Saint Louis University
School of Accountancy, Management, Computing and Information Studies
Department of Computing and Information Studies

Applications Development
CS 311 - 9374
MTH 4:30-5:30 / WS 1:30-3:00

TEAM 3 SWIFT CODING STANDARD

Members:
Agustin, Aljay
Aque, Eurecho
Avillanoza, Felitra
Diola, Josh McKenzie
Natividad, Rhyen Jan
Roque, Genrev
Sollorin, John Michael

Professor:
Sir Roderick Makil

Submission Date:
November 22, 2023

TABLE OF CONTENTS

I. NAMING CONVENTION

A. Package Names

1. Use the CamelCase naming convention.
2. Avoid using underscores.

```
Unset
Package (name:"StudenAttendance")
```

B. Module Names

1. Use short lowercase words separated with underscores.

```
Unset
// Module for handling user authentication
public struct UserAuth {
    // ...
}
```

C. Variable Names

1. Use camelCase.
2. Be descriptive and avoid abbreviations unless widely known.

```
Unset
var firstName: String
func appendFirstname() {}
```

D. Types(Classes, Structs, Enums)

1. Use PascalCase.
2. Be explicit and choose meaningful names.

```
Unset
struct familyNames{
  // properties and methods
}
```

E. Constants and Enum Cases:

1. Use uppercase and underscore for constants to easily distinguish from variables
2. Start enum cases with a capital letter.

```
Unset
//Constants
let MAX_NUMBER_OF_STUDENTS = 50

//enum
enum StudentStatus{
 case complete
 case incomplete
 case noFinalExam
}
```

F. Method Names

1. Use `camelCase` (initial lowercase letter) for function or method,

Acceptable: Using camelCase with an initial lowercase letter. If the function name is more than a word, add underscore after the first word.

```
Unset
func assign_StudentNumber() -> Int {
  // Function implementation
  return studentNumber
}
```

Unacceptable: Using a name with an initial uppercase letter.

```
Unset

func AssignStudenNumber() -> Int {
  // Function implementation
  return StudentNumber
}
```

G. Folders and File Names

It is recommended to use CamelCase for writing the names of folders and files and giving them a descriptive name.

Recommended

```
Unset
CustomerCart.swift
CustomerInterface.swift
ManagerInterface.swift
StoreItems.swift
```

Avoid

```
Unset
Cart.swift
UI.swift
Items.swift
```

II. FORMATTING

- No one wants to work with a clumsy developer and a developer who is not consistent in their coding style is bound to be clumsy with their code.

A. Line Length limit

1. a single line of code should not exceed 120 characters.
2. Consider breaking long expressions or statements into multiple lines for better clarity.

Acceptable:

```
Unset
// Breaking it into multiple lines
let sampleSentence = "This string does not exceed the recommended length, " +
        "because it's broken into multiple lines for better readability."
```

Unacceptable:

```
Unset
// Original code with a line length violation
let sampleSentence= "This is a long string that exceeds the recommended line length,
so it's broken into multiple lines for better readability."
```

B. Spaces

1. There should generally be no more than one space between any two characters.

Acceptable:

```
Unset
var oneSpace = 0;
```

Unacceptable:

```
Unset
var   threeSpaces = 0 ;
```

C. Blank Lines

1. There should be no more than one blank line between any two lines of code.

Acceptable:

```
Unset
print("Line of Code")

print("One Blank Line")
```

```
Unset
print("Line of Code")
print("No Blank Lines")
```

Unacceptable:

```
Unset
print("Line of Code")



print("Too Many Blank Lines")
```

D. Braces

1. Opening braces: Must be consistent with placing the opening brace, can be placed either immediately following or on the next line.
2. Closing braces: Should always be on their own line and either lined up with the opening brace.

Acceptable:

```
Unset
if <#condition#> {
	<#statement/s#>
}
```

```
Unset
if <#condition#>
{
<#statements#>
}
```

Unacceptable:

```
Unset
if <#condition#> {
	<#statements#>
	}
```

```
Unset
if <#condition#>
	{
	<#statements#>
}
```

E. Semicolons

1. Trailing semicolons (`;`) are not allowed.
2. Swift does not require a semicolon after each statement in your code. It is only required to be used as a delimiter if you are using multiple statements on a single line.

Acceptable:

```
Unset
self.backgroundColor = UIColor.whiteColor()
self.completion = {
  // ...
}

for var i = 0; i < 5; i++ {
        // loop body
}

let newInt = 5; print(newInt)
```

Unacceptable:

```
Unset
self.backgroundColor = UIColor.whiteColor();
self.completion = {
  // ...
};
```

F. Properties

1. The get and set statement and their close braces (}) should all be left-aligned. If the statement in the braces can be expressed in a single line, the get and set declaration can be inlined.

