latter always has us check to be sure there's a value. And more importantly, why does Interface Builder create its variables using `!` rather than `?`.

The answer is down to pragmatic programming. When your interface is initially created – when watchOS is busy allocating enough RAM for your screen – the table hasn't actually been created yet, because that comes a few milliseconds later. So, a regular `WKInterfaceTable` wouldn't work because our table starts life as being nil.

But as soon as the interface has been loaded and your code is ready to run, the table *has* been created and will *remain* created for as long as your screen exists. If we had to check for a possible `nil` value every time we wanted to use the table - which will definitely be there unless something has gone disastrously wrong – it would be frustrating to program.

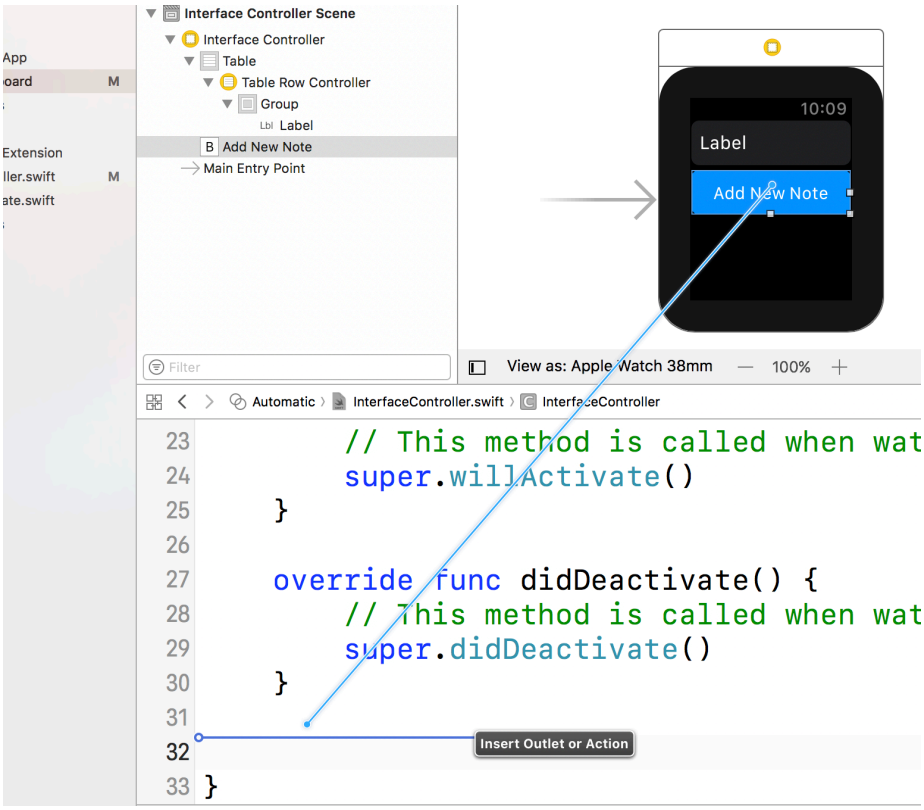
What `!` lets Xcode do is say "this thing will be `nil` very briefly at the very beginning, but by the time you want to use it will definitely have a value." This is perfect here because our table will always exist by the time we need it – and if it doesn't things have gone so wrong that perhaps a crash is the best outcome to avoid data corruption.

If you don't quite understand this just yet, don't worry about it - once you've used it a few times things should become clearer.

We just created an outlet for our table, so let's move on: we need to add an action to run some code when the button is clicked. To do that you need to Ctrl-drag from the button into your code in InterfaceController.swift. This time I'd like you to drag somewhere specific:

