Article

Previewing your app's interface in Xcode

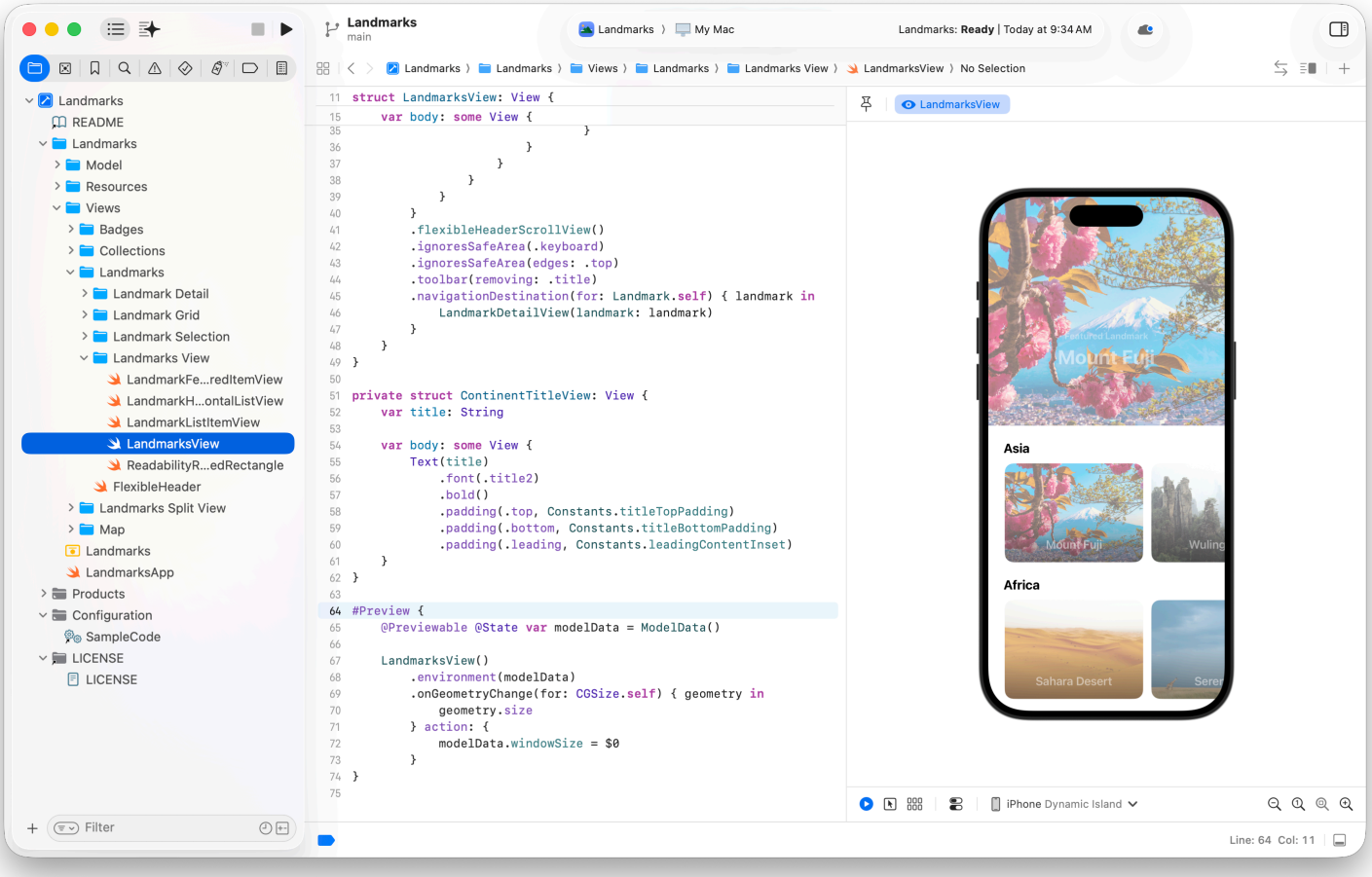
Iterate designs quickly and preview your apps' displays across different Apple devices.

