TERMS OF REFERENCE

Date: 15 Jan 2018
Rev: 1.0

1. Project Organization Structure	
Project Name	Livingworld Mobile Apps
Project Sponsor	Fanny Verona
Project Leader	Theresia Ateng
Project Manager	Lindung Manik (ICG)
Team Members	Yazied Dhiyauddien (ICG) Hatta Palino (ICG) Annas Fanani (ICG) Aan Isnaini (ICG)
Begin Date	1 Dec 2017
Estimated End Date	30 April 2018

2. Project Specifications	
Project Background	The mobile apps will be integrated with Indiepay wallet & e-cash as SOF to enable payment around Livingworld Mall area & automatic points conversion. This is part of hypelocalization strategy.
Goals & Objectives	To serve as use case for Indiepay/e-cash & to increase number of transation
Project Scope	Phase 1: Android & Backend Integration Phase 2: iOS **Inclusions**: CMS, bug fixing, documentation, handover, technical training **Exclusions**: -
Project Dependencies	The app project need integration with: • Livingworld member database • Indiepay & e-cash • Parking system
Project Deliverables (This can be listed as short/medium/long term with timelines)	• Source code application (Android & iOS) • User Manual

Resources	**Internal**: Indiepay API, e-cash API, integration PIC from DAM **External**: Indo Guna Cipta (Vendor)

3. Technical Requirements (iOS)

A. Correctness

Strive to make your code compile without warnings. This rule informs many style decisions such as using #selector types instead of string literals.

B. Naming

Descriptive and consistent naming makes software easier to read and understand. Use the Swift naming conventions described in the API Design Guidelines. Some key takeaways include:

- striving for clarity at the call site
- prioritizing clarity over brevity
- using camel case (not snake case)
- using uppercase for types (and protocols), lowercase for everything else
- including all needed words while omitting needless words
- using names based on roles, not types
- sometimes compensating for weak type information
- striving for fluent usage
- beginning factory methods with make
- naming methods for their side effects
  - verb methods follow the -ed, -ing rule for the non-mutating version
  - noun methods follow the formX rule for the mutating version
  - boolean types should read like assertions
  - protocols that describe what something is should read as nouns
  - protocols that describe a capability should end in -able or -ible
- using terms that don't surprise experts or confuse beginners
- generally avoiding abbreviations
- using precedent for names
- preferring methods and properties to free functions
- casing acronyms and initialisms uniformly up or down
- giving the same base name to methods that share the same meaning
- avoiding overloads on return type
- choosing good parameter names that serve as documentation
- labeling closure and tuple parameters
- taking advantage of default parameters

i. Prose

When referring to methods in prose, being unambiguous is critical. To refer to a method name, use the simplest form possible.

1. Write the method name with no parameters. **Example:** Next, you need to call the method `addTarget`
2. Write the method name with argument labels. **Example:** Next, you need to call the method `addTarget(_:action:)`.
3. Write the full method name with argument labels and types. **Example:** Next, you need to call the method `addTarget(_: Any?, action: Selector?)`.

For the above example using `UIGestureRecognizer`, 1 is unambiguous and preferred.

Pro Tip: You can use Xcode's jump bar to lookup methods with argument labels.

ii. Delegates

When creating custom delegate methods, use ClassNameDelegate naming convention (eg: UITableViewDelegate)

Example on MyClass:

```
// PROTOCOL DELEGATE
protocol MyClassDelegate: NSObjectProtocol {
  func show()
  func didSelectObject(with strTitle: String)
}

// MY CLASS
class MyClass: UIView {
  weak var delegate: MyClassDelegate!
}
```

Example on OtherClass:

```
// OTHER CLASS
class OtherClass: UIViewController {
  @IBOutlet var myClass: MyClass!

  // MARK: - LIFECYCLE
  override func viewDidLoad() {
    super.viewDidLoad()
    self.myClass.delegate = self
  }
}

// MARK: - IMPLEMENT DELEGATE AS EXTENSION
extension OtherClass: MyClassDelegate {
  func show() {}

  func didSelectObject(with strTitle: String) {}
}
```

iii. 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)
```

iv. Generics

Generic type parameters should be descriptive, upper camel case names. When a type name doesn't have a meaningful relationship or role, use a traditional single uppercase letter such as T, U, or V.

Preferred:

