Code Organization

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

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
```

```
}
```

Since the compiler does not allow you to re-declare protocol conformance in a derived class, it is not always required to replicate the extension groups of the base class. This is especially true if the derived class is a terminal class and a small number of methods are being overridden. When to preserve the extension groups is left to the discretion of the author.

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

Unused Code

Unused (dead) code, including Xcode template code and placeholder comments should be removed. An exception is when your tutorial or book instructs the user to use the commented code.

Aspirational methods not directly associated with the tutorial whose 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
}
```

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]
```

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)
```

Spacing

- Indent using 2 spaces rather than tabs to conserve space and help prevent line wrapping. Be sure to set this preference in Xcode and in the Project settings as shown below:

- 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.
- Tip: You can re-indent by selecting some code (or **Command-A** to select all) and then **Control-I** (or **Editor ▸ Structure ▸ Re-Indent** in the menu). Some of the Xcode template code will have 4-space tabs hard coded, so this is a good way to fix that.

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 organization. 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]
}
```

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()
}
```

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.

Lazy Initialization

Consider using lazy initialization 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.

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)
}
```

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()
}
```

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")
```