Overview

With Swift previews, you can make changes to your app's views in code, and see the result of those changes quickly in the preview canvas. Add previews to your SwiftUI, UIKit, and AppKit views using the preview macro. Then configure how you want your previews to display using the preview canvas controls, or programmatically in code.

Add a preview macro to your interface file

When you add views to your code, you can display them in the preview canvas. The preview canvas shows how your view appears on different devices in a variety of configurations.



The Swift preview macro is a snippet of code that makes and configures your view. You use one of the preview macros — such as `Preview(_:body:)` — to tell Xcode what to display. To manually show or hide the preview canvas, select Editor > Canvas from the Xcode menu.

SwiftUI UIKit AppKit

```
// A SwiftUI preview.
#Preview {
    // The view to preview.
}
```

To add a preview macro to your view:

1. Open the source file of the view you want to display.
2. Add the `#Preview` macro to the file.
3. Create and return an instance of the view configuration you want to display in the body of the trailing closure of the macro.

SwiftUI UIKit AppKit

```
struct ContentView: View {
    var body: some View {
        // ...
    }
}

// A SwiftUI preview.
#Preview {
    ContentView()
}
```

Generate previews using intelligence

You can use Xcode coding intelligence to generate a preview for you. In the source editor, select some view code and click the coding assistant icon that appears, or Control-click a symbol and choose Show Coding Tools > Show Coding Tools from the pop-up menu. In the coding tools popover that appears, click Generate a Preview.

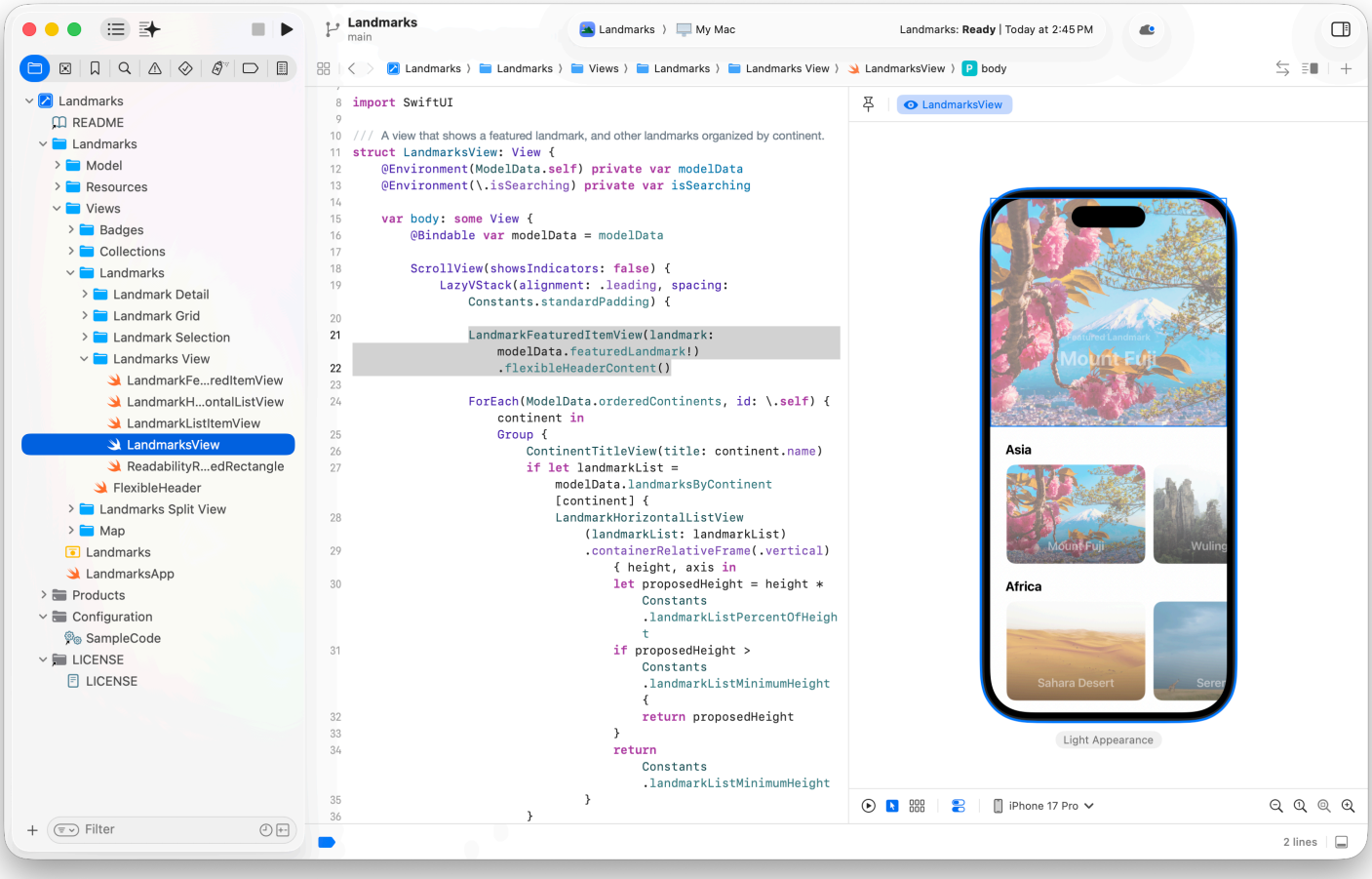
For more information on coding intelligence features, see Writing code with intelligence in Xcode.