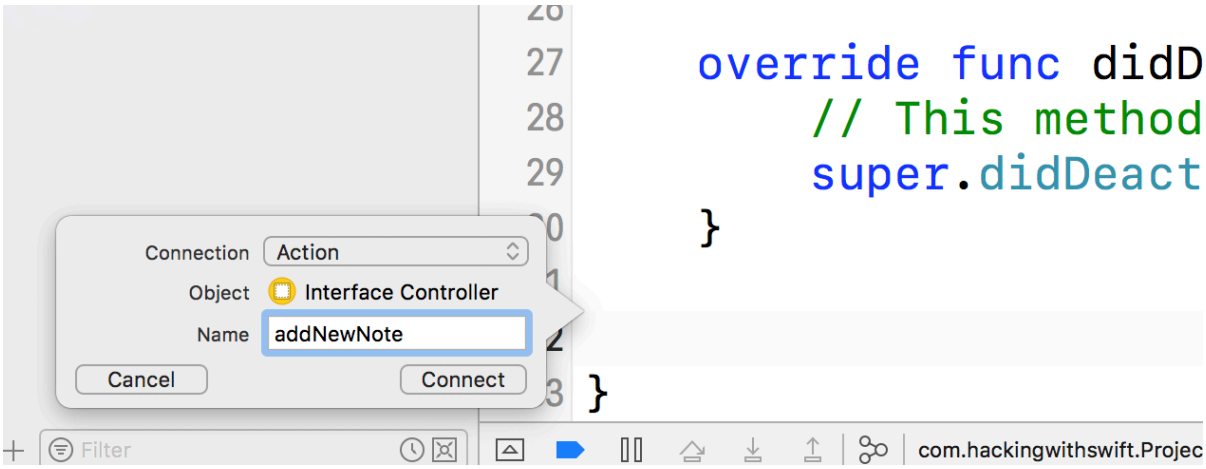
```
    override func didDeactivate() {
        // This method is called when watch view controller is no
longer visible
        super.didDeactivate()
    }

    <-- drag here!
}
```

This time the tooltip will say "Insert outlet or action", like this:



When you release your mouse button, the balloon that appears lets you change the Connection value from Outlet to Action, so please do that. Outlets let you refer to UI components in code, whereas actions are methods in your code that get executed when something happens to the component in question – in this case when the button is clicked.

For the Name value enter "addNewNote", then click Connect.



This will create two lines of code for you:

```
@IBAction func addNewNote() {
}
```

Just like **`@IBOutlet`**, the **`@IBAction`** attribute doesn't have any special meaning in Swift - it's just there to mean "this is something Interface Builder cares about."

What this code does is declare a new method called **`addNewNote()`**. Everything inside the opening and close braces form part of that method – code that will run when the method is called. Right now it's empty, but we'll fill it soon enough.

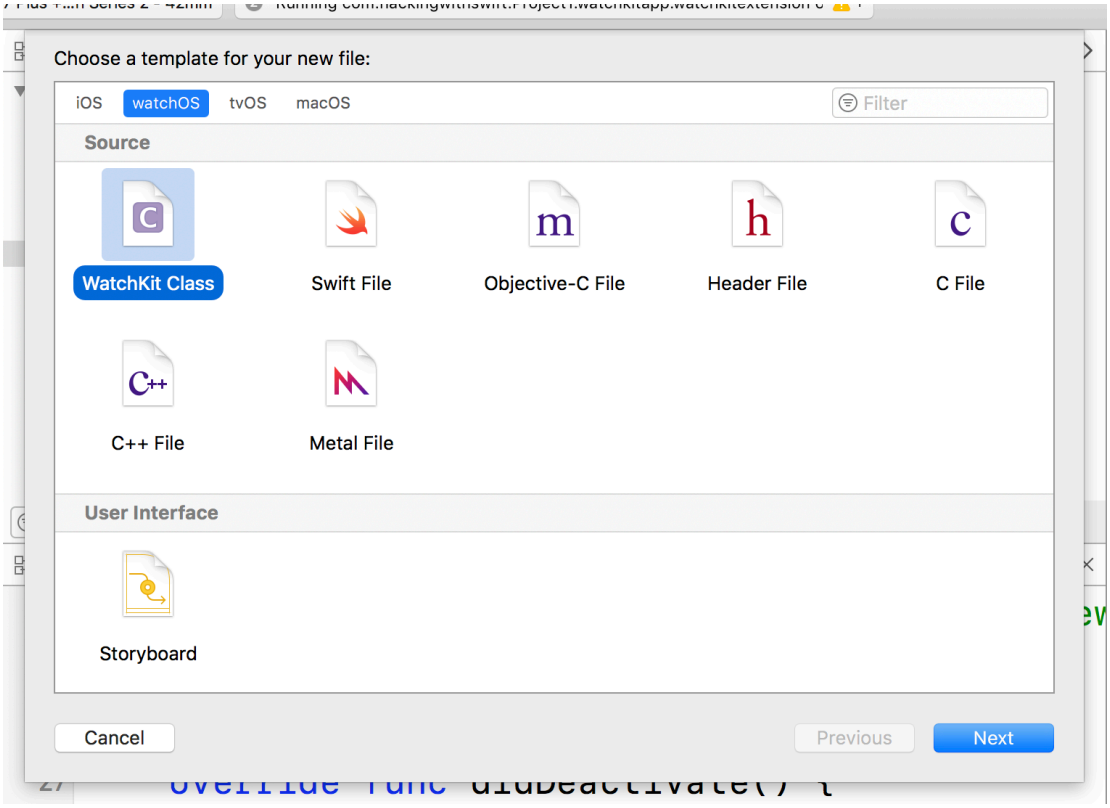
Tip: If you were wondering, the only difference between a function and a method is that methods are part of classes and structs, whereas functions are just loose. Swift uses the same **`func`** keyword for both of them.