```
struct          Stack<Element>          {          ...          }
func   write<Target:   OutputStream>(to   target:   inout   Target)
func swap<T>(_ a: inout T, _ b: inout T)
```

Not Preferred:

```
struct Stack<T> { ... }
func write<target: OutputStream>(to target: inout target)
func swap<Thing>(_ a: inout Thing, _ b: inout Thing)
```

v. Class Prefixes

Swift types are automatically namespaced by the module that contains them and you should not add a class prefix such as RW. If two names from different modules collide you can disambiguate by prefixing the type name with the module name. However, only specify the module name when there is possibility for confusion which should be rare.

```
import SomeModule
let myClass = MyModule.UsefulClass()
```

vi. Language

Use US English spelling to match Apple's API.

Preferred:

```
let color = "red"
```

Not Preferred:

```
let colour = "red"
```

C. 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.

i. 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.

ii. 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:

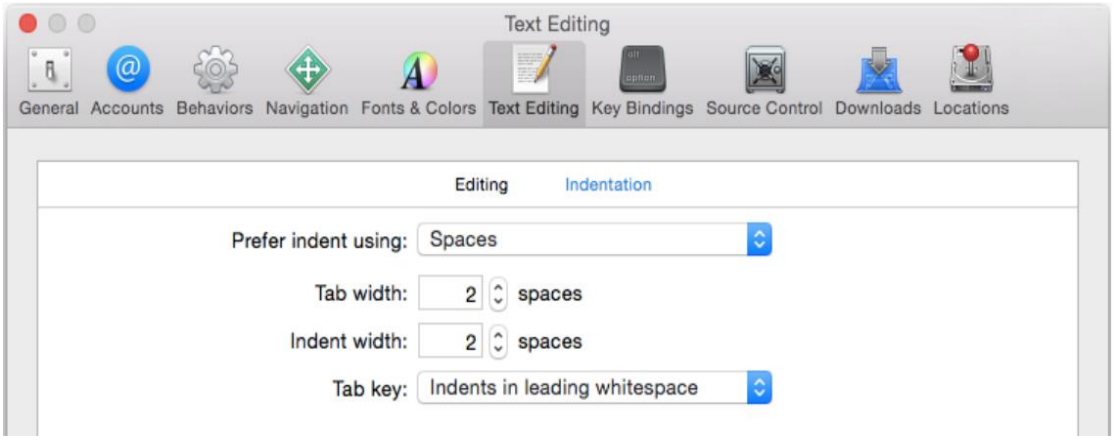
```
override func didReceiveMemoryWarning() {
  super.didReceiveMemoryWarning()
  // Dispose of any resources that can be recreated.
}

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
}
```

iii. Minimal Imports

Keep imports minimal. For example, don't import `UIKit` when importing `Foundation` will suffice.

D. 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 ⌘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.

Colons always have no space on the left and one space on the right. Exceptions are the ternary operator `? :`, empty dictionary `[:]` and `#selector` syntax for unnamed parameters `(_:)`.

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]
}
```

Long lines should be wrapped at around 70 characters. A hard limit is intentionally not specified.

Avoid trailing whitespaces at the ends of lines.

Add a single newline character at the end of each file.

E. Comments

When they are needed, use comments to explain why a particular piece of code does something. Comments must be kept up-to-date or deleted.

Avoid block comments inline with code, as the code should be as self-documenting as possible. *Exception: This does not apply to those comments used to generate documentation.*

F. Classes and Structures

Which one to use?

Remember, structs have value semantics. Use structs for things that do not have an identity. An array that contains [a, b, c] is really the same as another array that contains [a, b, c] and they are completely interchangeable. It doesn't matter whether you use the first array or the second, because they represent the exact same thing. That's why arrays are structs.

Classes have reference semantics. Use classes for things that do have an identity or a specific life cycle. You would model a person as a class because two person objects are two different things. Just because two people have the same name and birthdate, doesn't mean they are the same person. But the person's birthdate would be a struct because a date of 3 March 1950 is the same as any other date object for 3 March 1950. The date itself doesn't have an identity.

Sometimes, things should be structs but need to conform to `AnyObject` or are historically modeled as classes already (`NSDate`, `NSSet`). Try to follow these guidelines as closely as possible.

