Structure

State

A property wrapper type that can read and write a value managed by SwiftUI.

iOS 13.0+ | iPadOS 13.0+ | Mac Catalyst 13.0+ | macOS 10.15+ | tvOS 13.0+ | visionOS 1.0+ | watchOS 6.0+

```
@frozen @propertyWrapper
struct State<Value>
```

Mentioned in

- Managing user interface state
- Performing a search operation
- Understanding the navigation stack

Overview

Use state as the single source of truth for a given value type that you store in a view hierarchy. Create a state value in an App, Scene, or View by applying the @State attribute to a property declaration and providing an initial value. Declare state as private to prevent setting it in a memberwise initializer, which can conflict with the storage management that SwiftUI provides:

```
struct PlayButton: View {
    @State private var isPlaying: Bool = false // Create the state.

    var body: some View {
        Button(isPlaying ? "Pause" : "Play") { // Read the state.
            isPlaying.toggle() // Write the state.
        }
```

```
    }
}
```

SwiftUI manages the property's storage. When the value changes, SwiftUI updates the parts of the view hierarchy that depend on the value. To access a state's underlying value, you use its `wrappedValue` property. However, as a shortcut Swift enables you to access the wrapped value by referring directly to the state instance. The above example reads and writes the `isPlaying` state property's wrapped value by referring to the property directly.

Declare state as private in the highest view in the view hierarchy that needs access to the value. Then share the state with any subviews that also need access, either directly for read-only access, or as a binding for read-write access. You can safely mutate state properties from any thread.

Share state with subviews

If you pass a state property to a subview, SwiftUI updates the subview any time the value changes in the container view, but the subview can't modify the value. To enable the subview to modify the state's stored value, pass a `Binding` instead.

For example, you can remove the `isPlaying` state from the play button in the above example, and instead make the button take a binding:

```
struct PlayButton: View {
    @Binding var isPlaying: Bool // Play button now receives a binding.

    var body: some View {
        Button(isPlaying ? "Pause" : "Play") {
            isPlaying.toggle()
        }
    }
}
```

Then you can define a player view that declares the state and creates a binding to the state. Get the binding to the state value by accessing the state's `projectedValue`, which you get by prefixing the property name with a dollar sign ($):

```
struct PlayerView: View {
    @State private var isPlaying: Bool = false // Create the state here now.

    var body: some View {
        VStack {
            PlayButton(isPlaying: $isPlaying) // Pass a binding.
```

```
            // ...
        }
    }
}
```

Like you do for a `StateObject`, declare `State` as private to prevent setting it in a memberwise initializer, which can conflict with the storage management that SwiftUI provides. Unlike a state object, always initialize state by providing a default value in the state's declaration, as in the above examples. Use state only for storage that's local to a view and its subviews.

Store observable objects

You can also store observable objects that you create with the `Observable()` macro in `State`; for example:

```
@Observable
class Library {
    var name = "My library of books"
    // ...
}

struct ContentView: View {
    @State private var library = Library()

    var body: some View {
        LibraryView(library: library)
    }
}
```

A `State` property always instantiates its default value when SwiftUI instantiates the view. For this reason, avoid side effects and performance-intensive work when initializing the default value. For example, if a view updates frequently, allocating a new default object each time the view initializes can become expensive. Instead, you can defer the creation of the object using the `task(priority:_:)` modifier, which is called only once when the view first appears:

```
struct ContentView: View {
    @State private var library: Library?

    var body: some View {
        LibraryView(library: library)
```

```
            .task {
                library = Library()
            }
    }
}
```

Delaying the creation of the observable state object ensures that unnecessary allocations of the object doesn't happen each time SwiftUI initializes the view. Using the `task(priority: :)` modifier is also an effective way to defer any other kind of work required to create the initial state of the view, such as network calls or file access.

> **Note**
>
> It's possible to store an object that conforms to the `ObservableObject` protocol in a `State` property. However the view will only update when the reference to the object changes, such as when setting the property with a reference to another object. The view will not update if any of the object's published properties change. To track changes to both the reference and the object's published properties, use `StateObject` instead of `State` when storing the object.

Share observable state objects with subviews

To share an `Observable` object stored in `State` with a subview, pass the object reference to the subview. SwiftUI updates the subview anytime an observable property of the object changes, but only when the subview's `body` reads the property. For example, in the following code `BookView` updates each time `title` changes but not when `isAvailable` changes:

```
@Observable
class Book {
    var title = "A sample book"
    var isAvailable = true
}

struct ContentView: View {
    @State private var book = Book()

    var body: some View {
        BookView(book: book)
    }
}

struct BookView: View {
```

```
    var book: Book

    var body: some View {
        Text(book.title)
    }
}
```

`State` properties provide bindings to their value. When storing an object, you can get a `Binding` to that object, specifically the reference to the object. This is useful when you need to change the reference stored in state in some other subview, such as setting the reference to `nil`:

```
struct ContentView: View {
    @State private var book: Book?

    var body: some View {
        DeleteBookView(book: $book)
            .task {
                book = Book()
            }
    }
}

struct DeleteBookView: View {
    @Binding var book: Book?

    var body: some View {
        Button("Delete book") {
            book = nil
        }
    }
}
```

However, passing a `Binding` to an object stored in `State` isn't necessary when you need to change properties of that object. For example, you can set the properties of the object to new values in a subview by passing the object reference instead of a binding to the reference:

```
struct ContentView: View {
    @State private var book = Book()

    var body: some View {
        BookCheckoutView(book: book)
```

```
    }
}

struct BookCheckoutView: View {
    var book: Book

    var body: some View {
        Button(book.isAvailable ? "Check out book" : "Return book") {
            book.isAvailable.toggle()
        }
    }
}
```

If you need a binding to a specific property of the object, pass either the binding to the object and extract bindings to specific properties where needed, or pass the object reference and use the `Bindable` property wrapper to create bindings to specific properties. For example, in the following code `BookEditorView` wraps `book` with `@Bindable`. Then the view uses the $ syntax to pass to a `TextField` a binding to `title`:

```
struct ContentView: View {
    @State private var book = Book()

    var body: some View {
        BookView(book: book)
    }
}

struct BookView: View {
    let book: Book

    var body: some View {
        BookEditorView(book: book)
    }
}

struct BookEditorView: View {
    @Bindable var book: Book

    var body: some View {
        TextField("Title", text: $book.title)
    }
}
```

Topics

Creating a state

`init(wrappedValue: Value)`

Creates a state property that stores an initial wrapped value.

`init(initialValue: Value)`

Creates a state property that stores an initial value.

`init()`

Creates a state property without an initial value.

Getting the value

`var wrappedValue: Value`

The underlying value referenced by the state variable.

`var projectedValue: Binding<Value>`

A binding to the state value.

Relationships

Conforms To

`DynamicProperty`, `Sendable`, `SendableMetatype`

See Also

Creating and sharing view state

Managing user interface state

Encapsulate view-specific data within your app's view hierarchy to make your views reusable.

`struct` **`Bindable`**

A property wrapper type that supports creating bindings to the mutable properties of observable objects.

`struct` **`Binding`**

A property wrapper type that can read and write a value owned by a source of truth.