We need to do one more thing before we're done, which is to connect our table row label to something so we can refer to it in code. This is a little bit trickier: we can't just connect it straight to InterfaceController.swift, because there could be lots of table rows so which one

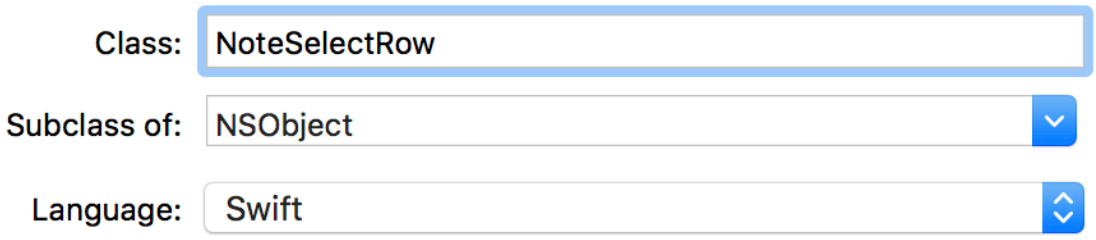
should the code refer to?

The watchOS solution is to create a new data type that refers specifically to one individual row of data. You can then connect the label to that new data type, so each row can refer to its own label.

Let's do it now: go to the File menu and choose New > File, then choose watchOS > WatchKit Class.



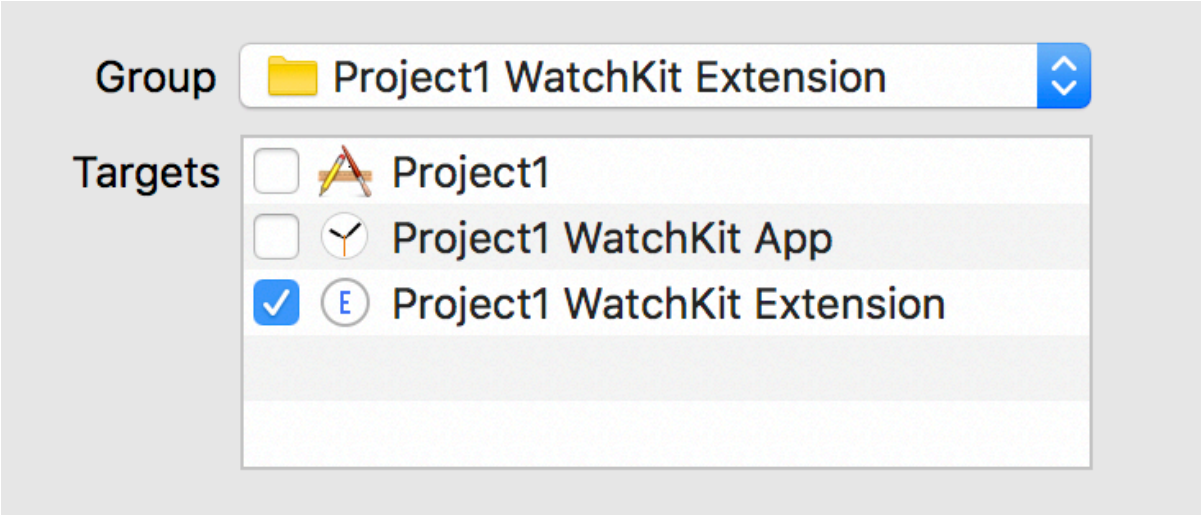
When you click next, you'll be asked to provide a Class and Subclass Of. For Subclass Of please enter NSObject, which is what's called a "universal base class" – the thing that all other data types build on in WatchKit. For Class please enter "NoteSelectRow", which is the name we want to use for this new data type.



When you click Next, you'll see another screen, and this is the part many people screw up. Don't be tempted to click Create without following these instructions carefully!

Warning: Seriously, if you don't read this bit, your code probably won't work.