Example definition

Here's an example of a well-styled class definition:

```
class Circle: Shape {
```

```
  var x: Int, y: Int
  var radius: Double
  var diameter: Double {
    get {
      return radius * 2
    }
    set {
      radius = newValue / 2
    }
  }

  init(x: Int, y: Int, radius: Double) {
    self.x = x
    self.y = y
    self.radius = radius
  }

  convenience init(x: Int, y: Int, diameter: Double) {
    self.init(x: x, y: y, radius: diameter / 2)
  }

  override func area() -> Double {
    return Double.pi * radius * radius
  }
}

extension Circle: CustomStringConvertible {
  var description: String {
    return "center = \(centerString) area = \(area())"
  }
  private var centerString: String {
    return "(\(x),\(y))"
  }
}
```

The example above demonstrates the following style guidelines:

- Specify types for properties, variables, constants, argument declarations and other statements with a space after the colon but not before, e.g. `x: Int`, and `Circle: Shape`.
- Define multiple variables and structures on a single line if they share a common purpose / context.
- Indent getter and setter definitions and property observers.
- Don't add modifiers such as `internal` when they're already the default. Similarly, don't repeat the access modifier when overriding a method.
- Organize extra functionality (e.g. printing) in extensions.
- Hide non-shared, implementation details such as `centerString` inside the extension using `private` access control.

i. 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

initializers to disambiguate properties from arguments). In other words, if it compiles without `self` then omit it.

ii. 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
  }
}
```

iii. Final

Marking classes or members as final in tutorials can distract from the main topic and is not required. Nevertheless, use of final can sometimes clarify your intent and is worth the cost. In the below example, Box has a particular purpose and customization in a derived class is not intended. Marking it final makes that clear.

```
// Turn any generic type into a reference type using this Box
class.
final class Box<T> {
  let value: T
  init(_ value: T) {
    self.value = value
  }
}
```

G. 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, add line breaks at appropriate points and add an extra indent on subsequent lines:

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

H. 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()
}
```

I. Types

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

Preferred:

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

Not Preferred:

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

i. 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
```

Not Preferred:

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

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

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

ii. Static Methods and Variable Type Properties

Static methods and type properties work similarly to global functions and global variables and should be used sparingly. They are useful when functionality is scoped to a particular type or when Objective-C interoperability is required.

iii. 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 initialized later before use, such as subviews that will be set up in viewDidLoad.

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

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

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

```
if let textContainer = self.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 when appropriate 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
}
```

Not Preferred:

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

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

iv. Lazy Initialization

Consider using lazy initialization for finer grain 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: CLLocationManager =
self.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.

v. Type Inference

Prefer explicit the type for constants or variables of single instances. Type explicit is also appropriate for small (non-empty) arrays and dictionaries. This is to enhance build/compile time for the project.

Preferred:

```
let width: Double = 120.0
let widthString: String = (width as NSNumber).stringValue
let names: [String] = [”Felicity”, ”Scarlet”]
```

Not Preferred:

```
let width = 120.0
let widthString = (width as NSNumber).stringValue
let names = [”Felicity”, ”Scarlet”]
```

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.

vi. 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>
```

J. 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)
```

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
```

K. Memory Management

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

i. Extending Object Lifetime

Extend object lifetime using the `[weak self]` and `guard let strongSelf = 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 unwrapping.

Preferred:

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

L. 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` when possible. Using extensions may require you to use `fileprivate`.

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 {
  fileprivate dynamic lazy var fluxCapacitor = FluxCapacitor()
}
```

Not Preferred:

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

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

M. 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
}
```

N. 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. 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")
}
```

i. 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.

O. 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";
```

P. Parentheses

Parentheses around conditionals are not required and should be omitted.

