@ObservedObject

The `@ObservedObject` property wrapper allows you to observe changes to a property of an object and automatically update your view accordingly.

To use `@ObservedObject`, you must first create a class that conforms to the `ObservableObject` protocol. This protocol requires the class to have a `@Published` property for each property that you want to observe.

Once you have created an `ObservableObject` class, you can bind it to your view using the `@ObservedObject` property wrapper. When the `ObservableObject` class's `@Published` properties change, the view will be automatically updated.

Here is an example of how to use `@ObservedObject`:

```
Swift
struct ObservableObjectClass: ObservableObject {
    @Published var count: Int = 0
}

struct ContentView: View {
    @ObservedObject var observableObject = ObservableObjectClass()

    var body: some View {
        Text("\(observableObject.count)")
    }
}
```

In this example, the `ContentView` struct has an `@ObservedObject` property that binds it to the `ObservableObjectClass` class. When the `count` property of the `ObservableObjectClass` class changes, the `ContentView` struct will be automatically updated to display the new value of the `count` property.

@Binding

The `@Binding` property wrapper allows you to bind a property of your view to a property of another view. This allows you to update the other view's property by changing the value of your view's property.

To use `@Binding`, you must first declare a property in your view using the `@Binding` property wrapper. Then, you must pass the binding to your view's

initializer.

Once you have passed the binding to your view's initializer, you can access the other view's property using the binding. When you change the value of the binding, the other view's property will be automatically updated.

Here is an example of how to use `@Binding`:

```
Swift
struct ChildView: View {
    @Binding var count: Int

    var body: some View {
        Button("Increment") {
            count += 1
        }
    }
}

struct ParentView: View {
    @State var count = 0

    var body: some View {
        ChildView(count: $count)
    }
}
```

In this example, the `ParentView` struct has a `@State` property called `count`. The `ChildView` struct has a `@Binding` property called `count` that is bound to the `count` property of the `ParentView` struct.

When the user taps the button in the `ChildView` struct, the `count` property of the `ParentView` struct will be incremented. The `ParentView` struct will then be automatically updated to display the new value of the `count` property.

.onReceive

The `.onReceive` modifier allows you to observe changes to a value and execute a closure when the value changes.

To use `.onReceive`, you must first pass the value that you want to observe to the `.onReceive` modifier. Then, you must pass a closure to the `.onReceive` modifier that will be executed when the value changes.

Here is an example of how to use `.onReceive`:

```
Swift
struct ContentView: View {
    @State var count = 0

    var body: some View {
        Text("\(count)")
            .onReceive(count) { count in
                print("The count is now \(count)")
            }
    }
}
```

In this example, the `ContentView` struct has a `@State` property called `count`. The `.onReceive` modifier is used to observe changes to the `count` property. When the `count` property changes, the closure passed to the `.onReceive` modifier will be executed.

@EnvironmentObject

The `@EnvironmentObject` property wrapper allows you to access an `ObservableObject` instance that has been registered with the environment.

To use `@EnvironmentObject`, you must first declare a property in your view using the `@EnvironmentObject` property wrapper. Then, you can access the `ObservableObject` instance using the property.

Here is an example of how to use `@EnvironmentObject`:

```
Swift
struct ObservableObjectClass: ObservableObject {
    @Published var count: Int = 0
}

struct ChildView: View {
    @EnvironmentObject var observableObject: ObservableObjectClass

    var body: some View {
        Text("\(observableObject.count)")
    }
}

struct ParentView: View {
    @State var observableObject = ObservableObjectClass()
```

```
    var body: some View {
        ChildView()
            .environmentObject(observableObject)
    }
}
```

In this example, the `ParentView` struct has a `@State` property called `observableObject`. The `ChildView` struct has an `@EnvironmentObject` property called `observableObject` that is bound to the `observableObject` property of the `ParentView` struct.

Even though the `ChildView` struct does not explicitly pass the `observableObject` instance to its initializer, it can still access it because it has been registered with the environment.