Interoperability

Building Bridges Between Languages and UI Frameworks

Swift & Objective-C Interoperability

1. Legacy Objective-C codebases
2. Access to existing libraries
3. Gradual migration to Swift
4. Team collaboration

Bridging Header

1. Access Objective-C classes
2. Use Objective-C methods
3. Import Objective-C frameworks

ProjectName-Bridging-Header.h

```
#import "Person.h" #import "ObjCUser.h"
```

Swift to Objective-C

1. Use @objc attribute
2. Inherit from NSObject
3. Mark properties with @objc

```
@objc class SwiftPerson: NSObject {
    @objc let name: String
    @objc private(set) var age: Int

    @objc init(name: String, age: Int) {
        self.name = name
        self.age = age
    }
}
```

Objective-C to Swift

```
// Person.h
@interface Person : NSObject
@property (nonatomic, copy) NSString *name;
@property (nonatomic, assign) NSInteger age;
- (void)sayHello;
@end
```

In Swift, you can use it like:

```
let person = Person(name: "John", age: 30)
person.sayHello()
```

Type Mapping

Swift	Objective-C
String	NSString
Int, Float, Double	NSNumber
Array	NSArray
Dictionary	NSDictionary

SwiftUI & UIKit Interoperability

1. Gradual migration to SwiftUI
2. Reuse existing UIKit components
3. Access UIKit-only features
4. Combine modern and legacy code

UIKit in SwiftUI

1. Conform to UIViewControllerRepresentable
2. Implement makeUIViewController
3. Handle updates if needed

```
struct UIKitViewControllerRepresentable: UIViewControllerRepresentable {
    func makeUIViewController(context: Context) -> UIKitViewController {
        UIKitViewController()
    }

    func updateUIViewController(
        _ uiViewController: UIKitViewController,
        context: Context
    ) {}
}
```

SwiftUI in UIKit

1. Create SwiftUI view
2. Wrap in UIHostingController
3. Present using UIKit methods

```
@objc private func showSwiftUIView() {
    let swiftUIView = SwiftUIView {
        self.dismiss(animated: true, completion: nil)
    }
    let hostingController = UIHostingController(rootView: swiftUIView)
    present(hostingController, animated: true)
}
```

Data Flow

1. Closures for callbacks
2. Bindings for state
3. ObservableObject for complex data

```
// SwiftUI to UIKit
struct SwiftUIView: View {
    let dismiss: () -> Void  // Closure for communication

    var body: some View {
        Button("Dismiss") {
            dismiss()
        }
    }
}
```

Best Practices

1. Keep framework-specific code isolated
2. Use coordinators for complex navigation
3. Handle memory management carefully
4. Test integration points thoroughly