HACKING WITH SWIFT

SwiftUI Bootcamp

Paul Hudson

SwiftUI Bootcamp

Paul Hudson

Version: 20190902

Contents

Introduction 5
- Welcome!
- Today's plan
- Quick-start project
- Opaque return types, property wrappers, and more

Lists 11
- Working with lists

Images 14
- How to draw images using Image views

VStacks 17
- How to create stacks using VStack

Alignments 20
- How to customize stack layouts with alignment

Fonts 23
- How to style text views with fonts, colors, and more

Navigation views 26
- How to embed a view in a navigation view

Detail Views 29
- This page left intentionally blank

Navigation links 31
- How to push a new view onto a NavigationView

Spacers 34

How to force views to one side inside a stack using Spacer

Padding 38
How to control spacing around individual views using padding and spacing

ZStacks 42
How to layer views on top of each other using ZStack

Offsets 45
How to adjust the position of a view using its offset

Environment objects 48
Working with environment objects

Tab views 52
How to embed views in a tab bar using TabView

Ordering 55
This page also left intentionally blank

Deleting 57
How to let users delete rows from a list

Editing 60
How to enable editing on a list using EditButton

Favoriting 63
How to add bar items to a navigation view

Check out 66
How to show an alert

Forms 69
Basic form design

Introduction

The least you need to know.

Welcome!

Welcome to the SwiftUI Bootcamp! This workshop is designed to be the fastest way to help you get into SwiftUI development, and will cover a huge range of fundamentals in a very short space of time.

In order to make the most of this workshop, please remember the following rules:

1. It's called a workshop because it involves you doing work – you need to open Xcode and get busy.
2. My only job today is to answer your questions. If you hit problems, if you're not sure how something works, or if you'd like me to show you some code examples, just ask.
3. Workshops are *always* better when you work with others, so please make sure you find a coding buddy or two. I highly recommend you follow pair programming principles and don't let one person do all the typing – switch every 20 minutes or so.

Before we dive in, there's one word of warning: SwiftUI leverages all sorts of advanced Swift functionality, and as a result it's common to see obscure and downright unhelpful error messages when you make a small mistake.

If you hit a problem, there's a 50/50 chance Swift's error message is in now way useful – it will say there's an error on line 10 when really the error is on line 5, or similar. The best – well, only! – way to resolve this is using a binary chop style approach just like you might do with Git: keep removing until the error goes away, then narrow down the problem until it's obvious.

With time – and great mental expense – it becomes easier to identify these sorts of issues at a glance, but in the meantime all I want you to remember is this: it's not you being incompetent, it's Swift being wrong.

Today's plan

You're going to build a food ordering app for iPhone. And when I say "you" I mean *you*, not me.

To help make this happen, I have broken down the job into 20 small tasks. Some of these are tricky (looking at you, #13!), but others require only one or two lines of code.

Obviously it would be deeply unfair of me to throw you into the SwiftUI deep end without providing support, so inside this mini-book I've provided edited excerpts from a larger book I wrote called SwiftUI By Example. Each of these excerpts provides you with the information you need to solve the task at hand, so you'll need to read it through, understand it, then try it out yourself in your own project.

Although you shouldn't need it, the original SwiftUI By Example book is available online here: **https://www.hackingwithswift.com/quick-start/swiftui**

Here are the tasks you need to complete in order to build the app:

1. Make a list of menu items (ContentView.swift)
2. Put thumbnails to the left of menu item names (ContentView.swift)
3. Make a **VStack** for the menu item so you can put its price below (ContentView.swift)
4. Modify the **VStack** so the menu item and price are left aligned (ContentView.swift)
5. Format the menu item so it has a headline font (ContentView.swift)
6. Wrap the list in a navigation view with the title "iDine" (ContentView.swift)
7. Press Cmd+N and choose SwiftUI View to create a new view for the detail screen, calling it DetailView.swift. (DetailView.swift)
8. Give the detail view a **MenuItem** property then use a navigation link to push to the detail view from the main content view (DetailView.swift and ContentView.swift)
9. Add to the detail view a **VStack** with menu item image, name, then description (DetailView.swift)
10. Format the detail view: adjust the fonts, and add some padding. (DetailView.swift)
11. Embed the menu item image in a **ZStack** then overlay a photo credit text. Give the credit

text a little padding, with a black background and white text so it stands out better. (DetailView.swift)

12. Align and offset the photo credit text so it sits near the bottom-right corner of the image (DetailView.swift)
13. Use the environment to put the order into ContentView.swift and DetailView.swift, then add a button to the menu item page that adds the item to the current order (SceneDelegate.swift and DetailView.swift)
14. Create a new view that uses **AppView**, create a tab for ContentView with an icon and title, then make the tab view get shown when the app launches (AppView.swift; SceneDelegate.swift)
15. Create a new view called **OrderView** to show a list of items in the current order, wrap it in a navigation view, then add it to the tab view. (OrderView.swift; AppView.swift)
16. Add swipe to delete for the order view (OrderView.swift)
17. Add an edit button to the order view (OrderView.swift)
18. Use the environment to put the user's favorites into ContentView.swift and DetailView.swift, then add a heart icon to the navigation bar for menu items to favorite/unfavorite an item (SceneDelegate.swift; DetailView.swift)
19. Add a Check Out button to the order view that shows a thank you alert when pressed (OrderView.swift)
20. Change the order **VStack** to be a form, then put the order items in a section to split up the table. (OrderView.swift)

In the following pages I list out each task along with excerpts from SwiftUI By Example to help you along.

After each step you'll see a screenshot showing you how the app should look after the step is complete – if yours looks significantly different you might want to double check your code!

Quick-start project

To get you started, I created a new Xcode project using the Single View App template and made some changes:

1. I added all the images for this project to the asset catalog.
2. New file: menu.json, which is a JSON file containing all the menu items for the project.
3. New file: MenuItem.swift, which is a struct containing fields from the JSON plus two helpers for image filenames.
4. New file: Order.swift, which contains an array of menu items the user wants to order.
5. New file: Favorites.swift, which contains an array of items the user likes.
6. Code change: I added **let menu = Bundle.main.decode([MenuItem].self, from: "menu.json")** to ContentView.swift, to load our menu items into the content view.

These things are all standard Swift rather than specific to SwiftUI, which is why I wanted to get them done ahead of time – we have a lot to cram in as it is without having to cover **Codable** and similar!

Note: Make sure you change the bundle ID and signing team to something you control otherwise you'll hit problems.

Opaque return types, property wrappers, and more

SwiftUI leverages lots of Swift features, but four will affect you particularly while using SwiftUI. There isn't time to cover them in detail here, but you do at least need to be aware of them:

1. When you see a return type like **some View** this is an *opaque return type*. It means exactly one view must be returned and it must always be the same type – you can't return zero or two views, and you can't sometimes return some text and other times return an image.
2. When you see a property declaration that starts with an **@** it usually means some special work is happening behind the scenes in order to make a property managed by SwiftUI. For example, **@Environment var…** means that this property comes from the SwiftUI environment.
3. If you a function that has only one expression, you can omit the **return** keyword.

So much of Swift 5.1 centered around shipping features required to make SwiftUI work, and some of these are really advanced features – don't be worried if you haven't seen them previously!

Chapter 1

Lists

Make a list of menu items (ContentView.swift)

Working with lists

SwiftUI's **List** view is similar to **UITableView** in that it can show static or dynamic table view cells based on your needs. However, it is significantly simpler to use: we don't need to create prototype cells in storyboards, or register them in code; we don't need to tell it how many rows there are; we don't need to dequeue and configure cells by hand, and more.

