iOS Unit and UI Testing

From Basics to Advanced Techniques

Introduction

1. Introduction to Testing in iOS
2. Unit Testing Basics
3. Writing Effective Unit Tests
4. Advanced Unit Testing Techniques
5. Introduction to UI Testing
6. Understanding XCUIApplication and UI Element

Introduction to Testing in iOS

1. What is software testing?

2. Why is testing important?

3. Unit Types of tests in iOS development

    - Unit tests

    - Integration tests

    - UI tests

Introduction to Testing in iOS

1. What is software testing?

2. Why is testing important?

3. Unit Types of tests in iOS development

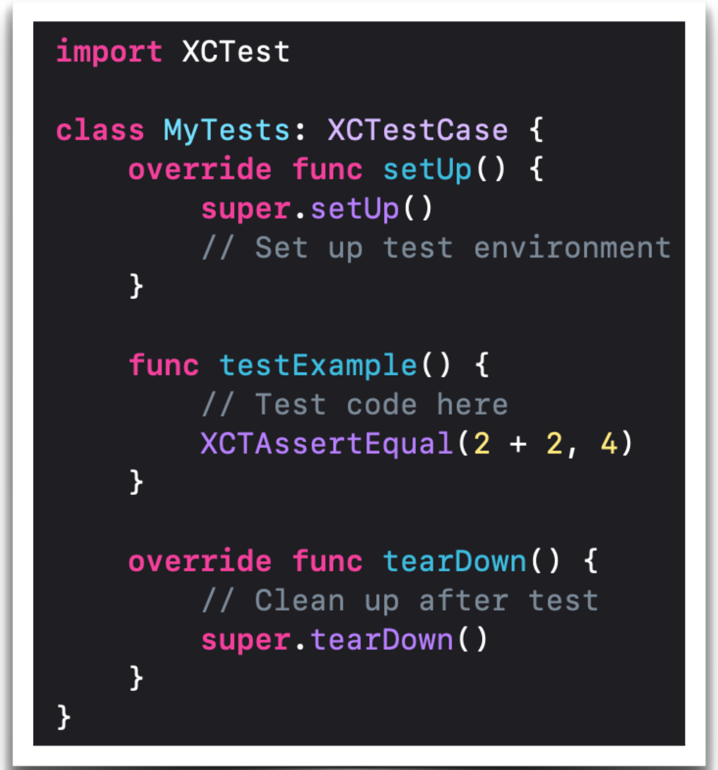
    - Unit tests

    - Integration tests

    - UI tests

Unit Testing Basics

1. **What is a unit test**? - A method that instantiates a small portion of our application and verifies its behaviour independently from other parts.
2. **XCTest framework** - Apple's testing framework for Swift and Objective-C, integrated into Xcode.
3. Anatomy of a test case
   - Test class: Inherits from XCTestCase
   - setUp() and tearDown() methods for test initialization and cleanup
   - Individual test methods prefixed with 'test'



