Contents

1. Naming....3
1.1 Delegates....3
1.2 Use Type Inferred Context....3
1.3 Class Prefixes....4
2. Code Organisation....4
2.1 Protocol Conformance....4
2.2 Unused Code....5
2.3 Minimal Imports....6
3. Spacing....6
4. Comments....8
5. Class and Structure....8
5.1 Use of Self....8
5.2 Computed Properties....8
6. Function Declarations....9
7. Function Calls....10
8. Closure Expressions....10
9. Types....12
9.1 Constants....12
9.2 Optionals....13
9.3 Lazy Initialisation....15
9.4 Type Inference....15
9.5 Syntactic Sugar....16
10.Functions vs Methods....17
10.1 Free Function Exceptions....17
11. Memory Management....17
11.1 Extending object lifetime....18
12. Access Control....18
13. Control Flow....19
13.1 Ternary Operator....20
14. Golden Path....21
14.1 Failing Guards....22
15. Semicolons....23
16. Parentheses....23
17. Multi-line String Literals....24
18. Folder structure *....25

19. # Mark Arrangement * 26

1. Naming

1.1 Delegates

When creating custom delegate methods, an unnamed first parameter should be the delegate source. (UIKit contains numerous examples of this.)

Preferred:

```
func namePickerView(_ namePickerView: NamePickerView, didSelectName name: String)
```

```
func namePickerViewShouldReload(_ namePickerView: NamePickerView) -> Bool
```

Not Preferred:

```
func didSelectName(namePicker: NamePickerViewController, name: String)
```

```
func namePickerShouldReload() -> Bool
```

1.2 Use Type Inferred Context

Use compiler inferred context to write shorter, clear code.

Preferred:

```
let selector = #selector(viewDidLoad)
view.backgroundColor = .red
let toView = context.view(forKey: .to)
let view = UIView(frame: .zero)
```

Not Preferred:

```
let selector = #selector(ViewController.viewDidLoad)
view.backgroundColor = UIColor.red
let toView = context.view(forKey: UITransitionContextViewKey.to)
let view = UIView(frame: CGRect.zero)
```

1.3 Class Prefixes

Application first name and name should used ad prefix. Prefix should be mandatorily appended to all files.

Example : Mahindra connect - MC, Filename - MCLoginViewController.

2. Code Organisation

Use extensions to organise your code into logical blocks of functionality. Each extension should be set off with a // MARK: - comment to keep things well-organized.

2.1 Protocol Conformance

In particular, when adding protocol conformance to a model, prefer adding a separate extension for the protocol methods. This keeps the related methods grouped together with the protocol and can simplify instructions to add a protocol to a class with its associated methods.

Preferred:

```
class MyViewController: UIViewController {
  // class stuff here
}

// MARK: - UITableViewDataSource
extension MyViewController: UITableViewDataSource {
  // table view data source methods
}

// MARK: - UIScrollViewDelegate
extension MyViewController: UIScrollViewDelegate {
  // scroll view delegate methods
}
```

Not Preferred:

```
class MyViewController: UIViewController, UITableViewDataSource,
UIScrollViewDelegate {
  // all methods
}
```

For UIKit view controllers, consider grouping lifecycle, custom accessors, and IBAction in separate class extensions.

2.2 Unused Code

Unused (dead) code, including Xcode template code and placeholder comments should be removed.

Implementation simply calls the superclass should also be removed. This includes any empty/unused UIApplicationDelegate methods.

Preferred:

```
override func tableView(_ tableView: UITableView, numberOfRowsInSection
section: Int) -> Int {
  return Database.contacts.count
}
```

Not Preferred:

```
override func didReceiveMemoryWarning() {
  super.didReceiveMemoryWarning()
  // Dispose of any resources that can be recreated.
}

override func numberOfSections(in tableView: UITableView) -> Int {
  // #warning Incomplete implementation, return the number of sections
  return 1
}
```

```
override func tableView(_ tableView: UITableView, numberOfRowsInSection
section: Int) -> Int {
  // #warning Incomplete implementation, return the number of rows
  return Database.contacts.count
}
```

2.3 Minimal Imports

Import only the modules a source file requires. For example, don't import UIKit when importing Foundation will suffice. Likewise, don't import Foundation if you must import UIKit.