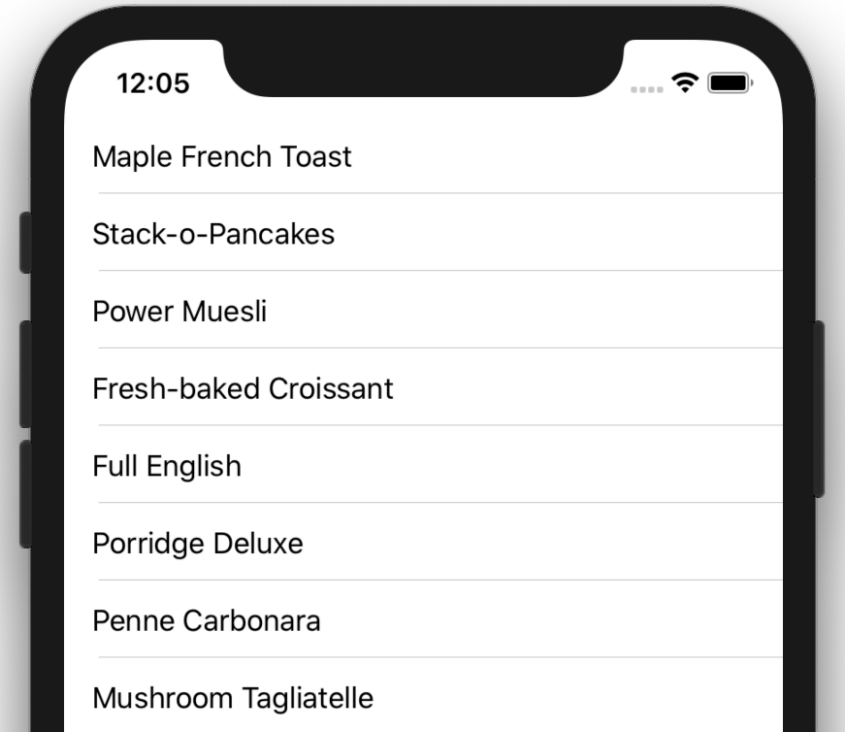
In order to handle dynamic items, you must first tell SwiftUI how it can identify which item is which. This is done using Swift's **Identifiable** protocol, which was introduced in Swift 5.1. This has only one requirement: some sort of **id** value that SwiftUI can use to see which item is which.

In our project, we already have a **MenuItem** struct that conforms to **Identifiable** by using a UUID for its **id** property. Even better, I already loaded an array of menu items using the **Bundle** extension declared in Bundle-Decodable.swift, which means the **menu** property contains an array of menu items ready to show in a list.

So, your job here is to create a list view that shows the menu. To do this we need a **List** view, giving it an array of identifiable things it can loop over, and a closure to run for each item in the list. In our case, that closure needs to accept one menu item, then put it inside a text field:

```
var body: some View {
   List(menu) { item in
      Text("Hello World")
   }
}
```

Obviously we don't want Hello World in there, but I'm sure you can make the leap yourself!


12:05
Maple French Toast
Stack-o-Pancakes
Power Muesli
Fresh-baked Croissant
Full English
Porridge Deluxe
Penne Carbonara
Mushroom Tagliatelle

Chapter 2

Images

Put thumbnails to the left of menu item names (ContentView.swift)

How to draw images using Image views

Use the **Image** view to render images inside your SwiftUI layouts. These can load images from your bundle, from system icons, from a **UIImage**, and more, but those three will be the most common.

To load an image from your bundle and display it inside an image view, you'd just use this:

```
var body: some View {
    Image("example-image")
}
```

To load icons from Apple's San Francisco Symbol set, use the **Image(systemName:)** initializer, passing in the icon string to load, like this:

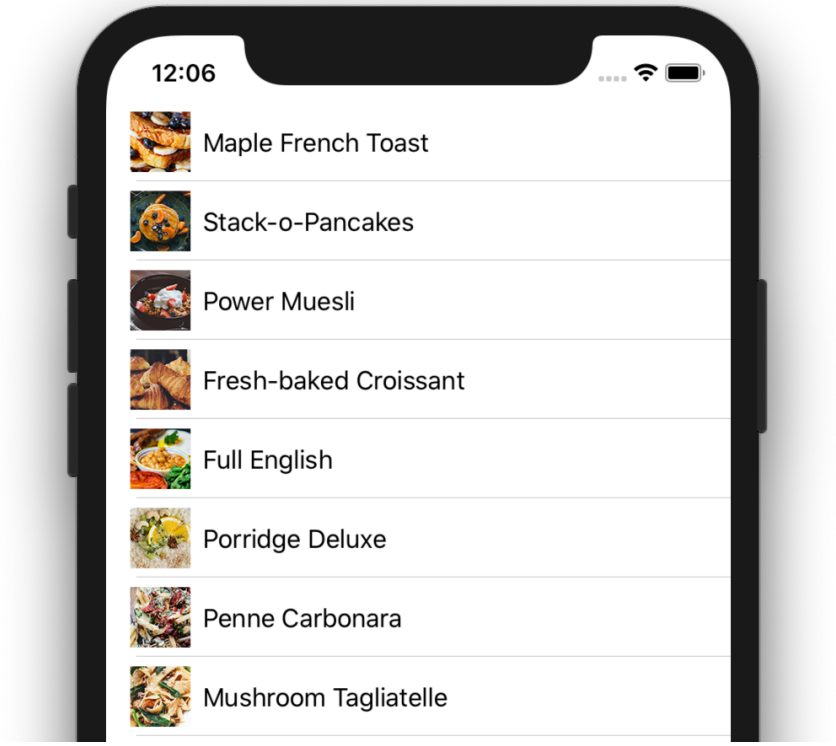
```
Image(systemName: "cloud.heavyrain.fill")
```

When it comes to putting this image inside a list row, there's a problem: where should it go? Well, SwiftUI does a smart thing here by assuming you mean to put them side by side just like you would get with a **UITableViewCell**.

So, if you have a list that provides two views in its closure, they will be placed side by side:

```
List(menu) { item in
    Text("Left")
    Text("Right")
}
```

Now extend that to your project, using the thumbnail image on the left and its name on the right.


12:06
Maple French Toast
Stack-o-Pancakes
Power Muesli
Fresh-baked Croissant
Full English
Porridge Deluxe
Penne Carbonara
Mushroom Tagliatelle

Chapter 3

VStacks

Make a VStack for the menu item so you can put its price below (ContentView.swift)

How to create stacks using VStack

When we put things side by side in a table view cell we used something called *implicit stacking* – SwiftUI looked at the layout we asked for, and decided the only sensible thing to do was place the image and text side by side like we'd get in a **UIStackView**.

As well as implicit stacking, we can also use *explicit* stacking to make things line up vertically or horizontally however we need.

SwiftUI gives us three stack types for this purpose: horizontal (**HStack**), vertical (**VStack**) and depth-based (**ZStack**), with the latter being used when you want to place child views so they overlap.

Let's start with something simple. Here's one text view:

```
Text("SwiftUI")
```

If we want to place another below, we can't just write this:

```
var body: some View {
   Text("SwiftUI")
   Text("rocks")
}
```

Remember, **some View** is an opaque return type, which means we need to return precisely one **View** – that code won't work.

Instead, we need to place it in a vertical stack so our text views are placed above each other:

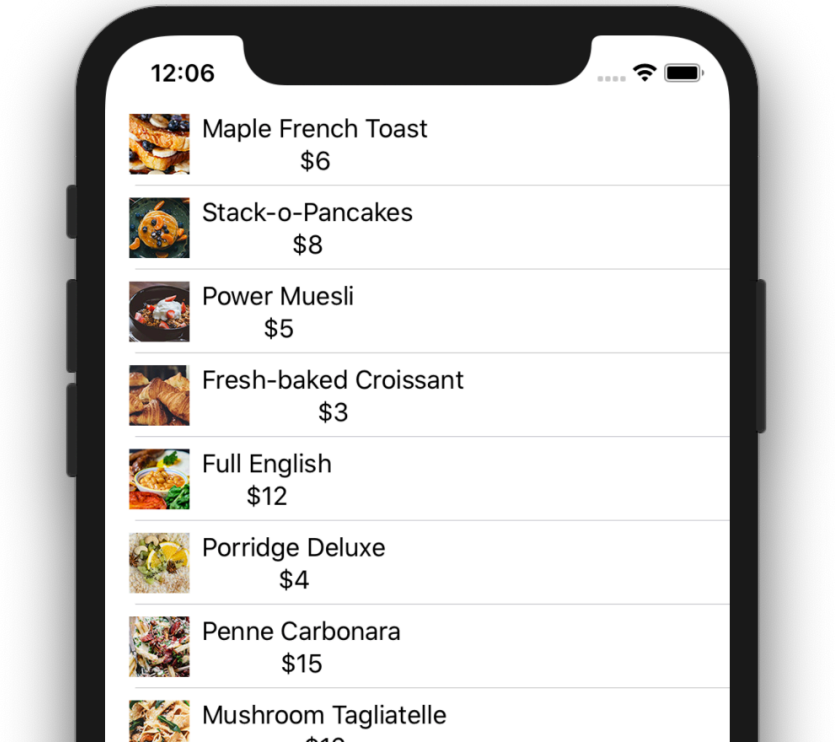
```
VStack {
   Text("SwiftUI")
   Text("rocks")
```

```
}
```

You'll notice that the vertical stack is placed at the center of the screen, with the labels also being centered and having some automatic space between them.