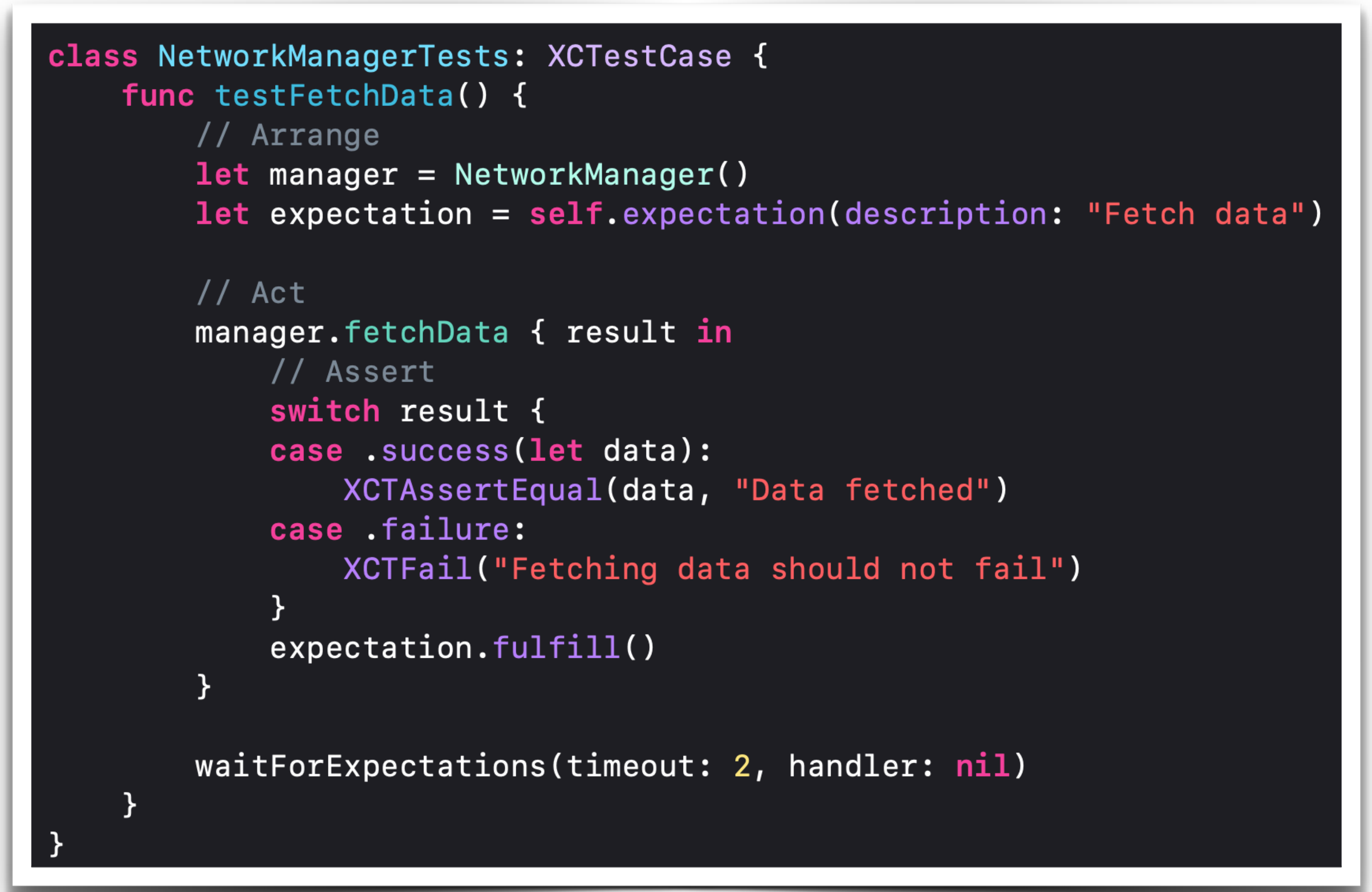
```swift
import XCTest

class MyTests: XCTestCase {
    override func setUp() {
        super.setUp()
        // Set up test environment
    }

    func testExample() {
        // Test code here
        XCTAssertEqual(2 + 2, 4)
    }

    override func tearDown() {
        // Clean up after test
        super.tearDown()
    }
}
```

XCTest Commands

Command	Description
XCTAssertTrue()	Asserts that an expression is true
XCTAssertFalse()	Asserts that an expression is false
XCTAssertEqual()	Asserts that two values are equal
XCTAssertNotEqual()	Asserts that two values are not equal
XCTAssertNil()	Asserts that an expression is nil
XCTAssertNotNil()	Asserts that an expression is not nil
XCTFail()	Generates a failure immediately