Preferred:

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

Not Preferred:

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

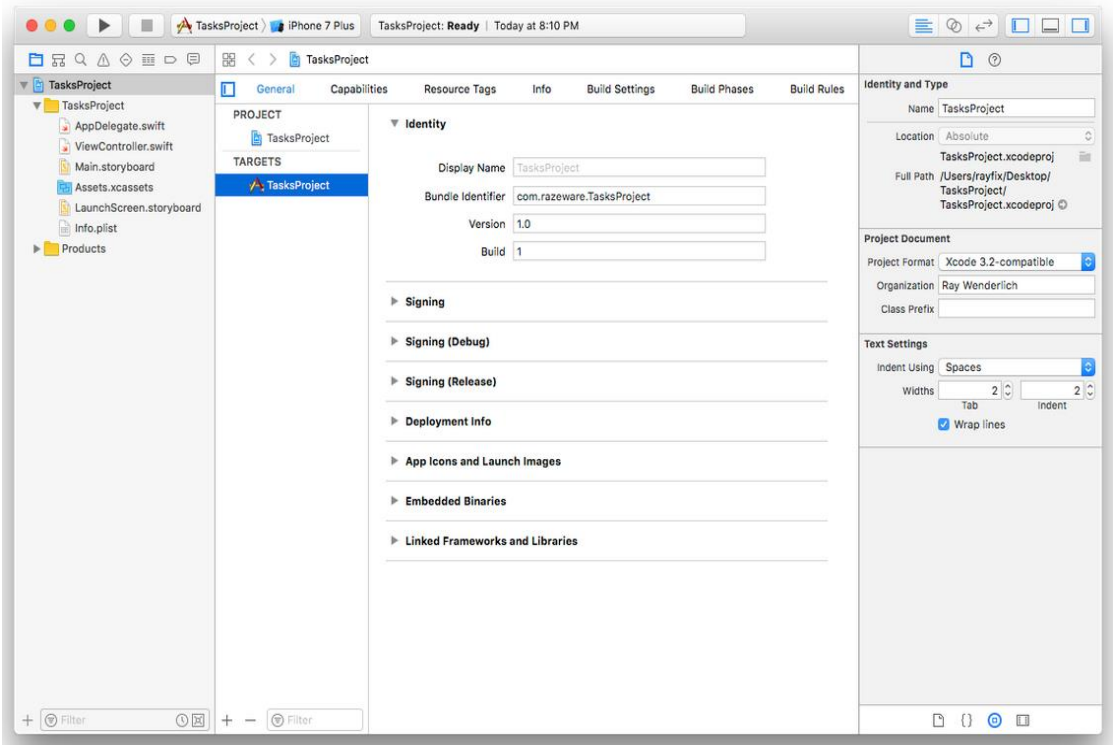
In larger expressions, optional parentheses can sometimes make code read more clearly.

Preferred:

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

Q. Organization and Bundle Identifier

Where an Xcode project is involved, the organization should be set to PT DAM and the Bundle Identifier set to com.ptdam.projectname.dev where projectname is the name of the project.



R. Copyright Statement

The following copyright statement should be included at the top of every source file:

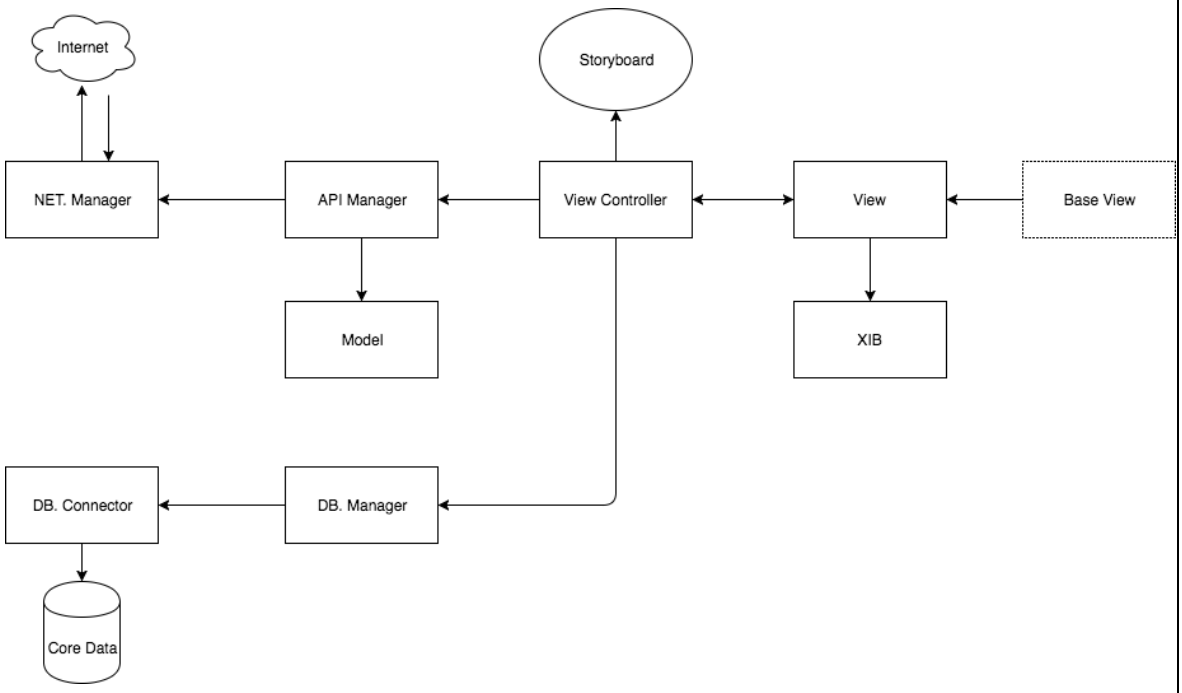
```
//
//  ___FILENAME___
//  ___PROJECTNAME___
//
//  Created by ___FULLUSERNAME___ on ___DATE___.
//  Copyright (c) ___YEAR___ PT. DAM. All rights reserved.
//
```