If you wanted the labels side by side horizontally, replace **VStack** with **HStack** like this:

```
HStack {
    Text("SwiftUI")
    Text("rocks")
}
```



Chapter 4
Alignments

Modify the VStack so the menu item and price are left aligned (ContentView.swift)

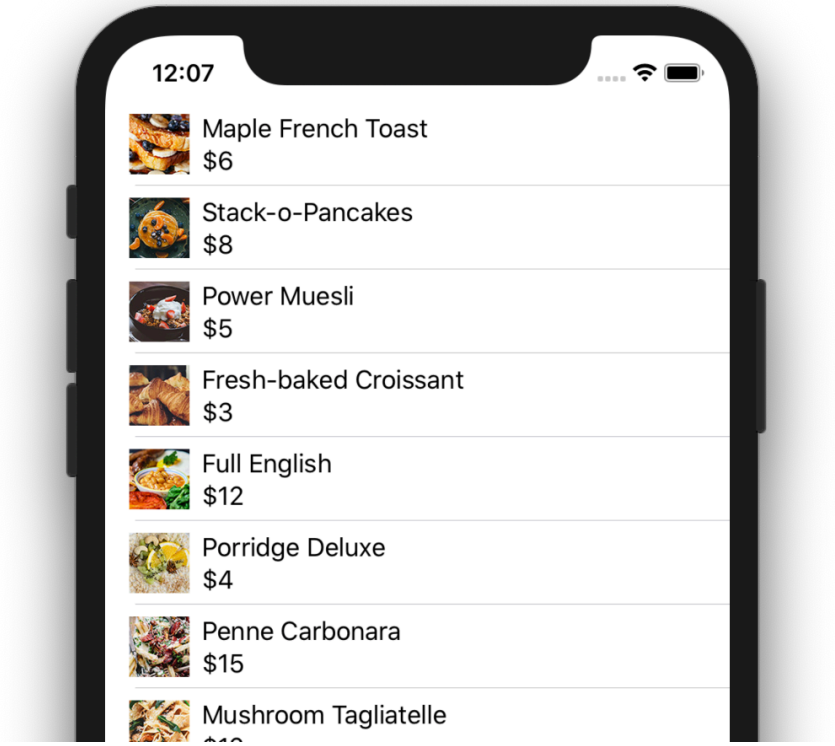
How to customize stack layouts with alignment

By default, items in your stacks are aligned centrally. In the case of **VStack** that means items are aligned to be horizontally in the middle, so if you have two text views of different lengths they would both be aligned to their horizontal center.

To adjust this, pass in an alignment when you create your stack, like this:

```
VStack(alignment: .leading) {
   Text("SwiftUI")
   Text("rocks")
}
```

That will align both “SwiftUI” and “rocks” to their left edge, but they will still ultimately sit in the middle of the screen because the stack takes up only as much space as it needs.


12:07
Maple French Toast
$6
Stack-o-Pancakes
$8
Power Muesli
$5
Fresh-baked Croissant
$3
Full English
$12
Porridge Deluxe
$4
Penne Carbonara
$15
Mushroom Tagliatelle

Chapter 5
Fonts

Format the menu item so it has a headline font (ContentView.swift)

How to style text views with fonts, colors, and more

Not only do text views give us a predictably wide range of control in terms of how they look, they are also designed to work seamlessly alongside core Apple technologies such as Dynamic Type.

By default a Text view has a "Body" Dynamic Type style, but you can select from other sizes and weights by calling **.font()** on it like this:

```
Text("This is an extremely long text string that will never fit
even the widest of Phones")
    .font(.largeTitle)
```

In particular, you will probably want to adjust their text alignment so the lines are centered, like this:

```
Text("This is an extremely long text string that will never fit
even the widest of Phones")
    .font(.largeTitle)
    .multilineTextAlignment(.center)
```

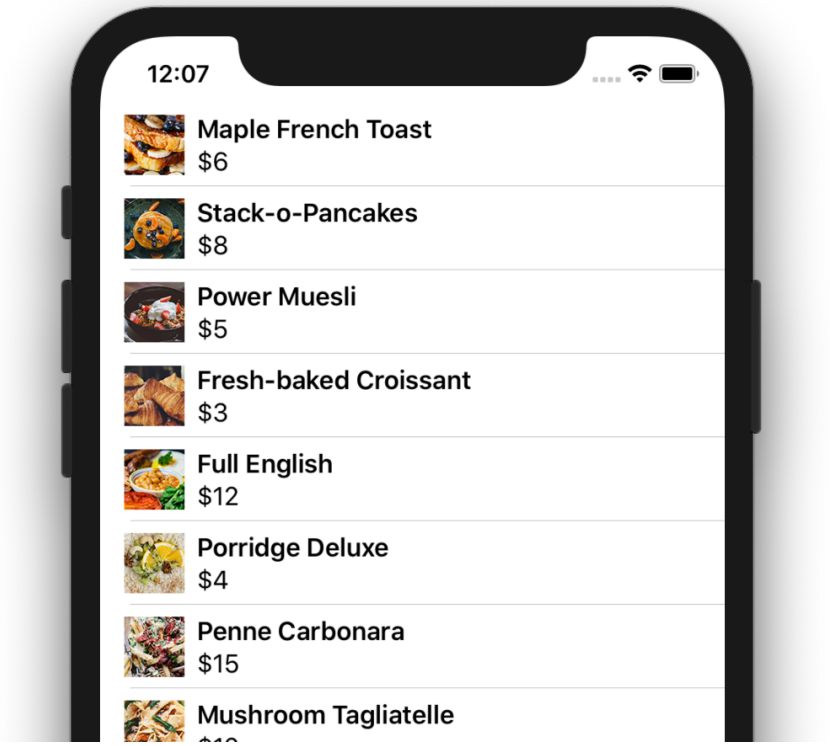
We can control the color of text using the **.foregroundColor()** modifier, like this:

```
Text("The best laid plans")
    .foregroundColor(Color.red)
```

You can also set the background color, but that uses **.background()** because it's possible to use more advanced backgrounds than just a flat color. Anyway, to give our layout a yellow background color we would add this:

```
Text("The best laid plans")
    .background(Color.yellow)
```

```
.foregroundColor(Color.red)
```



Chapter 6

Navigation views

Wrap the list in a navigation view with the title "iDine" (ContentView.swift)

How to embed a view in a navigation view

SwiftUI's **NavigationView** maps more or less to UIKit's **UINavigationController** in that it presents content, it's able to handle navigation between views, and it places a navigation bar at the top of the screen.

In its simplest form you can place a text view into a navigation view like this:

```
NavigationView {
   Text("This is a great app")
}
```

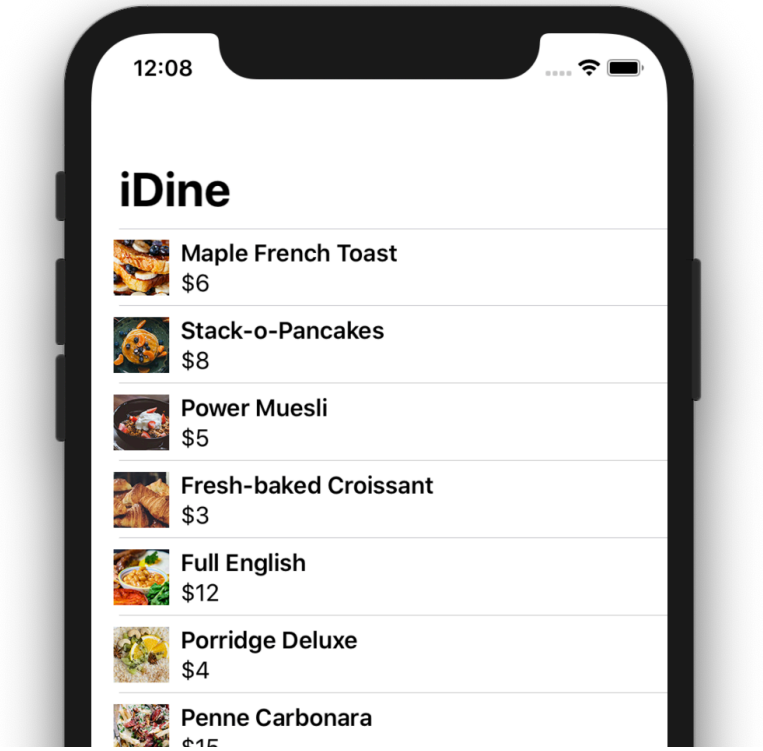
However, that leaves the navigation bar at the top empty. So, you will usually use the **navigationBarTitle()** modifier on whatever you're embedding, so you can add a title at the top of your screen, like this:

```
NavigationView {
   Text("SwiftUI")
      .navigationBarTitle("Welcome")
}
```

The **navigationBarTitle()** modifier gives us some customization options. For example, by default it will inherit large title display mode from whatever view presented it, or if it's the initial view then it will use large titles. But if you'd prefer to force enable or disable large titles you should use the inline parameter like this:

```
.navigationBarTitle("Welcome", displayMode: .inline)
```

That will make small navigation titles, but you can also use **.large** to force a large title.