Preferred:

```
import UIKit
var view: UIView
var deviceModels: [String]
```

Preferred:

```
import Foundation
var deviceModels: [String]
```

Not Preferred:

```
import UIKit
import Foundation
var view: UIView
var deviceModels: [String]
```

Not Preferred:

```
import UIKit
var deviceModels: [String]
```

3. Spacing

Method braces and other braces (if/else/switch/while etc.) always open on the same line as the statement but close on a new line.

Preferred:

```
if user.isHappy {
  // Do something
} else {
  // Do something else
}
```

Not Preferred:

```
if user.isHappy
{
  // Do something
}
else {
  // Do something else
}
```

- There should be exactly one blank line between methods to aid in visual clarity and organisation. Whitespace within methods should separate functionality, but having too many sections in a method often means you should refactor into several methods.
- There should be no blank lines after an opening brace or before a closing brace.
- Colons always have no space on the left and one space on the right. Exceptions are the ternary operator ? :, empty dictionary [:] and #selector syntax addTarget(_:action:).

Preferred:

```
class TestDatabase: Database {
  var data: [String: CGFloat] = ["A": 1.2, "B": 3.2]
}
```

Not Preferred:

```
class TestDatabase : Database {
  var data :[String:CGFloat] = ["A" : 1.2, "B":3.2]
}
```

4. Comments

Use comments to explain why a particular piece of code does something. Comments must be kept up-to-date or deleted.

Example :

```
/*
    @methodName     : viewDidLoad
    @description    : viewDidLoad - Life cycle
    @param          : None
    @result         : None
    */
```

5. Class and Structure

5.1 Use of Self

For conciseness, avoid using self since Swift does not require it to access an object's properties or invoke its methods.

Use self only when required by the compiler (in @escaping closures, or in initialisers to disambiguate properties from arguments). In other words, if it compiles without self then omit it.

5.2 Computed Properties

For conciseness, if a computed property is read-only, omit the get clause. The get clause is required only when a set clause is provided.

Preferred:

```
var diameter: Double {
  return radius * 2
}
```

Not Preferred:

```
var diameter: Double {
  get {
    return radius * 2
  }
}
```

6. Function Declarations

Keep short function declarations on one line including the opening brace:

```
func reticulateSplines(spline: [Double]) -> Bool {
  // reticulate code goes here
}
```

For functions with long signatures, put each parameter on a new line and add an extra indent on subsequent lines:

```
func reticulateSplines(
  spline: [Double],
  adjustmentFactor: Double,
  translateConstant: Int, comment: String
) -> Bool {
  // reticulate code goes here
}
```

Don't use (Void) to represent the lack of an input; simply use (). Use Void instead of () for closure and function outputs.

Preferred:

```
func updateConstraints() -> Void {
  // magic happens here
}

typealias CompletionHandler = (result) -> Void
```

Not Preferred:

```
func updateConstraints() -> () {
  // magic happens here
}

typealias CompletionHandler = (result) -> ()
```

7. Function Calls

Mirror the style of function declarations at call sites. Calls that fit on a single line should be written as such:

```
let success = reticulateSplines(splines)
```

If the call site must be wrapped, put each parameter on a new line, indented one additional level:

```
let success = reticulateSplines(
  spline: splines,
  adjustmentFactor: 1.3,
  translateConstant: 2,
  comment: "normalize the display”)
```

8. Closure Expressions

Use trailing closure syntax only if there's a single closure expression parameter at the end of the argument list. Give the closure parameters descriptive names.

Preferred:

```
UIView.animate(withDuration: 1.0) {
  self.myView.alpha = 0
}

UIView.animate(withDuration: 1.0, animations: {
  self.myView.alpha = 0
}, completion: { finished in
  self.myView.removeFromSuperview()
})
```

Not Preferred:

```
UIView.animate(withDuration: 1.0, animations: {
  self.myView.alpha = 0
})

UIView.animate(withDuration: 1.0, animations: {
  self.myView.alpha = 0
}) { f in
  self.myView.removeFromSuperview()
}
```

For single-expression closures where the context is clear, use implicit returns:

```
attendeeList.sort { a, b in
  a > b
}
```

Chained methods using trailing closures should be clear and easy to read in context. Decisions on spacing, line breaks, and when to use named versus anonymous arguments is left to the discretion of the author. Examples:

```
let value = numbers.map { $0 * 2 }.filter { $0 % 3 == 0 }.index(of: 90)

let value = numbers
  .map {$0 * 2}
  .filter {$0 > 50}
  .map {$0 + 10}
```

9. Types

Always use Swift's native types and expressions when available. Swift offers bridging to Objective-C so you can still use the full set of methods as needed.