Acceptable:

```
Unset
struct Rectangle {
// ...
var right: Float {

    get {

        return self.x + self.width
    }
    set {

        self.x = newValue - self.width
    }
}

}
```

Unacceptable:

```
Unset
struct Rectangle {
// ...
var right: Float {

    get
    {
        return self.x + self.width
    }
    set
    {
        self.x = newValue - self.width
    }
}

}
```

G. Property and Variable Declarations

1. A given property or variable declaration must be for one and only one property or variable.

Acceptable:

```
Unset
var firstName:String
var lastName:String
```

Unacceptable:

```
Unset
var firstName:String, lastName:String
```

H. Control Flow Statements

1. if, else, switch, do, catch, repeat, guard, for, while, and defer statements should be left-aligned with their respective close braces (}).

Acceptable:

```
Unset
if array.isEmpty {
    // ...
}
else {
    // ...
}
```

Unacceptable:

```
Unset
if array.isEmpty {
    // ...
} else {
    // ...
}
```

2. case statements should be left-aligned with the switch statement. Single-line case statements can be inlined and written compact. Multi-line case statements should be indented below case: and separated with one empty line.

Acceptable:

```
Unset
switch inputType{
case .Type1:
self.doFirst()
self.doSecond()
case .Type2:
self.doFirst()
self.doSecond()
}
```

Unacceptable:

```
Unset
switch result {
case .Success: self.doSomething()
self.doSomethingElse()
case .Failure: self.doSomething()
```

```
self.doSomethingElse()
}
```

3. Conditions for if, switch, for, and while statements should not be enclosed in parentheses (()).

Acceptable:

```
Unset
if array.isEmpty {
    // ...
}
```

Unacceptable:

```
Unset
if (array.isEmpty) {
    // ...
}
```

4. Try to avoid nesting statements by returning early when possible.

Acceptable:

```
Unset
guard let strongSelf = self else {
return

}
// do many things with strongSelf
```

Unacceptable:

```
Unset
if let strongSelf = self {
// do many things with strongSelf

}
```

III. CODE ORGANIZATION

A. Dependencies

1. import statements for OS frameworks and external frameworks should be separated and alphabetized.

Acceptable:

```
Unset
import Foundation
import UIKit
import Alamofire
import Cartography
import SwiftyJSON
```

Unacceptable:

```
Unset
import Foundation
import Alamofire
import SwiftyJSON
import UIKit
import Cartography
```

B. Declaration Order

The MARK comment is a useful tool for organizing code within a file, especially in larger projects. It is recognized by Xcode, the integrated development environment (IDE) for Swift and Objective-C. The MARK comment is followed by a hyphen and a brief description, creating a visually distinct section marker in the code editor.

1. All type declarations such as class, struct, enum, extension, and protocols, should be marked with // MARK: - <name of declaration> (with hyphen)

Acceptable:

```
Unset
// MARK: - Icon
class Icon {
// MARK: - CornerType
```

```
enum CornerType {

  case Square
  case Rounded
}
// ...

}
```

Unacceptable:

```
Unset
       // Icon
class Icon {
// MARK: CornerType

enum CornerType {

    case Square
    case Rounded
}
// ...

}
```

2. All properties and methods should be grouped into the superclass/protocol they implement and should be tagged with `// MARK: <superclass/protocol name>`. The rest should be marked as either `// MARK: Public`, `// MARK: Internal`, or `// MARK: Private`.

C. File Encoding

1. Source files are encoded in UTF-8.

D. Documentation

1. If a function is more complicated than a simple O(1) operation, you should generally consider adding a doc comment for the function since there could be some information that the method signature does not make immediately obvious. If there are any quirks to the way that something

was implemented, whether technically interesting, tricky, not obvious, etc., this should be documented. Documentation should be added for complex classes/structs/enums/protocols and properties. All `public` functions/classes/properties/constants/structs/enums/protocols/etc. should be documented as well (provided, again, that their signature/name does not make their meaning/functionality immediately obvious).

2. 160 character column limit (like the rest of the code).
3. Even if the doc comment takes up one line, use block (`/** */`).
4. Do not prefix each additional line with a `*`.
5. Use the new `- parameter` syntax as opposed to the old `:param:` syntax (make sure to use lower case `parameter` and not `Parameter`). Option-click on a method you wrote to make sure the quick help looks correct.

```
Unset
class Human {
    /**
     This method feeds a certain food to a person.

     - parameter food: The food you want to be eaten.
     - parameter person: The person who should eat the food.
     - returns: True if the food was eaten by the person; false otherwise.
    */
    func feed(_ food: Food, to person: Human) -> Bool {
        // ...
    }
}
```

6. When mentioning code, use code ticks

```
Unset
/**
 This does something with a `UIViewController`, perchance.
 - warning: Make sure that `someValue` is `true` before running this function.
 */
func myFunction() {
    /* ... */
}
```

7. When writing doc comments, prefer brevity where possible.

E. Directories

The most recommended structure for a Swift project is the Model-View-Controller by separating the data, user interfaces and controlling logic.