12:08
iDine
Maple French Toast
$6
Stack-o-Pancakes
$8
Power Muesli
$5
Fresh-baked Croissant
$3
Full English
$12
Porridge Deluxe
$4
Penne Carbonara

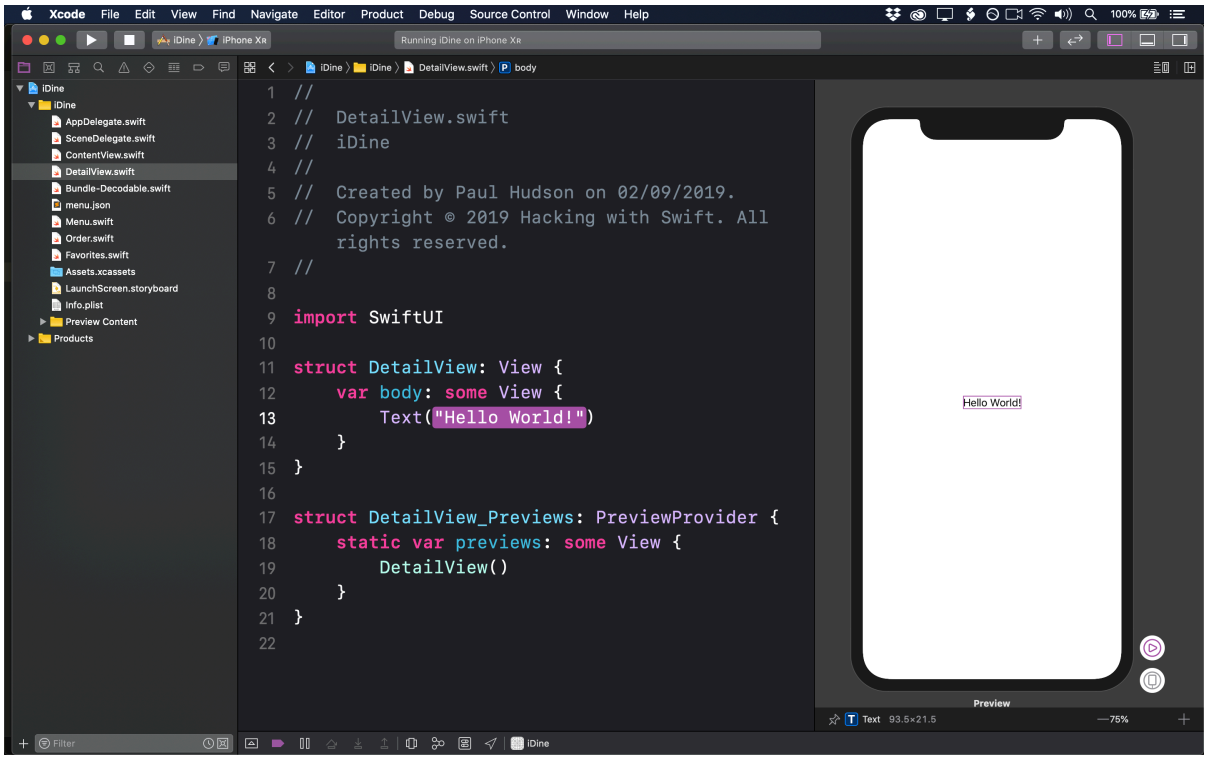
Chapter 7
Detail Views

Press Cmd+N and choose SwiftUI View to create a new view for the detail screen, calling it DetailView.swift. (DetailView.swift)

This page left intentionally blank

You just need to create a new SwiftUI view using Xcode's file templates.

(Yes, you could just put it into ContentView.swift, but then you'll have problems with Xcode's UI previews, so just bite the bullet and make a new file!)



Chapter 8

Navigation links

Give the detail view a MenuItem property then use a navigation link to push to the detail view from the main content view (DetailView.swift and ContentView.swift).

How to push a new view onto a NavigationView

If you have a navigation view and you want to push a new view onto SwiftUI's navigation stack, you should use **NavigationLink**. This take a destination as its first parameter and what to show inside the button as its second parameter (or as a trailing closure), and takes care of pushing the new view on the stack for us along with animation.

For example, if you had a detail view like this:

```
struct DetailView: View {
   var body: some View {
      Text("This is the detail view")
   }
}
```

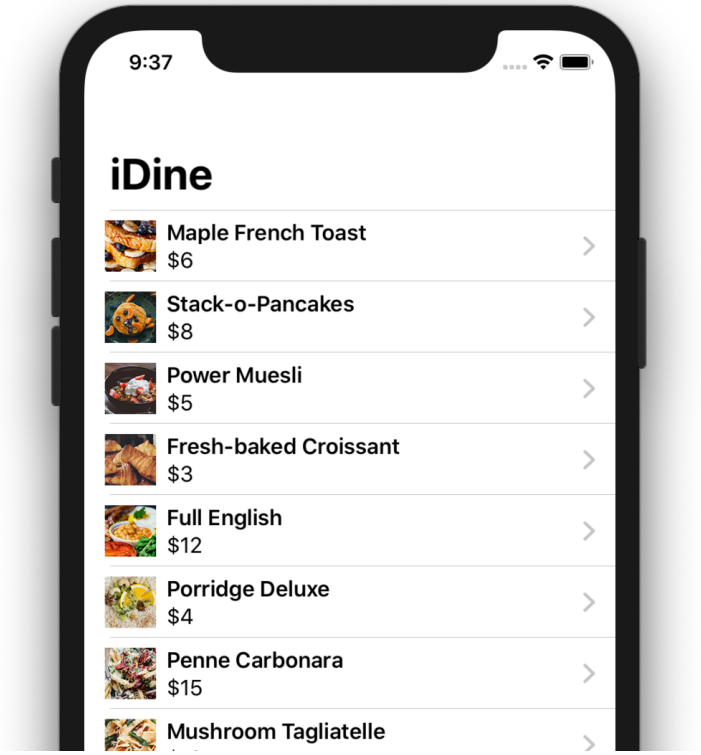
Then you could present it like this:

```
struct ContentView: View {
   var body: some View {
      NavigationView {
         VStack {
            NavigationLink(destination: DetailView()) {
               Text("Show Detail View")
            }.navigationBarTitle("Navigation")
         }
      }
   }
}
```

The destination view can be given any values you want, so if you happened (ahem!) to be inside a list of items you could create your detail view by passing in the current item from your

list.

Tip: Once you add a **MenuItem** property to your detail view Xcode's previews won't work any more because they are created without the property. To get things working again, modify the **DetailView_Previews** struct so that it creates the detail view using **DetailView(menuItem: MenuItem.example)**.



Chapter 9

Spacers

Add to the detail view a VStack with menu item image, name, then description, with a spacer below (DetailView.swift)

How to force views to one side inside a stack using Spacer

Adding a **VStack** with a picture and some text is old news, but forcing the items to the the top is new.

SwiftUI centers its views by default, which means if you place three text views inside a **VStack** all three will sit centered vertically in the screen. If you want to change this – if you want to force views towards the top, bottom, left, or right of the screen – then you should use a **Spacer** view.