S. Directory Structure

```
├── Cartfile
├── Carthage
├── README.md
├── myProject
│   ├── AppDelegate.swift
│   ├── Assets.xcassets
│   ├── Info.plist
│   ├── Localize
│   │   ├── Base.lproj
│   │   │   └── Localizable.strings
│   │   ├── Localization.swift
│   │   ├── en.lproj
│   │   │   └── Localizable.strings
│   │   └── id.lproj
│   │       └── Localizable.strings
│   ├── Model
│   │   ├── API
│   │   │   ├── ECAPI.swift
│   │   │   └── ECAPIClient.swift
│   │   ├── Core\ Data
│   │   │   ├── ECCoreData.swift
│   │   │   ├── ECCoreDataHandler.swift
│   │   │   ├── ECFavoriteTransactionGroup.swift
│   │   │   └── myProject.xcdatamodeld
│   │   │       └── myProject.xcdatamodel
│   │   │           └── contents
│   │   ├── ECCommon.swift
│   │   ├── ECField.swift
│   │   ├── ECHistory.swift
│   │   ├── ECTransaction.swift
│   │   └── ECUser.swift
│   ├── Shared
│   │   ├── Classes
│   │   │   ├── Security
│   │   │   │   └── ECSecurity.swift
│   │   │   └── View
│   │   │       ├── BaseView.swift
│   │   │       ├── Cell
│   │   │       │   ├── ECDefaultCell.swift
│   │   │       │   ├── ECDefaultCell.xib
│   │   │       │   ├── ECTransactionCell.swift
│   │   │       │   └── ECTransactionCell.xib
│   │   │       ├── Field
│   │   │       │   ├── ECFieldCell.swift
│   │   │       │   ├── ECFieldCell.xib
│   │   │       │   ├── ECFieldPIN.swift
│   │   │       │   └── ECFieldPIN.xib
│   │   │       ├── Header
│   │   │       │   ├── ECHeader.swift
│   │   │       │   └── ECHeader.xib
│   │   │       ├── Keyboard
│   │   │       │   ├── ECNumericKeyboard.swift
│   │   │       │   └── ECNumericKeyboard.xib
│   │   │       └── Modal
│   │   │           ├── ECModal.swift
│   │   │           ├── ECModalSMS.swift
│   │   │           └── ECModalSMS.xib
```

```
        ├── Constants.swift
        ├── Extension
        │   ├── ECNotification.swift
        │   ├── ECString.swift
        │   ├── ECUIColor.swift
        │   ├── ECUILabel.swift
        │   ├── ECUIView.swift
        │   └── ECViewController.swift
        ├── Library
        └── Stylesheet.swift
    ├── Story\ Board
    │   ├── Base.lproj
    │   │   ├── LaunchScreen.storyboard
    │   │   └── Main.storyboard
    │   ├── LoginRegister.storyboard
    │   ├── Transaction.storyboard
    │   ├── en.lproj
    │   └── id.lproj
    ├── Suporting\ File
    │   ├── Fonts
    ├── View\ Controller
    │   ├── Home
    │   │   ├── ECHomeMenuCell.swift
    │   │   ├── ECHomeMenuCell.xib
    │   │   ├── ECHomeVC.swift
    │   │   ├── ECHomeView.swift
    │   │   └── ECHomeView.xib
    │   ├── LoginRegister
    │   │   ├── Login
    │   │   │   ├── ECLoginVC.swift
    │   │   │   ├── ECLoginView.swift
    │   │   │   └── ECLoginView.xib
    │   └── dev-info.plist
└── myProject.xcodeproj
```