Writing Effective Unit Tests

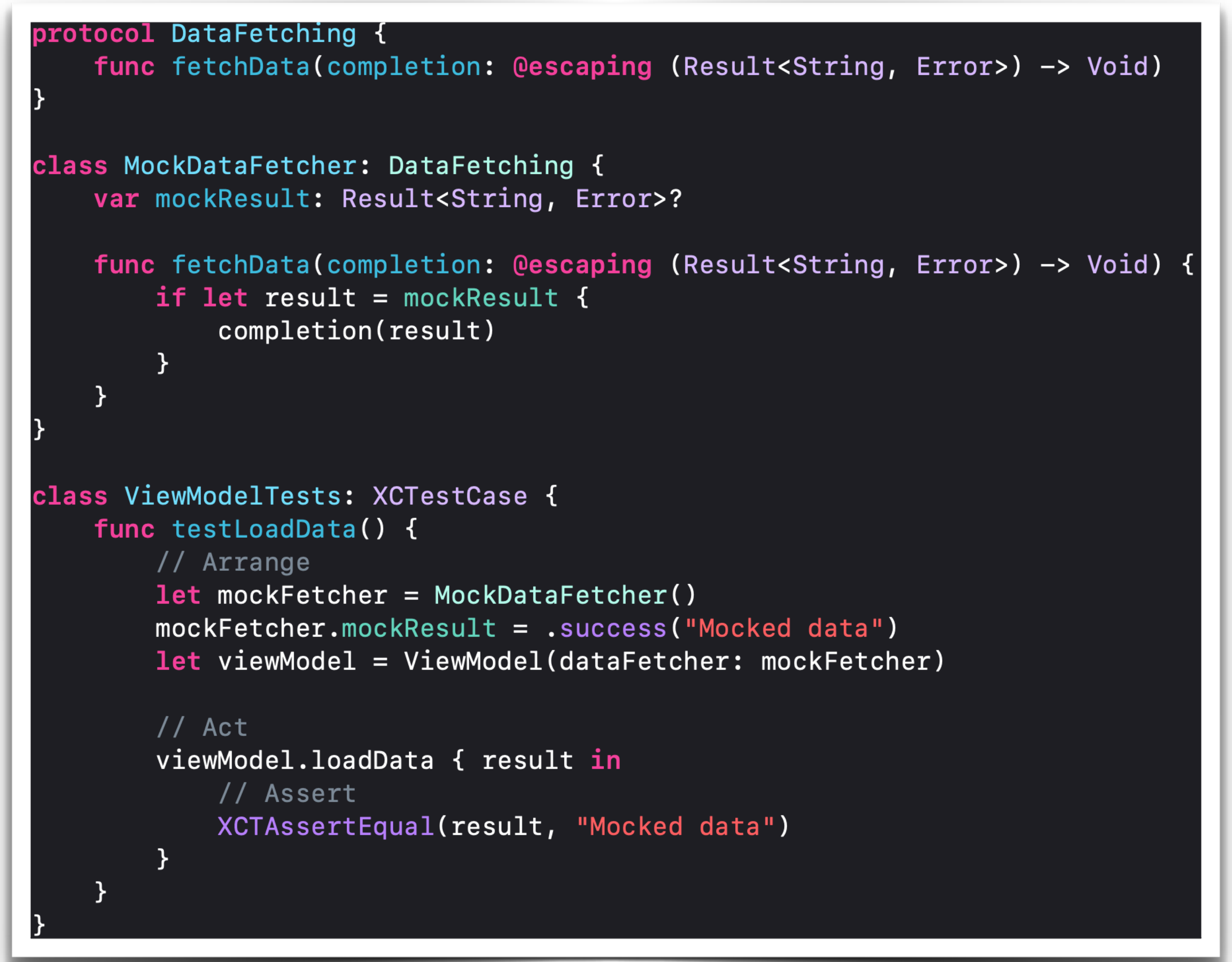
1. **Naming conventions** - Use descriptive name that explains test purpose
2. Arrange-Act-Assert pattern
3. **Testing asynchronous code** - Use XCTestExpectation for async operations
4. **Mocking and stubbing** - Create fake objects to isolate the unit being tested



```swift
class NetworkManagerTests: XCTestCase {
    func testFetchData() {
        // Arrange
        let manager = NetworkManager()
        let expectation = self.expectation(description: "Fetch data")

        // Act
        manager.fetchData { result in
            // Assert
            switch result {
            case .success(let data):
                XCTAssertEqual(data, "Data fetched")
            case .failure:
                XCTFail("Fetching data should not fail")
            }
            expectation.fulfill()
        }

        waitForExpectations(timeout: 2, handler: nil)
    }
}
```

Advanced Unit Testing Techniques

1. Testing with protocols - Use protocol-oriented programming to make code more testable
2. Dependency injection - Inject dependencies to make units more isolated and easier to test
3. Performance testing - Use measure() method to test performance-critical code
4. Code coverage - Use Xcode's code coverage feature to identify untested code paths