For example, this places one text view inside a **VStack**, which means it will be centered vertically:

```
struct ContentView: View {
   var body: some View {
      VStack {
         Text("Hello World")
      }
   }
}
```

To push that text view to the top of the parent, we'd place a spacer below it, like this:

```
VStack {
   Text("Hello World")
   Spacer()
}
```

If you wanted two pieces of text on the leading and trailing edges of a **HStack**, you would use a spacer like this:

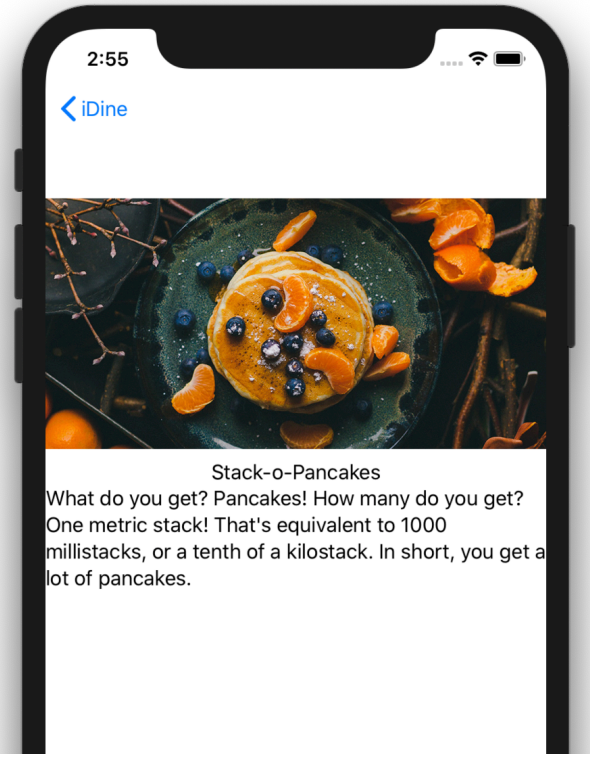
```
HStack {
```

```
    Text("Hello")
    Spacer()
    Text("World")
}
```

Spacers automatically divide up all remaining space, which means if you use if you use several spacers you can divide up the space in varying amounts.

For example, this will place a text view one third of the way down its parent view by putting one spacer before it and two after:

```
VStack {
    Spacer()
    Text("Hello World")
    Spacer()
    Spacer()
}
```


2:55
iDine
Stack-o-Pancakes
What do you get? Pancakes! How many do you get? One metric stack! That's equivalent to 1000 millistacks, or a tenth of a kilostack. In short, you get a lot of pancakes.

Chapter 10

Padding

Format the detail view: Adjust the fonts and add some padding around the description. (DetailView.swift)

How to control spacing around individual views using padding and spacing

SwiftUI lets us set individual padding around views using the **padding()** modifier. If you use this with no parameters you'll get system-default padding on all sides, like this:

```
VStack {
   Text("SwiftUI")
      .padding()
   Text("rocks")
}
```

But you can also customize how much padding to apply and where. So, you might want to apply system padding to only one side:

```
Text("SwiftUI")
   .padding(.bottom)
```

Or you might want to control how much padding is applied to all sides:

```
Text("SwiftUI")
   .padding(100)
```

Or you can combine the two to add a specific amount of padding to one side of the view:

```
Text("SwiftUI")
   .padding(.bottom, 100)
```

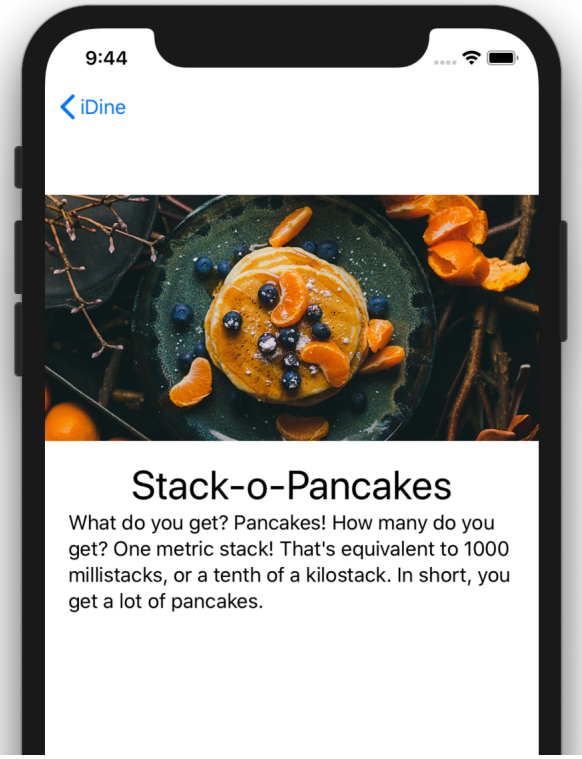
Stacks automatically apply some spacing between their views, but you can control that by providing a value in their initializer, like this:

```
VStack(spacing: 50) {
    Text("SwiftUI")
    Text("rocks")
}
```

You can of course use both alignment and spacing at the same time, like this:

```
VStack(alignment: .leading, spacing: 20) {
    Text("SwiftUI")
    Text("rocks")
}
```

That will align both text views horizontally to the leading edge (that's left for left to right languages), and place 20 points of vertical space between them.



Tip: You might want to use **spacing: 0** for the **VStack** to give you more control over

individual item padding.

Chapter 11

ZStacks

Embed the menu item image in a ZStack then overlay a photo credit text. Give the credit text a little padding, with a black background and white text so it stands out better.
(DetailView.swift)

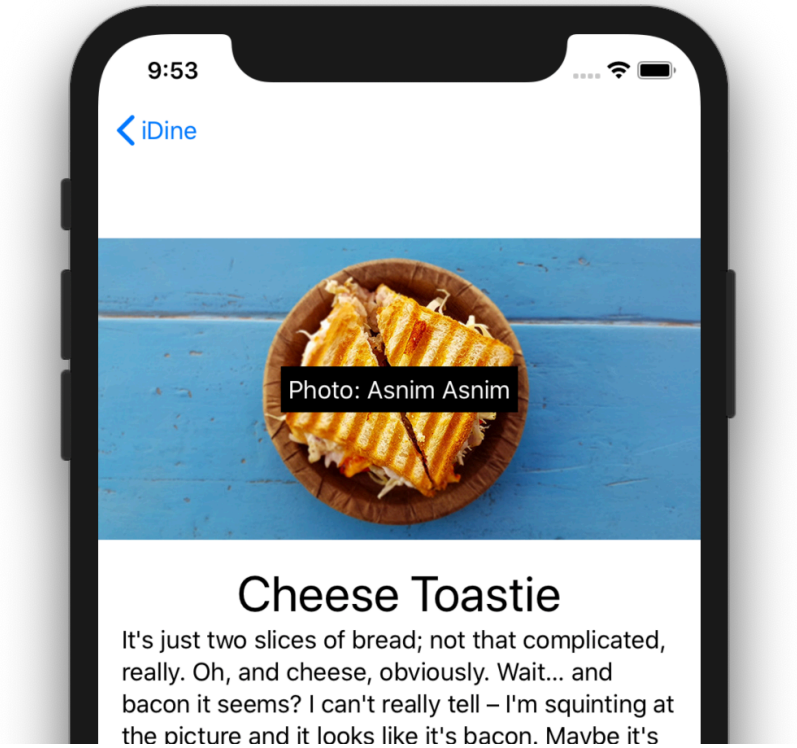
How to layer views on top of each other using ZStack

SwiftUI has a dedicated stack type for creating overlapping content, which is useful if you want to place some text over a picture for example. It's called **ZStack**, and it works identically to the other two stack types.

For example, we could place a large image underneath some text like this:

```
ZStack {
   Image("example-image")
   Text("Hacking with Swift")
}
```

The order items appear in the **ZStack** affect how they are layered. In the above example, the image comes before the text, so the text is drawn on top.