Preferred:

```
let width = 120.0                           // Double
let widthString = "\(width)"                    // String
```

Less Preferred:

```
let width = 120.0                           // Double
let widthString = (width as NSNumber).stringValue    // String
```

Not Preferred:

```
let width: NSNumber = 120.0                     // NSNumber
let widthString: NSString = width.stringValue       // NSString
```

In drawing code, use CGFloat if it makes the code more succinct by avoiding too many conversions.

9.1 Constants

Constants are defined using the let keyword and variables with the var keyword.

Always use let instead of var if the value of the variable will not change.

Tip: A good technique is to define everything using let and only change it to var if the compiler complains!

You can define constants on a type rather than on an instance of that type using type properties. To declare a type property as a constant simply use static let. Type properties declared in this way are generally preferred over global constants because they are easier to distinguish from instance properties. Example:

Preferred:

```
enum Math {
  static let e = 2.71828182845904523536028747
  static let root2 = 1.41421356237309504880168872
}

let hypotenuse = side * Math.root2
```

Note: The advantage of using a case-less enumeration is that it can't accidentally be instantiated and works as a pure namespace.

Not Preferred:

```
let e = 2.71828182845904523536028747  // pollutes global namespace
let root2 = 1.41421356237309504880168872

let hypotenuse = side * root2 // what is root2?
```

9.2 Optionals

Declare variables and function return types as optional with ? where a nil value is acceptable.

Use implicitly unwrapped types declared with ! only for instance variables that you know will be initialised later before use, such as subviews that will be set up in viewDidLoad(). Prefer optional binding to implicitly unwrapped optionals in most other cases.

When accessing an optional value, use optional chaining if the value is only accessed once or if there are many optionals in the chain:

```
textContainer?.textLabel?.setNeedsDisplay()
```

Use optional binding when it's more convenient to unwrap once and perform multiple operations:

```
if let textContainer = textContainer {
  // do many things with textContainer
}
```

When naming optional variables and properties, avoid naming them like optionalString or maybeView since their optional-ness is already in the type declaration.

For optional binding, shadow the original name whenever possible rather than using names like unwrappedView or actualLabel.

Preferred:

```
var subview: UIView?
var volume: Double?

// later on...
if let subview = subview, let volume = volume {
  // do something with unwrapped subview and volume
}

// another example
UIView.animate(withDuration: 2.0) { [weak self] in
  guard let self = self else { return }
  self.alpha = 1.0
}
```

Not Preferred:

```
var optionalSubview: UIView?
var volume: Double?

if let unwrappedSubview = optionalSubview {
  if let realVolume = volume {
    // do something with unwrappedSubview and realVolume
  }
}
```

```
// another example
UIView.animate(withDuration: 2.0) { [weak self] in
  guard let strongSelf = self else { return }
  strongSelf.alpha = 1.0
}
```

9.3 Lazy Initialisation

Consider using lazy initialisation for finer grained control over object lifetime. This is especially true for UIViewController that loads views lazily. You can either use a closure that is immediately called { }() or call a private factory method.

Example:

```
lazy var locationManager = makeLocationManager()

private func makeLocationManager() -> CLLocationManager {
  let manager = CLLocationManager()
  manager.desiredAccuracy = kCLLocationAccuracyBest
  manager.delegate = self
  manager.requestAlwaysAuthorization()
  return manager
}
```

Notes:

- [unowned self] is not required here. A retain cycle is not created.
- Location manager has a side-effect for popping up UI to ask the user for permission so fine grain control makes sense here.

9.4 Type Inference

Prefer compact code and let the compiler infer the type for constants or variables of single instances. Type inference is also appropriate for small, non-empty arrays and dictionaries. When required, specify the specific type such as CGFloat or Int16.

Preferred:

```
let message = "Click the button"
let currentBounds = computeViewBounds()
```

```
var names = ["Mic", "Sam", "Christine"]
let maximumWidth: CGFloat = 106.5
```

Not Preferred:

```
let message: String = "Click the button"
let currentBounds: CGRect = computeViewBounds()
var names = [String]()
```

Type Annotation for Empty Arrays and Dictionaries

For empty arrays and dictionaries, use type annotation. (For an array or dictionary assigned to a large, multi-line literal, use type annotation.)