Interact with your view in live mode

When you select the live or interactive preview option, your view appears and interacts just like it would on a device or simulator. Use live mode to test control logic, animations, and text entry as well as responses to asynchronous code. When you click the Live button at the bottom of the preview canvas, a single device preview appears in the canvas that you can interact with. This is the default mode for new previews that you add to your files.

Try out new designs quickly with selectable mode

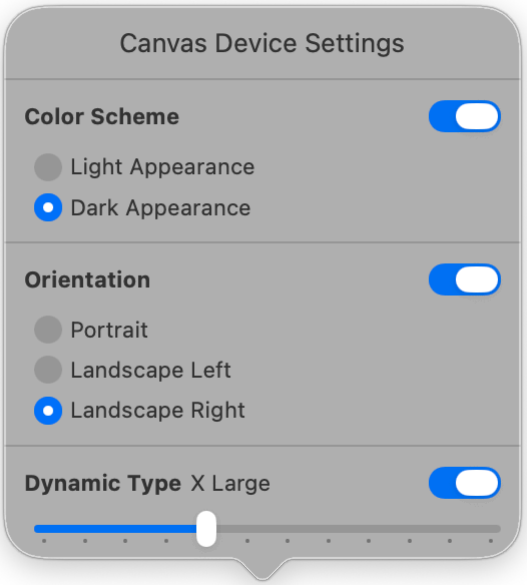
In selectable mode, the preview displays a snapshot of your view so you can interact with your view's UI elements in the canvas. To highlight the code for an element in the source editor, click the Selectable button at the bottom of the preview canvas, and double-click the element in the preview canvas. Xcode highlights both the element in the preview canvas and the corresponding code in the source editor. Then you can make code changes to the element in the source editor and see the results immediately in the preview canvas.