9:53
iDine
Photo: Asnim Asnim
Cheese Toastie
It's just two slices of bread; not that complicated, really. Oh, and cheese, obviously. Wait... and bacon it seems? I can't really tell – I'm squinting at the picture and it looks like it's bacon. Maybe it's

Chapter 12
Offsets

Align and offset the photo credit text so it sits near the bottom-right corner of the image (DetailView.swift)

How to adjust the position of a view using its offset

Like the other stack types, **ZStack** can be created with an alignment so that it doesn't always center things inside itself:

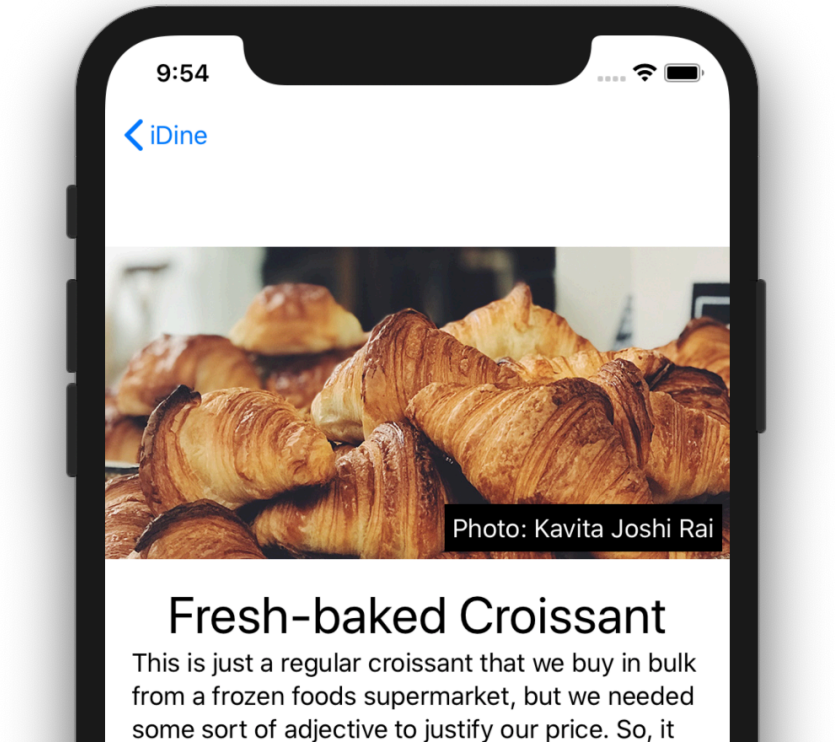
```
ZStack(alignment: .leading) {
    Image("example-image")
    Text("Hacking with Swift")
}
```

However, it doesn't have a *spacing* property because it doesn't make sense. Instead, you should use the **offset()** modifier to move views relative to that natural position once the alignment has been applied. This is particularly useful inside **ZStack**, where it lets you control how views should overlap.

Important: Using **offset()** will cause a view to be moved relative to its natural position, but *won't* affect the position of other views. This means you can make one view overlap another when normally it would have been positioned next to it, which may not be what you want.

For example, in this **VStack** we can use **offset()** to move the second item down by 15 points so that it begins to overlap the third item:

```
VStack {
    Text("Home")
    Text("Options")
        .offset(y: 15)
    Text("Help")
}
```


9:54
iDine
Photo: Kavita Joshi Rai
Fresh-baked Croissant
This is just a regular croissant that we buy in bulk from a frozen foods supermarket, but we needed some sort of adjective to justify our price. So, it

Chapter 13
Environment objects

Use the environment to put the order into ContentView.swift and DetailView.swift, then add a button to the menu item page that adds the item to the current order (SceneDelegate.swift and DetailView.swift)

Working with environment objects

SwiftUI gives us several ways of storing state in our application, but for now you need to understand just one of them: environment objects. These are objects that act as singletons in your application, because we create them once and share them in several views.

We can make any data type work as an environment object by making two changes:

1. Add a conformance to the **ObservableObject** protocol, which tells SwiftUI we want it to monitor this thing for changes.
2. Add the **@Published** property wrapper to any properties we want SwiftUI to watch.

That's it! With those changes, when any published property changes SwiftUI will reload any views that use it, anywhere in our app, thus automatically ensuring our UI stays up to date.

So, if we had a class like this:

```
class User {
   var name = "Taylor Swift"
}
```

We could upgrade it so it could be used as an environment object like this:

```
class User: ObservableObject {
   @Published var name = "Taylor Swift"
}
```

Now that you have something SwiftUI can observe, you can create one instance of it when the app starts then give it to your content view to be used.

If you open SceneDelegate.swift you'll find this line of code inside the **scene(_:willConnectTo:options:)** method:

```
let contentView = ContentView()
```

That creates our initial content view, and is where we need to pass in any environment objects that we've created so that SwiftUI can make them available inside **ContentView** but also any other views it uses.

My quick start project already includes an **order** property, so we just need to pass it into **ContentView** by changing the above line to this:

```
let contentView = ContentView().environmentObject(order)
```

Once that's done, the shared **Order** instance is available to our content view and any other views it hosts or presents. All you need to do is create a property using the **@EnvironmentObject** property wrapper, like this:

```
@EnvironmentObject var order: Order
```

That doesn't need a default value, because it will automatically be read from the environment.

So, once you've injected an object into the environment you can start using it immediately either in your top-level view or ten levels down – it doesn't matter. And most importantly, whenever any view changes the environment, all views relying on it are automatically refreshed so they stay in sync.

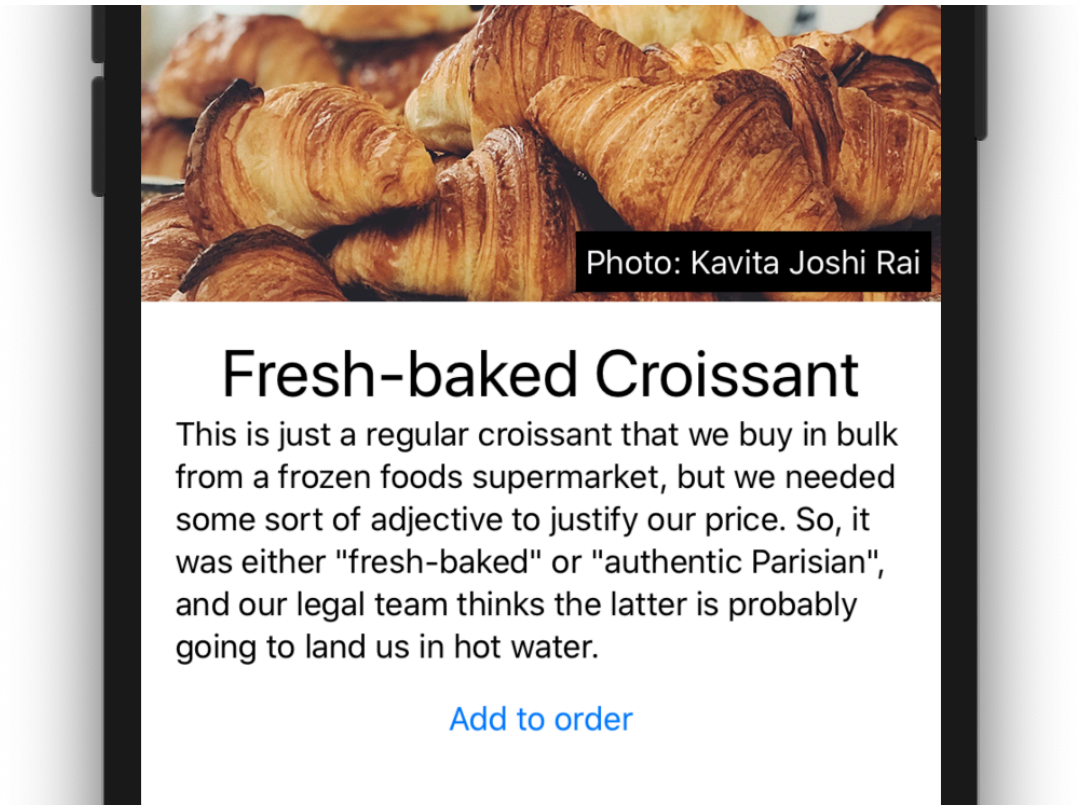
Warning: Now that our views rely on an environment object being present, it's important that you also update your preview code to provide some example settings to use. For example, using something like **ContentView().environmentObject(Order())** for your preview ought to do it.

Now that you have a shared **Order** instance you can use in your detail view, it's time for the real task: you need to add a button to let folks add items to their order.

SwiftUI's button is similar to **UIButton**, except it's more flexible in terms of what content it shows and it uses a closure for its action rather than the old target/action system.