1. Source
2. Scripts
3. Plotting
4. Docs
5. Notebooks
6. Tests
7. Examples

IV. BEST PRACTICES

A. Error Handling

Guidelines:

1. When a function, such as readFile, encounters a situation where it can't perform its intended operation, it's common to use Swift's optionals (String?) to indicate failure by returning nil. However, for more informative and structured error handling, prefer Swift's try/catch mechanism
2. Prefer try/catch for error handling instead of returning optionals.
3. Use a custom error type, like CustomError, for structured error information.
4. Throw errors to convey meaningful reasons for failures.
5. Use optionals only when the result should semantically be nil, not when something goes wrong during the operation.

```
Unset
struct CustomError: Swift.Error {
    public let file: StaticString
    public let function: StaticString
    public let line: UInt
    public let message: String

        public  init(message:  String,  file:  StaticString  =  #file,  function:
StaticString = #function, line: UInt = #line) {
        self.file = file
        self.function = function
```

```
        self.line = line
        self.message = message
    }
}

func readFile(named filename: String) throws -> String {
    guard let file = openFile(named: filename) else {
        throw CustomError(message: "Unable to open file named \(filename).")
    }

    let fileContents = file.read()
    file.close()
    return fileContents
}

func printSomeFile() {
    do {
        let fileContents = try readFile(named: filename)
        print(fileContents)
    } catch {
        print(error)
    }
}
```

B. Use Test Driven Development (TDD)

Majority are "testing" applications all the time: They write the code, test the app manually (via the browser or the console/terminal), make code revisions, repeat. This is how they currently develop an app. But with TDD, the order is different:

TDD or Test Driven Development is the practice of writing tests before we write code. The idea is we would make assertions of what we would like our code to do before we actually write the implementation.

Guidelines:

1. Write Tests First:
    a. Begin by creating tests that define the expected behavior before implementing any new feature or addressing a bug.
2. TDD Cycle:
    a. Follow the Red-Green-Refactor cycle: Start with a failing test (Red), implement the minimum code to make the test pass (Green), and then refactor the code while maintaining functionality.
3. Tests as Living Documentation:

a. Recognize tests as living documentation, providing insights into the intended functionality of the code and serving as a reliable reference for future development.

Why Write The Tests First? TDD forces you to consider how you design your code.

C. Version Control

1. Git commits
    a. Start with a Type/Use Specific Commit Types:
        i. Begin your commit message with a type that describes the purpose:
            1. Feat: for adding a new feature
            2. Fix: for fixing a bug
            3. Docs: for documentation changes
            4. Style: for style or formatting adjustments
            5. Perf: for performance improvements
            6. Test: for adding or modifying tests
        ii. Write a Clear Message:
        iii. Craft a concise and clear commit message using the imperative mood. Describe what the commit does.
    b. Include Details (if needed):
    c. Optionally, provide more details in the description. Mention related tickets or issues for context.

Example:

```
Unset
git commit -m "Feat: Add login feature

Closes #123

Implemented a secure login feature for users to access the application. Changes
cover both backend API and frontend UI components."
```

2. Avoid Direct Commits to Main Branch
    a) Always create and work in feature branches for new features or enhancements.

```
Unset
git checkout -b feature/new-feature
# ... make changes ...
```

```
git checkout develop
git merge --no-ff feature/new-feature
git push origin develop
```

V. REFERENCES

Swift Style Guide. (n.d.). Google.github.io. https://google.github.io/swift/

Inc, A. (n.d.). *Swift.org*. Swift.org. Retrieved November 22, 2023, from https://swift.org/documentation/api-design-guidelines/

Improving build efficiency with good coding practices. (n.d.). Apple Developer Documentation. Retrieved November 22, 2023, from https://developer.apple.com/documentation/xcode/improving-build-efficiency-with-good-coding-practices

Documentation. (n.d.). Docs.swift.org. Retrieved November 22, 2023, from https://docs.swift.org/swift-book/documentation/the-swift-programming-language/statements/

Coding Standard for Swift Programs. (n.d.). Retrieved November 22, 2023, from https://www.cs.utexas.edu/~bulko/2023fall/SwiftCodingStandard.pdf

Sithara, M. S. (2020, February 28). *iOS Best Practices and SWIFT Coding Standards: A Developer's Guide*. Perfomatix | Product Engineering Services Company. https://www.perfomatix.com/ios-best-practices-and-swift-coding-standards/

Swift Coding Standards. (n.d.). Engineering.vokal.io. https://engineering.vokal.io/iOS/CodingStandards/Swift.md.html

Swift Style Guide. (2023, November 20). GitHub. https://github.com/linkedin/swift-style-guide

How to implement coding standards in your organization. (n.d.). Www.linkedin.com. https://www.linkedin.com/pulse/how-implement-coding-standards-your-organization-codacy/

How to Write a Good Git Commit Message | Git Best Practices. (n.d.). Www.gitkraken.com. https://www.gitkraken.com/learn/git/best-practices/git-commit-message#:~:text=Git%20Commit%20Message%20Structure&text=The%20commit%20message%20title%20is