Control how your previews display with device settings

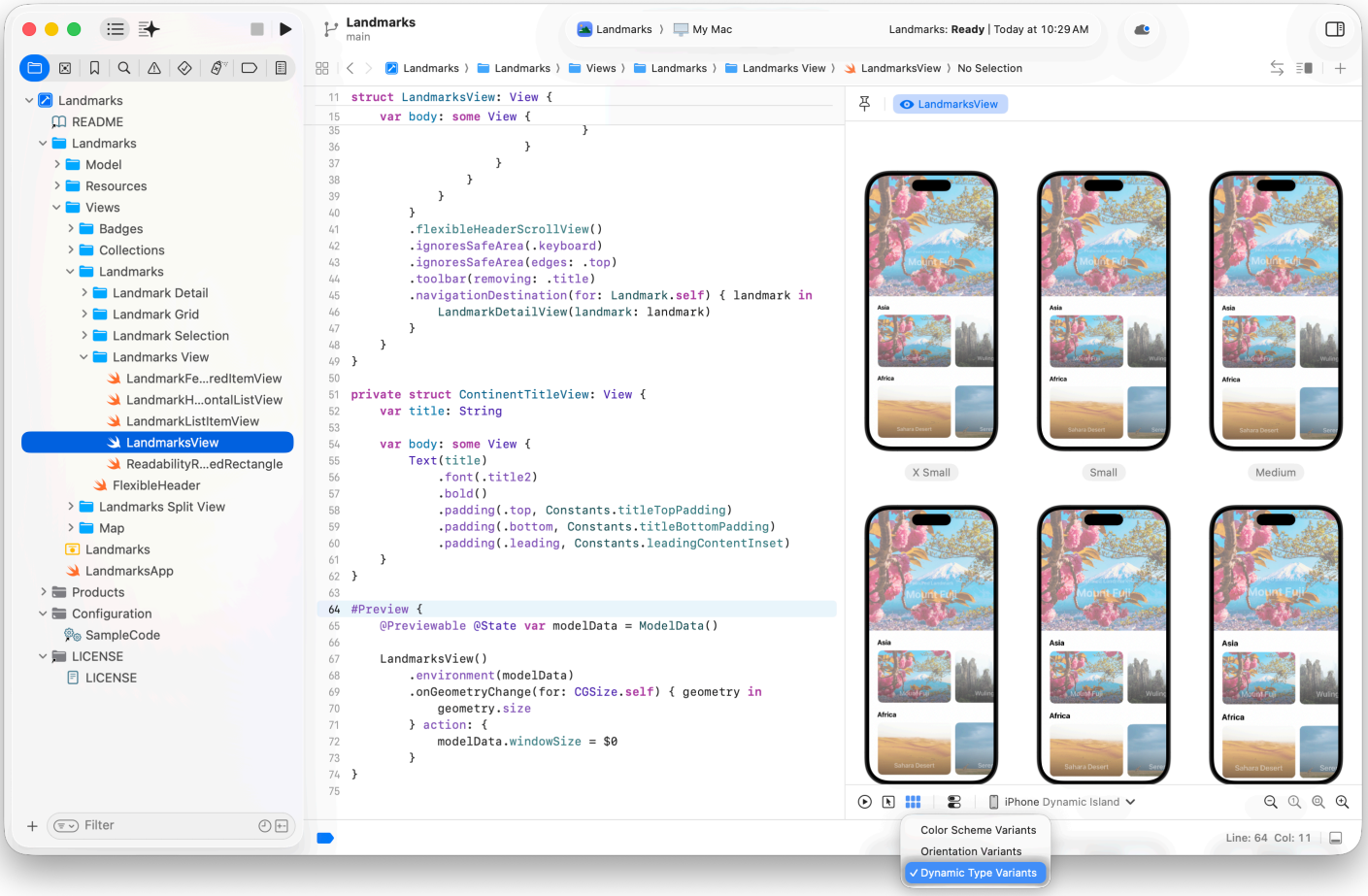
Use Device settings to control how a preview displays for a specific device. For example, to see how your view looks in the dark appearance, in a landscape right orientation, with extra large text:

1. Click Device Settings at the bottom of the preview canvas.
2. Toggle Color Scheme on, and select Dark Appearance under Color Scheme.
3. Toggle Orientation on, and select Landscape Right under Orientation.
4. Toggle Dynamic Type on, and move the Dynamic Type slider to the X Large text setting.