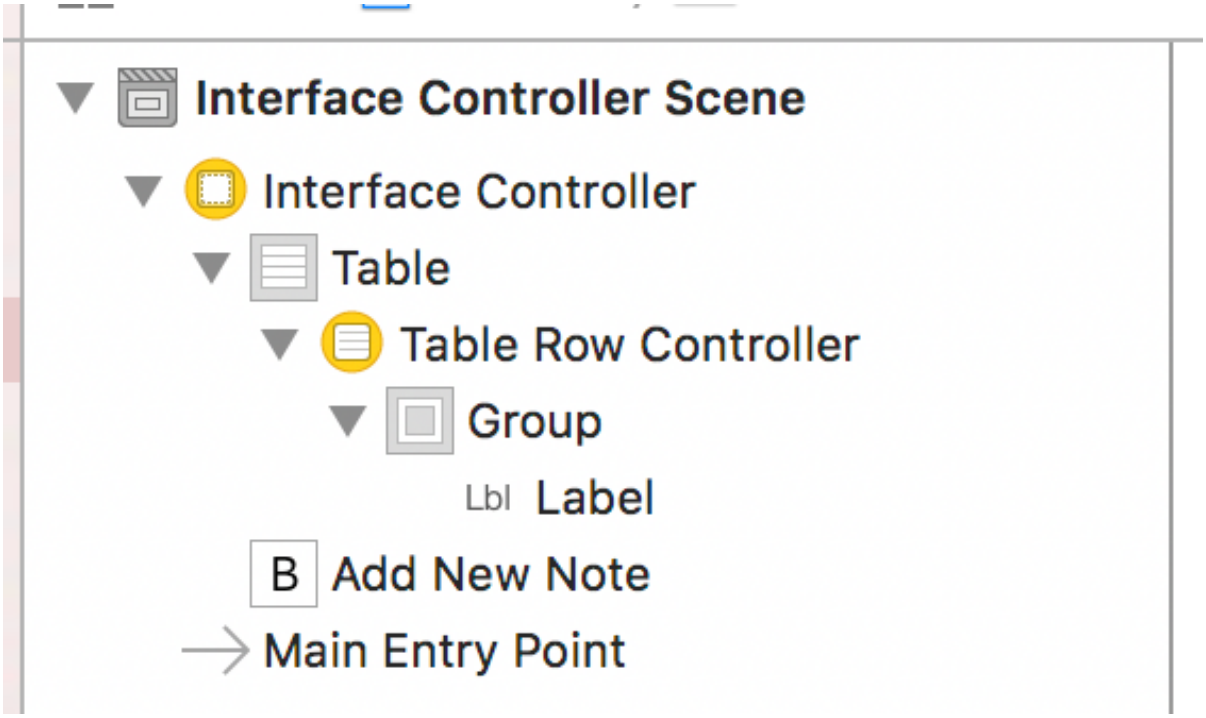
Xcode is asking you both where you want to save the file, and how you want it to be configured for your project. Leave the file selection part alone, and instead focus on the bottom of this window: for Group you should make sure "Project1 WatchKit Extension" is chosen, and in the Targets box make sure "Project1 WatchKit Extension" is the only checkbox that is selected.



When you've done that, click Create. This will create a new file called NoteSelectRow.swift and open it for editing immediately, but I'd like you to re-select Interface.storyboard because we want to hook that up to our UI.

Using the document outline (the tree layout to the left of your IB canvas), look for "Table". You should see it has a small disclosure arrow to its left, so hold down the Alt key and click it

– that will open it, then open all its children, grand-children, and so on. You can open them all by hand if you like, but you do need them all open.



What this tells us is that the interface has a table, which has a table row controller, which has a group, which has a label. The table row controller is what we care about for now, because that's what we need to connect to the `NoteSelectRow` data type we just made.

This isn't done by Ctrl-dragging. Instead, I'd like you to select Table Row Controller in the document outline, then Alt+Cmd+3 to activate the identity inspector. You've already seen the attributes inspector, which is where you answer the question "what should this look like?" The identity inspector is where you answer the question "how should the code for this work?"

For our table row controller we need to change its Class value from an empty box (well, it has "NSObject" written in very faint gray) to be "NoteSelectRow". When you do that, the Module value will automatically be filled with some other text, but that's OK - that's Xcode being helpful.

As well as telling IB what code sits behind our row, we also need to give this it a name. This allows us to write code later on saying "please give me ten rows of the type XXX", and watchOS understands that means creating ten of these rows with labels. So, press Alt+Cmd+4