To create a button use code like this:

```
Button(action: {
    // your action here
}) {
    Text("Button title")
}
```



Chapter 14

Tab views

Create a new view that uses a AppView, create a tab for ContentView with an icon and title, then make the tab view get shown when the app launches (AppView.swift; SceneDelegate.swift)

How to embed views in a tab bar using TabView

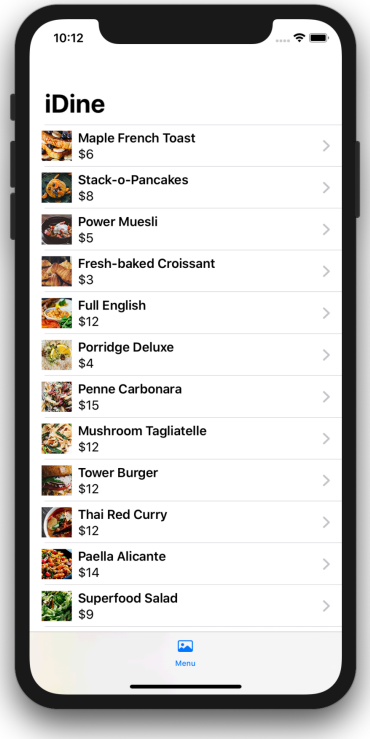
SwiftUI's **TabView** provides an equivalent to **UITabBarController**, allowing us to let the user switch between several active views using a bar at the bottom of the screen.

In its basic form, you should provide each item with an image and title, along with a tag so that SwiftUI knows which view is which. For example, this creates two views with different images, titles, and tags:

```
struct ContentView: View {
    var body: some View {
        TabView {
            Text("First View")
                .tabItem {
                    VStack {
                        Image(systemName: "1.circle")
                        Text("First")
                    }
                }.tag(0)
            Text("Second View")
                .tabItem {
                    VStack {
                        Image(systemName: "2.circle")
                        Text("Second")
                    }
                }.tag(1)
        }
    }
}
```

Even though we've used a **VStack** for the image and title, SwiftUI is smart enough to render

them horizontally as appropriate – on iPhone in landscape, for example.



Chapter 15
Ordering

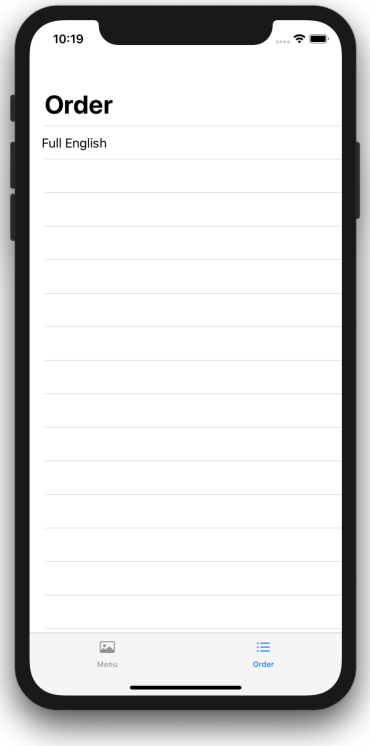
Create a new view called OrderView to show a list of items in the current order, wrap it in a navigation view, then add it to the tab view. (OrderView.swift; AppView.swift)

This page also left intentionally blank

This step isn't trivial, but it doesn't present anything new:

- Create a new SwiftUI view
- Give it an **Order** environment object
- Create a list that loops over items in the order
- Modify AppView.swift to add the new order view as a tab

So, nothing new, but this ought to be good practice!



Chapter 16
Deleting

Add swipe to delete for the order view (OrderView.swift)

How to let users delete rows from a list

SwiftUI makes it easy to let users swipe to delete rows by attaching an **onDelete(perform:)** handler to some or all of your data. This handler needs to have a specific signature that accepts multiples indexes to delete, like this:

```
func delete(at offsets: IndexSet) {
```

Inside there you will usually want to call Swift's **remove(atOffsets:)** method to delete the requested rows from your sequence. Because SwiftUI is watching your state, any changes you make will automatically be reflected in your UI.

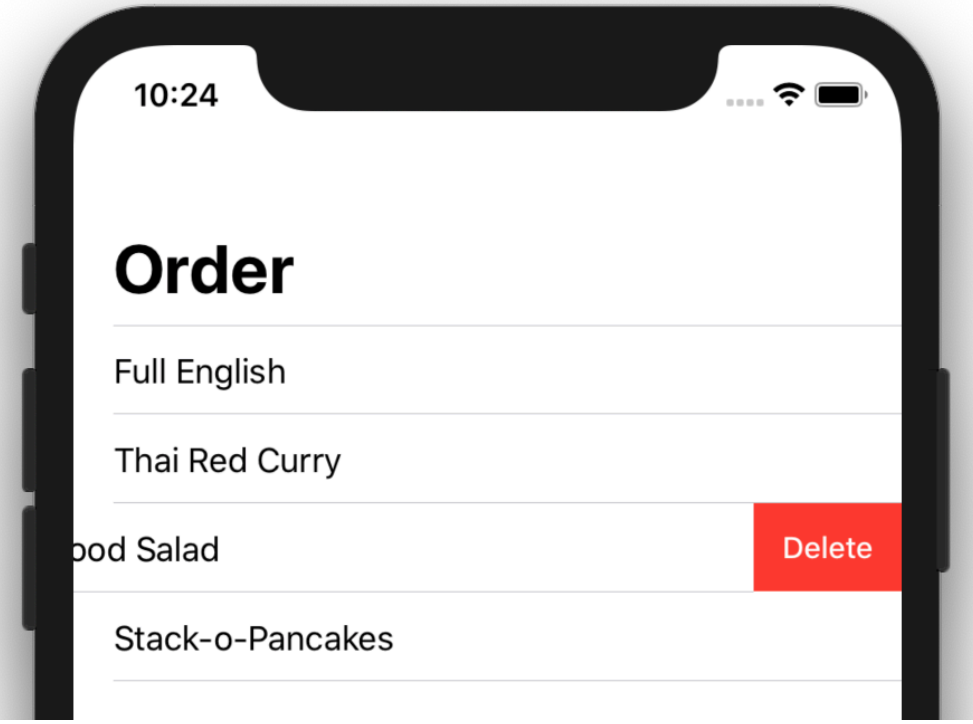
However, there is one catch: SwiftUI doesn't let us directly delete items from a list. Instead, we need to use a special struct called **ForEach** inside the list, which can then have an **onDelete** modifier attached, like this:

```
List {
   ForEach(users, id: \.self) { user in
      Text(user)
   }
   .onDelete(perform: delete)
}
```

Although it isn't needed just yet, this gives the helpful advantage that we can now specify exactly which rows we want to be deletable rather than applying that to the whole list.

In that code **delete** is a method that calls down to Swift's **remove(atOffsets:)** method, like this:

```
func delete(at offsets: IndexSet) {
   users.remove(atOffsets: offsets)
}
```


10:24
Order
Full English
Thai Red Curry
ood Salad
Delete
Stack-o-Pancakes

Chapter 17

Editing

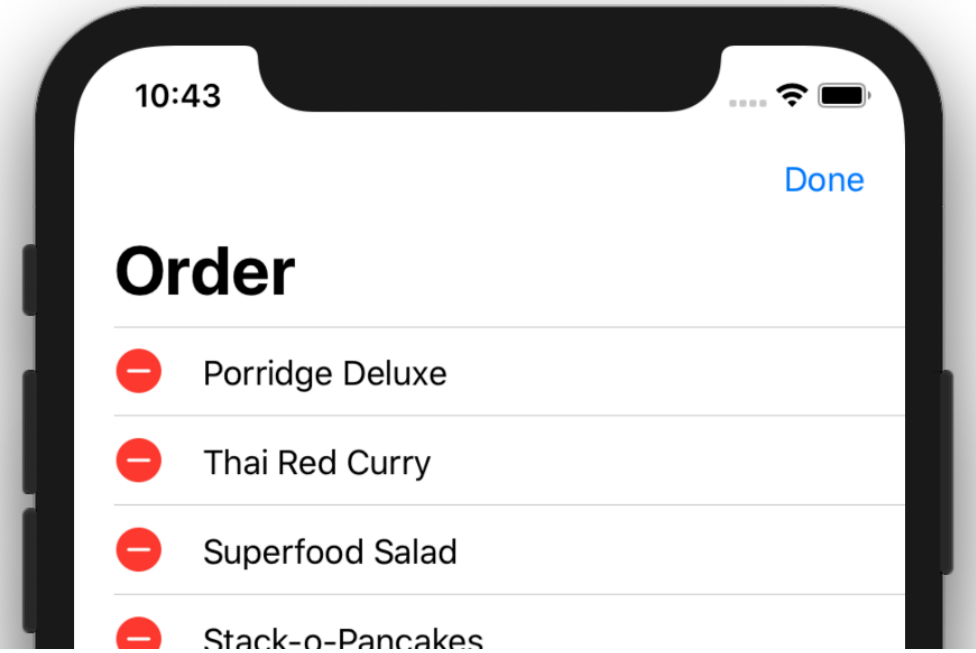
Add an edit button to the order view (OrderView.swift)

How to enable editing on a list using EditButton

If you have configured a SwiftUI list view to support deletion or editing of its items, you can allow the user to toggle editing mode for your list view by adding an **EditButton** to your navigation bar.