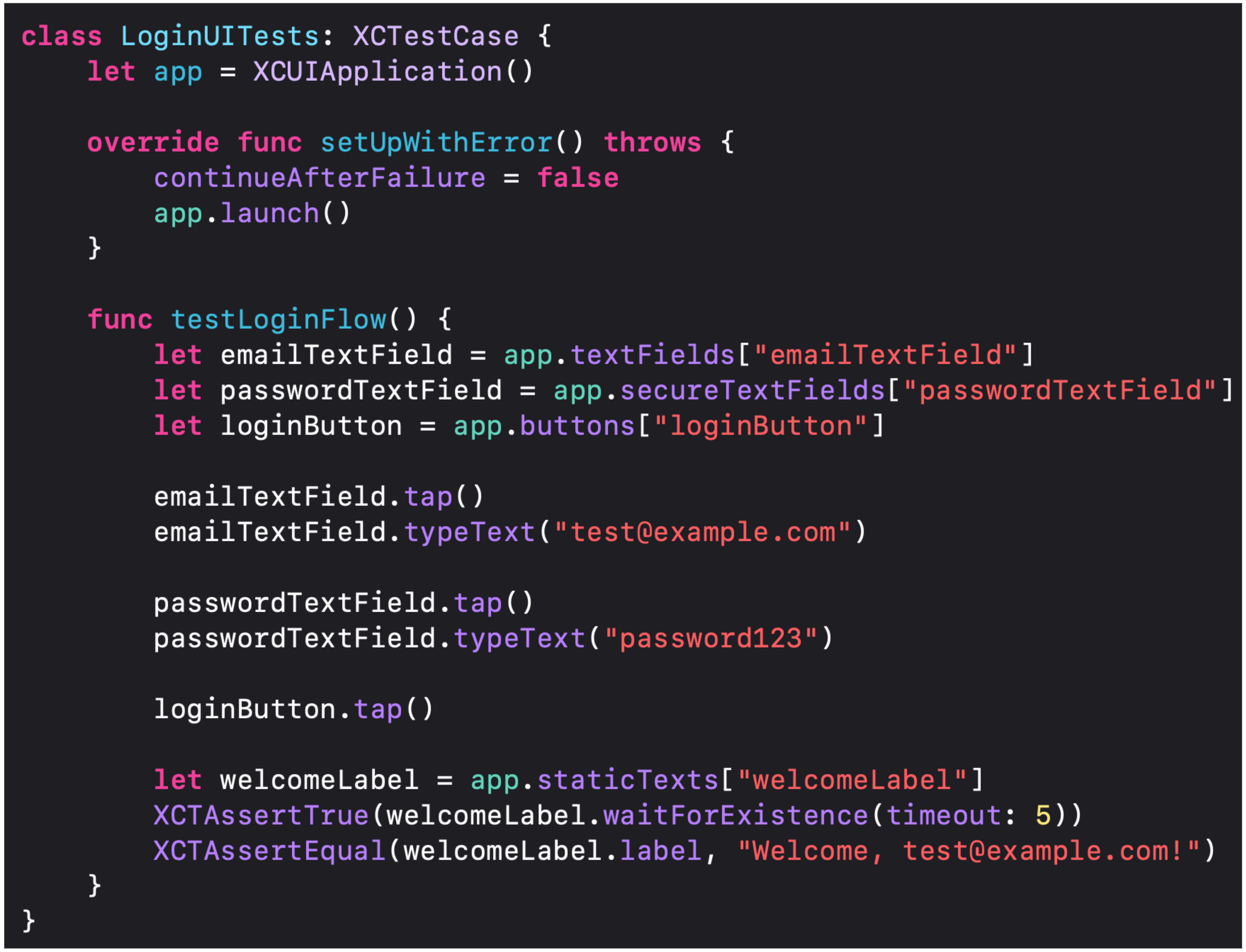


```swift
protocol DataFetching {
    func fetchData(completion: @escaping (Result<String, Error>) -> Void)
}

class MockDataFetcher: DataFetching {
    var mockResult: Result<String, Error>?

    func fetchData(completion: @escaping (Result<String, Error>) -> Void) {
        if let result = mockResult {
            completion(result)
        }
    }
}

class ViewModelTests: XCTestCase {
    func testLoadData() {
        // Arrange
        let mockFetcher = MockDataFetcher()
        mockFetcher.mockResult = .success("Mocked data")
        let viewModel = ViewModel(dataFetcher: mockFetcher)

        // Act
        viewModel.loadData { result in
            // Assert
            XCTAssertEqual(result, "Mocked data")
        }
    }
}
```

Introduction to UI Testing

1. **What is UI testing**? - Automated tests that interact with your app's user interface to verify correct behavior.
2. **XCUITest framework** - Apple's UI testing framework, an extension of XCTest
3. **Recording UI tests in Xcode** - Using Xcode's UI recording feature to generate test code
4. **Writing UI tests manually** - Creating custom UI tests for more complex scenarios Use protocol-oriented programming to make code more testable