Test different view configurations

Use variant mode to see how your view appears in different variations for a given configuration. For example, to test how well your view supports accessibility, select Variant mode from the bottom of the preview canvas, and select the Dynamic Type Variants option. Xcode displays your view with different sizes of text.



Preview canvas supports the following variations:

Color Scheme Variants
: Displays a light and dark preview of your view.

Orientation Variants
: Displays your view in all the different portrait and landscape orientations.

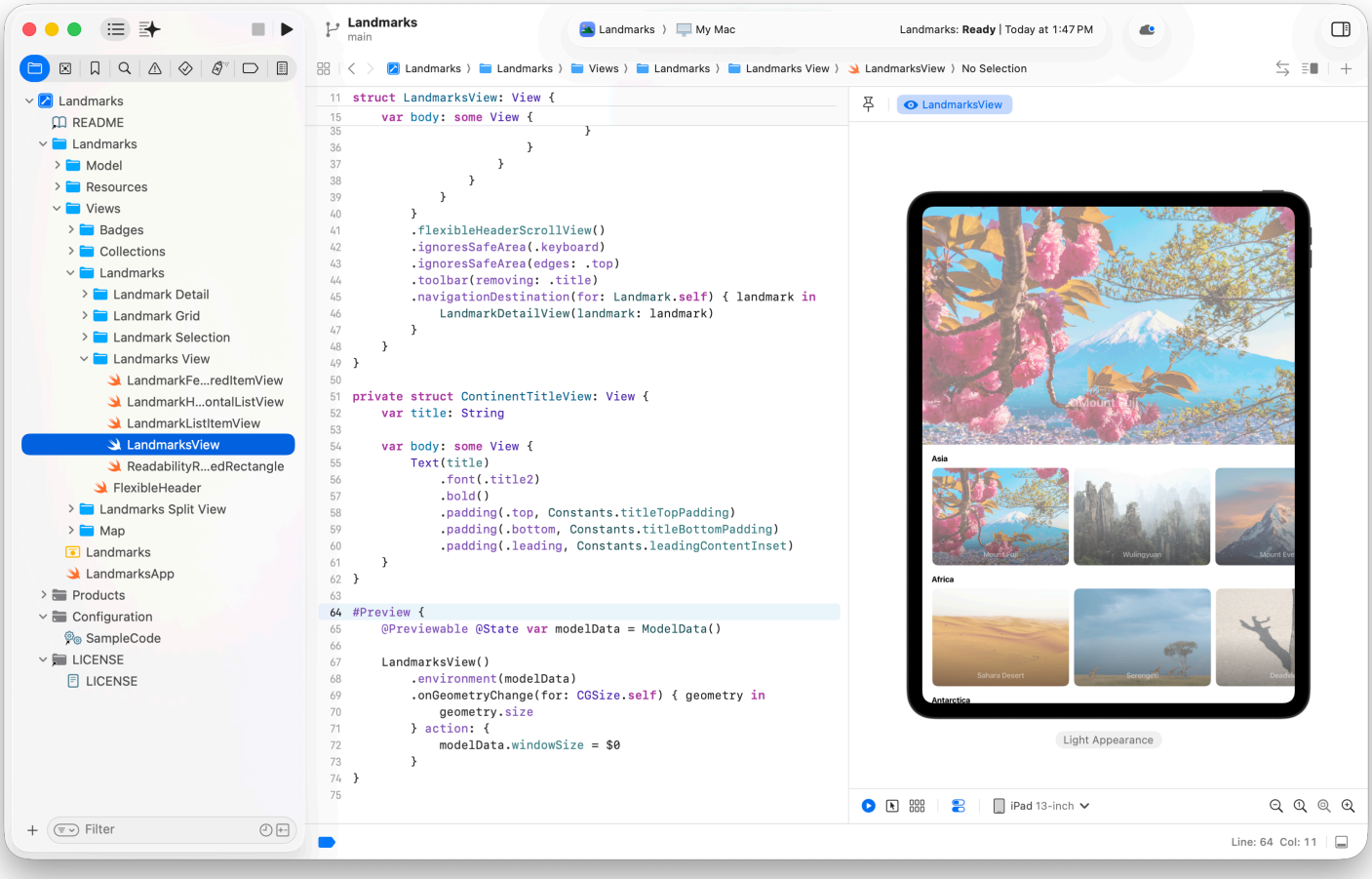
Dynamic Type Variants
: Displays your view in all the accessibility text sizes for your app.