For example:

```
var body: some View {
   NavigationView {
      List {
         ForEach(users) { user in
            Text(user.name)
         }
         .onDelete(perform: delete)
      }
      .navigationBarItems(trailing: EditButton())
   }
}
```


10:43
Done
Order
Porridge Deluxe
Thai Red Curry
Superfood Salad
Stack-o-Pancakes

Chapter 18
Favoriting

Use the environment to put the user's favorites into ContentView.swift and DetailView.swift, then add a heart icon to the navigation bar for menu items to favorite/unfavorite an item (SceneDelegate.swift; DetailView.swift)

How to add bar items to a navigation view

The **navigationBarItems()** modifier lets us add custom bar button items to the leading and trailing edge of a navigation view. These might be a tappable button, but there are no restrictions – you can add any sort of view.

For example, this adds a Help button the to trailing edge of a navigation view:

```
var body: some View {
   NavigationView {
      Text("SwiftUI")
         .navigationBarTitle("Welcome")
         .navigationBarItems(trailing:
            Button(action: {
               print("Help tapped!")
            }) {
               Text("Help")
            })
   }
}
```

10:44
iDine

Chapter 19

Check out

Add a Check Out button to the order view that shows a thank you alert when pressed (OrderView.swift)

How to show an alert

The code to create a basic SwiftUI alert looks like this:

```
Alert(title: Text("Important message"), message: Text("Wear
sunscreen"), dismissButton: .default(Text("Got it!")))
```

That defines a title and message, like you'd see in a **UIAlertController**, then adds a dismiss button with a default style and the text "Got it!".

To *show* that alert you need to define some sort of condition that determines whether the alert should be visible or not. You then attach that to your main view, which presents the alert as soon as its condition becomes true.

You already met **@ObservableObject**, which is used to make SwiftUI watch an external class for changes. For simpler things like "should the alert be showing or not?" we use a different property wrapper called **@State**, which is used to modify simple values like strings, integers, and Booleans that are local to the view.

For example, this code creates a **showingAlert** Boolean that tracks whether the sunscreen message should be shown or not, sets that Boolean to true when a button is tapped, then creates and attaches an alert view using that Boolean so it appears when the button is tapped:

```
struct ContentView: View {
   @State private var showingAlert = false

   var body: some View {
      Button(action: {
         self.showingAlert = true
      }) {
         Text("Show Alert")
      }
      .alert(isPresented: $showingAlert) {
         Alert(title: Text("Important message"), message:
```

```
Text("Wear sunscreen"), dismissButton: .default(Text("Got
it!")))
        }
    }
}
```

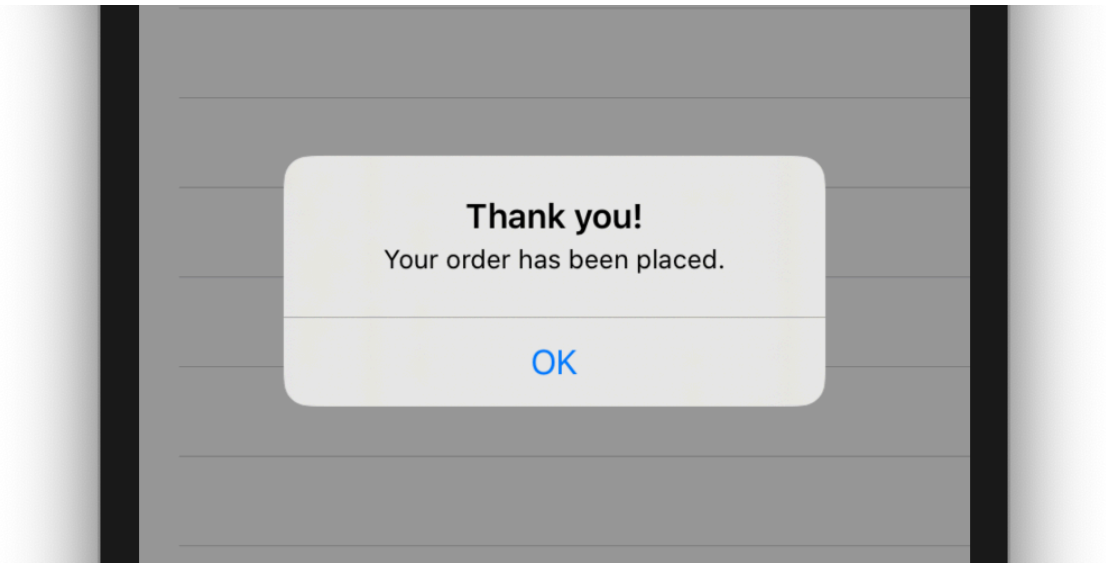
Record scratch.

Freeze frame.

Yes, that uses **$showingAlert** rather than just **showingAlert**, and the difference *matters*.

If you refer to a property normally, it means "read this value". For example, **alert(isPresented: showingAlert)** might mean "when **showingAlert** is true, show the alert." But here we actually want extra functionality: when the alert is dismissed we want **showingAlert** to be set back to false again, so this is a case where we want to both read *and* write the value.

This is what's called a two-way binding, and it's very common in SwiftUI. In this instance, it means presenting an alert like this will automatically set **showingAlert** back to false when the dismiss button is tapped, which means we can show and hide it repeatedly without problems.



Chapter 20

Forms

Change the order VStack to be a form, then put the order items in a section to split up the table. (OrderView.swift)

Basic form design

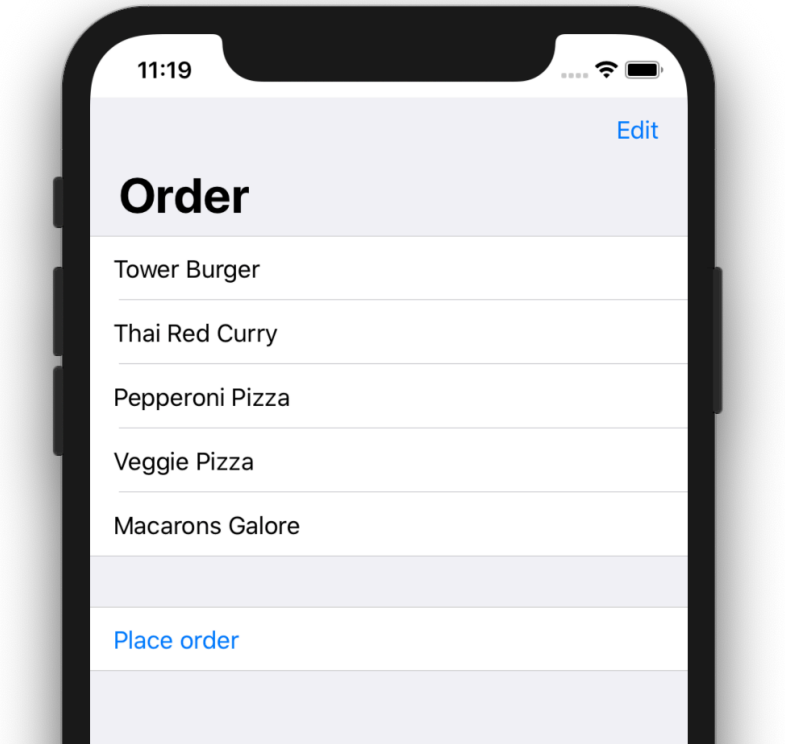
SwiftUI's forms work as containers, just like **VStack**, which means you can add other views inside it as needed. However, they automatically adapt the behavior and styling of some controls so they fit better in the form environment.

For example, this creates a form with a button:

```
NavigationView {
   Form {
      Button(action: {
         // do something important
      }) {
         Text("Save changes")
      }
   }.navigationBarTitle("Settings")
}
```

When that code is run you'll see the button has adapted itself to look like an actionable list row – it's left-aligned and in blue.

SwiftUI's forms often work best when broken into sections, just like regular lists of content. As an example, you would use code like this to create a list of user names with a button at the end:

```
Form {
   List {
      Section {
         ForEach(users) { user in
            Text(user.name)
         }
      }
```

```
        Button(action: {
            print("Hello world!"
        }) {
            Text("Print message")
        }
    }
}
```