Preferred:

```
var names: [String] = []
var lookup: [String: Int] = [:]
```

Not Preferred:

```
var names = [String]()
var lookup = [String: Int]()
```

NOTE: Following this guideline means picking descriptive names is even more important than before.

9.5 Syntactic Sugar

Prefer the shortcut versions of type declarations over the full generics syntax.

Preferred:

```
var deviceModels: [String]
var employees: [Int: String]
var faxNumber: Int?
```

Not Preferred:

```
var deviceModels: Array<String>
var employees: Dictionary<Int, String>
var faxNumber: Optional<Int>
```

10.Functions vs Methods

Free functions, which aren't attached to a class or type, should be used sparingly. When possible, prefer to use a method instead of a free function. This aids in readability and discoverability.

Free functions are most appropriate when they aren't associated with any particular type or instance.

Preferred

```
let sorted = items.mergeSorted()  // easily discoverable
rocket.launch()  // acts on the model
```

Not Preferred

```
let sorted = mergeSort(items)  // hard to discover
launch(&rocket)
```

10.1 Free Function Exceptions

```
let tuples = zip(a, b)  // feels natural as a free function (symmetry)
let value = max(x, y, z)  // another free function that feels natural
```

11. Memory Management

Code (even non-production, tutorial demo code) should not create reference cycles. Analyse your object graph and prevent strong cycles with weak and unowned references. Alternatively, use value types (struct, enum) to prevent cycles altogether.

11.1 Extending object lifetime

Extend object lifetime using the [weak self] and guard let self = self else { return } idiom. [weak self] is preferred to [unowned self] where it is not immediately obvious that self outlives the closure. Explicitly extending lifetime is preferred to optional chaining.

Preferred

```
resource.request().onComplete { [weak self] response in
  guard let self = self else {
    return
  }
  let model = self.updateModel(response)
  self.updateUI(model)
}
```

Not Preferred

```
// might crash if self is released before response returns
resource.request().onComplete { [unowned self] response in
  let model = self.updateModel(response)
  self.updateUI(model)
}
```

Not Preferred

```
// deallocate could happen between updating the model and updating UI
resource.request().onComplete { [weak self] response in
  let model = self?.updateModel(response)
  self?.updateUI(model)
}
```

12. Access Control

Full access control annotation in tutorials can distract from the main topic and is not required. Using private and fileprivate appropriately, however, adds clarity and promotes encapsulation. Prefer private to fileprivate; use fileprivate only when the compiler insists.

Only explicitly use open, public, and internal when you require a full access control specification.

Use access control as the leading property specifier. The only things that should come before access control are the static specifier or attributes such as @IBAction, @IBOutlet and @discardableResult.

Preferred:

```
private let message = "Great Scott!"

class TimeMachine {
  private dynamic lazy var fluxCapacitor = FluxCapacitor()
}
```

Not Preferred:

```
fileprivate let message = "Great Scott!"

class TimeMachine {
  lazy dynamic private var fluxCapacitor = FluxCapacitor()
}
```

13. Control Flow

Prefer the for-in style of for loop over the while-condition-increment style.

Preferred:

```
for _ in 0..<3 {
  print("Hello three times")
}

for (index, person) in attendeeList.enumerated() {
  print("\(person) is at position #\(index)")
}

for index in stride(from: 0, to: items.count, by: 2) {
  print(index)
}
```

```
for index in (0...3).reversed() {
  print(index)
}
```

Not Preferred:

```
var i = 0
while i < 3 {
  print("Hello three times")
  i += 1
}


var i = 0
while i < attendeeList.count {
  let person = attendeeList[i]
  print("\(person) is at position #\(i)")
  i += 1
}
```

13.1 Ternary Operator

The Ternary operator, ?: , should only be used when it increases clarity or code neatness. A single condition is usually all that should be evaluated. Evaluating multiple conditions is usually more understandable as an if statement or refactored into instance variables. In general, the best use of the ternary operator is during assignment of a variable and deciding which value to use.

Preferred:

```
let value = 5
result = value != 0 ? x : y

let isHorizontal = true
result = isHorizontal ? x : y
```

Not Preferred:

```
result = a > b ? x = c > d ? c : d : y
```

14. Golden Path

When coding with conditionals, the left-hand margin of the code should be the "golden" or "happy" path. That is, don't nest if statements. Multiple return statements are OK. The guard statement is built for this.