Experiment

Because variant mode shows all the values for a given device setting, you can override what variant mode displays by making further changes in Canvas Device Settings. For example, to see how your view appears in different sizes of text in dark appearance, toggle Dynamic Type on, toggle Color Scheme on, and select Dark Appearance under Color Scheme.

Preview on a specific device

To see how your view displays on a specific device, choose the device from the Preview Device pop-up menu at the bottom of the preview canvas. When you do, Xcode displays a preview of your view on that device.



Capture specific previews in code

In addition to the preview options Xcode provides, you can also customize and configure previews you want to reuse programmatically.

For example, you can add a name to more easily track what each preview displays. When you pass the name of your preview as a string into the preview macro, the name appears in the title of the preview in the preview canvas.

```
// A preview with an assigned name.
#Preview("2x2 Grid Portrait") {
   Content()
}
```

> **Note**
>
> If you add multiple preview and playground macros to a file, you can switch between them using the tabs that appear at the top of the canvas. Xcode uses the name that you pass to the macro as the label for that preview. To add playgrounds to your Swift code, see Running code snippets using the playground macro.

You can also control how your preview displays by passing one or more configuration traits as a variadic argument list into the preview macro. For example, to display your view in the landscape left orientation, pass the `landscapeLeft` type property into the `init(_:traits:body:)` preview initializer to tell Xcode which orientation to display.

SwiftUI UIKit AppKit

```
// A SwiftUI preview with name and orientation.
#Preview("2x2 grid", traits: .landscapeLeft) {
    CollageView(layout: .twoByTwoGrid)
}
```

Use inline dynamic properties with Previewable

When a view depends on a `Binding` property wrapper, you can create a functional binding for that property and pass it into your preview using the `Previewable()` macro. This macro works on any variable conforming to the `DynamicProperty` protocol.

```
struct PlayButton: View {
    @Binding var isPlaying: Bool

    var body: some View {
        Button(action: {
            self.isPlaying.toggle()
        }) {
            Image(systemName: isPlaying ? "pause.circle" : "play.circle")
            .resizable()
            .scaledToFit()
            .frame(maxWidth: 80)
        }
    }
}

#Preview {
    // Tag the dynamic property with `Previewable`.
    @Previewable @State var isPlaying = true

    // Pass it into your view.
    PlayButton(isPlaying: $isPlaying)
}
```

Tagging a dynamic property with the `Previewable` macro gets rid of the need to create wrapper views in previews.

> **Note**
>
> `Previewable()` is a SwiftUI only macro and doesn't apply to UIKit or AppKit previews.

Make complex objects reusable with a preview modifier

To avoid recreating expensive objects for every preview that needs them, in SwiftUI you can create these objects once with the `PreviewModifier` and then pass the preview modifier into your preview using the `Preview(_:traits:_:body:)` macro.

Expensive objects — such as objects that make network calls, perform disk access, or just take considerable time and effort to setup — can make your previews take longer to load. By creating these expensive objects once, and sharing them across all your previews, you make your previews more efficient.