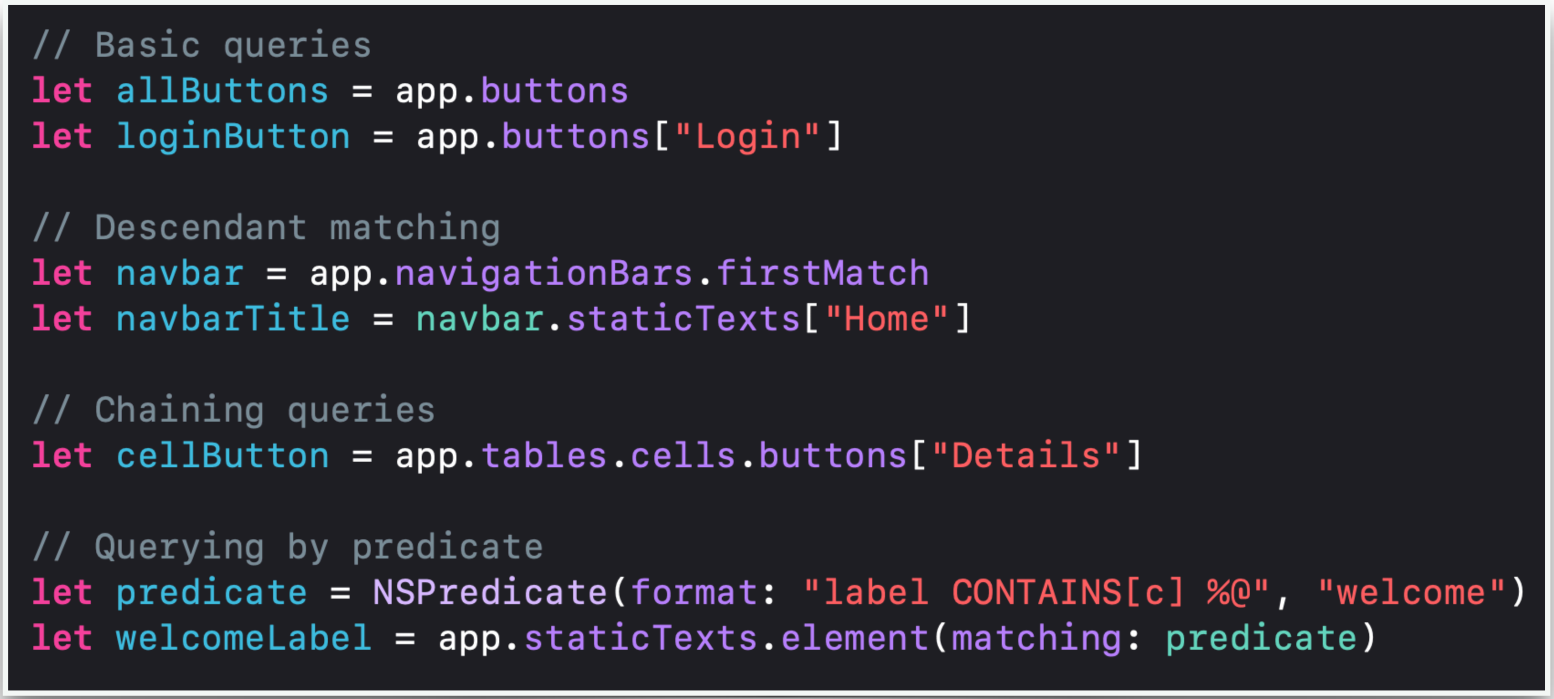
Introduction to UI Testing



```swift
class LoginUITests: XCTestCase {
    let app = XCUIApplication()

    override func setUpWithError() throws {
        continueAfterFailure = false
        app.launch()
    }

    func testLoginFlow() {
        let emailTextField = app.textFields["emailTextField"]
        let passwordTextField = app.secureTextFields["passwordTextField"]
        let loginButton = app.buttons["loginButton"]

        emailTextField.tap()
        emailTextField.typeText("test@example.com")

        passwordTextField.tap()
        passwordTextField.typeText("password123")

        loginButton.tap()

        let welcomeLabel = app.staticTexts["welcomeLabel"]
        XCTAssertTrue(welcomeLabel.waitForExistence(timeout: 5))
        XCTAssertEqual(welcomeLabel.label, "Welcome, test@example.com!")
    }
}
```

Understanding XCUIApplication and UI Element Selection

1. **XCUIApplication** - The main entry point for UI Testing, representing your app
2. **Querying for UI elements** - Using element queries to find and interact with UI components
3. **Working with different UI controls** - Interacting with buttons, text fields, switches, etc.
4. **Best practices for element identification** - Using accessibility identifiers for reliable element selection



```swift
// Basic queries
let allButtons = app.buttons
let loginButton = app.buttons["Login"]

// Descendant matching
let navbar = app.navigationBars.firstMatch
let navbarTitle = navbar.staticTexts["Home"]

// Chaining queries
let cellButton = app.tables.cells.buttons["Details"]

// Querying by predicate
let predicate = NSPredicate(format: "label CONTAINS[c] %@", "welcome")
let welcomeLabel = app.staticTexts.element(matching: predicate)
```