Preferred:

```
func computeFFT(context: Context?, inputData: InputData?) throws ->
Frequencies {

  guard let context = context else {
    throw FFTError.noContext
  }
  guard let inputData = inputData else {
    throw FFTError.noInputData
  }

  // use context and input to compute the frequencies
  return frequencies
}
```

Not Preferred:

```
func computeFFT(context: Context?, inputData: InputData?) throws ->
Frequencies {

  if let context = context {
    if let inputData = inputData {
      // use context and input to compute the frequencies

      return frequencies
    } else {
      throw FFTError.noInputData
    }
  } else {
    throw FFTError.noContext
  }
}
```

When multiple optionals are unwrapped either with guard or if let, minimize nesting by using the compound version when possible. In the compound version, place the guard on its own line, then indent each condition on its own line. The else clause is indented to match the conditions and the code is indented one additional level, as shown below. Example:

Preferred:

```
guard
  let number1 = number1,
  let number2 = number2,
  let number3 = number3
  else {
    fatalError("impossible")
}
// do something with numbers
```

Not Preferred:

```
if let number1 = number1 {
  if let number2 = number2 {
    if let number3 = number3 {
      // do something with numbers
    } else {
      fatalError("impossible")
    }
  } else {
    fatalError("impossible")
  }
} else {
  fatalError("impossible")
}
```

14.1 Failing Guards

Guard statements are required to exit in some way. Generally, this should be simple one line statement such as return, throw, break, continue, and fatalError(). Large code blocks should be avoided. If cleanup code is required for multiple exit points, consider using a defer block to avoid cleanup code duplication.

15. Semicolons

Swift does not require a semicolon after each statement in your code. They are only required if you wish to combine multiple statements on a single line.
Do not write multiple statements on a single line separated with semicolons.

Preferred:

```
let swift = "not a scripting language”
```

Not Preferred:

```
let swift = "not a scripting language”;
```

NOTE: Swift is very different from JavaScript, where omitting semicolons is generally considered unsafe

16. Parentheses

Parentheses around conditionals are not required and should be omitted.

Preferred:

```
if name == "Hello" {
  print("World")
}
```

Not Preferred:

```
if (name == "Hello") {
  print("World")
}
```

In larger expressions, optional parentheses can sometimes make code read more clearly.

Preferred:

```
let playerMark = (player == current ? "X" : "O")
```

17. Multi-line String Literals

When building a long string literal, you're encouraged to use the multi-line string literal syntax. Open the literal on the same line as the assignment but do not include text on that line. Indent the text block one additional level.

Preferred:

```
let message = """
  You cannot charge the flux \
  capacitor with a 9V battery.
  You must use a super-charger \
  which costs 10 credits. You currently \
  have \(credits) credits available.
  “””
```

Not Preferred:

```
let message = """You cannot charge the flux \
  capacitor with a 9V battery.
  You must use a super-charger \
  which costs 10 credits. You currently \
  have \(credits) credits available.
  “””
```

Not Preferred:

```
let message = "You cannot charge the flux " +
  "capacitor with a 9V battery.\n" +
  "You must use a super-charger " +
  "which costs 10 credits. You currently " +
  "have \(credits) credits available.”
```

18. Folder structure *

—> (Feature) - Folder Name

—> StoryBoard - Sub Folder Name
—> (Feature).Storyboard

—> WireFrame - Sub Folder
—> (Feature)Wireframe

—> Network - Sub Folder Name
—> (Feature)Manager - All parsing logic
—> (Feature)Service - All API Calls

—> Controllers - Sub Folder Name
—> SubFeature - Sub Folder Name
—> (SubFeature)ViewController
—> (SubFeature)ViewModel

—> Models - Sub Folder Name
—> (Feature)Model

—> Views - Sub Folder Name
—> ViewName - Sub Folder Name (Name should be purpose of view)
—> ViewName.swift
—> ViewName.xib
—> TableviewCells - Sub Folder Name (Name should be type of cell)
—> cellName - Sub Folder Name
—> cellName.swift
—> cellName.xib

19. # Mark Arrangement *

—> IBOUTLETS

—> VARIABLES

—> LIFE CYCLE METHODS

—> PRIVATE METHODS

—> PUBLIC METHODS

—> IBACTION METHODS

—> EXTENSIONS