For example, if you have an app with an expensive `Observable()` object:

```
@Observable
class AppState {
    // An expensive, complex, bulky object.
    var expensiveObject = "Some expensive object"
}

@main
struct MyApp: App {
    @State private var appState = AppState()

    var body: some Scene {
        WindowGroup {
            ComplexView()
                .environment(appState)
        }
    }
}
```

You reuse that expensive object across multiple views in your app:

```
struct ComplexView: View {
    @Environment(AppState.self) var appState

    var body: some View {
        Text("\(appState.expensiveObject)")
    }
}
```

For every view you want to preview, you recreate and pass in that expensive object:

```
#Preview {
    ComplexView()
        // Potentially expensive if `AppState` is large or complex.
        .environment(AppState())
}
```

Instead, define the expensive object once and share it across multiple previews using the `PreviewModifier` protocol.

1. Define a structure conforming to the `PreviewModifier` protocol.

2. Implement the static `makeSharedContext()` function returning the object with the expensive state.
3. Inject that shared context into the view you want to preview using the `body(content:context:)` function.
4. Add the modifier to the preview using the `Preview(_:traits:_:body:)` macro.

```
// Create a struct conforming to the PreviewModifier protocol.
struct SampleData: PreviewModifier {

    // Define the object to share and return it as a shared context.
    static func makeSharedContext() async throws -> AppState {
        let appState = AppState()
        appState.expensiveObject = "An expensive object to reuse in previews"
        return appState
    }

    func body(content: Content, context: AppState) -> some View {
        // Inject the object into the view to preview.
        content
            .environment(context)
    }
}

// Add the modifier to the preview.
#Preview(traits: .modifier(SampleData())) {
    ComplexView()
}
```

Pass views only the data they need

When creating views, pass in only the data the view needs to display. Avoid passing in objects that fetch data; objects make setting up a view's preview more complicated and less performant.

Instead, create views with the minimal amount of data they need, favoring simpler, immutable data types. Creating views this way makes testing and previewing your views easier and helps them perform better.

The following example shows how simple data types, like `String` and `enum`, can be used to preview a view in various ways using the preview macro.

SwiftUI UIKit AppKit

```
struct CollaboratorCell: View {
    // Construct your view with only the data it needs.
    let name: String
    let image: Image?
    let connectionStatus: ConnectionStatus

    enum ConnectionStatus {
        case online
        case offline
    }

    // ...
}

#Preview("Supported cell combinations", traits: .sizeThatFitsLayout) {
    let image = Image(systemName: "person.circle")
    VStack {
        // Then test each scenario in your preview macro.
        CollaboratorCell(name: "Tom Clark", image: nil, connectionStatus: .offline)
        CollaboratorCell(name: "Tom Clark", image: image, connectionStatus: .offline
        CollaboratorCell(name: "Tom Clark", image: nil, connectionStatus: .online)
        CollaboratorCell(name: "Tom Clark", image: image, connectionStatus: .online)
        CollaboratorCell(name: "Tom Long Middle Clark", image: nil, connectionStatus
        CollaboratorCell(name: "Tom Long Middle Clark", image: image, connectionStat
    }
}
```

Reduce your app size with development assets

To access resources in your previews, without shipping them in the final version of your app, use development assets. Development assets give you access to resources such as images, video, JSON data, and code files in your previews and Simulator, without increasing the overall size of your app.

Add items to the Development Assets of a project target as follows:

1. Select the project folder in the Project navigator.
2. Select the target you want to add the development assets to.
3. In the General tab, scroll down to Development Assets.
4. In the lower-left corner, click the Add items button (+).

5. In the dialog that appears, select the items that you want to add and click Add.

See Also

Essentials

- Creating an Xcode project for an app
  Start developing your app by creating an Xcode project from a template.
- Creating your app's interface with SwiftUI
  Develop apps in SwiftUI with an interactive preview that keeps the code and layout in sync.
- Building and running an app
  Compile your source files and assemble an app bundle to run on a device or simulator.
- Xcode updates
  Learn about important changes to Xcode.