T. Architecture Design Pattern



U. UI and Layout

Create UI for per screen using xib and AutoLayout for easy maintenance in the future, make sure compile without autolayout issue(warning and error), all constraint issue must resolve.

Avoid build autolayout constraint using code, when necessary we prefer using **EasyPeasy Library**(https://github.com/nakiostudio/EasyPeasy).

When updating view autolayout contraints on run time, use constraint outlets and update the constant or priority.

Use Storyboard to define ViewController flow and separate storyboard based on UX story flow (eg: Transaction.storyboard, LoginRegister.storyboard).

Use **Assets.xcassets** to store assets image or icon. Create folder for each category to make assets maintenance easier and use assets naming meaningful yet not to long.

V. Third-Party Library

Use Carthage Package Manager to embed third-party library, avoid Cocoapods if possible (there's Carthage method installation).

W.References

- The Swift API Design Guidelines
  https://swift.org/documentation/api-design-guidelines/

- The Swift Programming Language
  https://developer.apple.com/library/prerelease/ios/documentation/swift/conceptual/swift_programming_language/index.html

- Using Swift with Cocoa and Objective-C
  https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/BuildingCocoaApps/index.html

- Swift Standard Library Reference
  https://developer.apple.com/library/prerelease/ios/documentation/General/Reference/SwiftStandardLibraryReference/index.html

4. Technical Requirements (Android)

Develop Using Android Studio

Build system

Your default option should be Gradle using the Android Gradle plugin.

It is important that your application's build process is defined by your Gradle files, rather than being reliant on IDE specific configurations. This allows for consistent builds between tools and better support for continuous integration systems.

Project structure

Although Gradle offers a large degree of flexibility in your project structure, unless you have a compelling reason to do otherwise, you should accept its default structure as this simplify your build scripts.

Gradle configuration

General structure. Follow Google's guide on Gradle for Android.

minSdkVersion: 21 We recommend to have a look at the Android version usage chart before defining the minimum API required. Remember that the statistics given are global statistics and may differ when targeting a specific regional/demographic market. It is worth mentioning that some material design features are only available on Android 5.0 (API level 21) and above. And also, from API 21, the multidex support library is not needed anymore.

Small tasks. Instead of (shell, Python, Perl, etc) scripts, you can make tasks in Gradle. Just follow Gradle's documentation for more details. Google also provides some helpful Gradle recipes, specific to Android.

Passwords. In your app's `build.gradle` you will need to define the `signingConfigs` for the release build. Here is what you should avoid:
Don't do this. This would appear in the version control system.

```
signingConfigs {
    release {
        // DON'T DO THIS!!
        storeFile file("myapp.keystore")
        storePassword "password123"
        keyAlias "thekey"
```

```
        keyPassword "password789"
    }
}
```

Instead, make a `gradle.properties` file which should *not* be added to the version control system:

```
KEYSTORE_PASSWORD=password123
KEY_PASSWORD=password789
```

That file is automatically imported by Gradle, so you can use it in `build.gradle` as such:

```
signingConfigs {
    release {
        try {
            storeFile file("myapp.keystore")
            storePassword KEYSTORE_PASSWORD
            keyAlias "thekey"
            keyPassword KEY_PASSWORD
        }
        catch (ex) {
            throw new InvalidUserDataException("You should define
KEYSTORE_PASSWORD and KEY_PASSWORD in gradle.properties.")
        }
    }
}
```

Prefer Maven dependency resolution to importing jar files. If you explicitly include jar files in your project, they will be a specific frozen version, such as `2.1.1`. Downloading jars and handling updates is cumbersome and is a problem that Maven already solves properly. Where possible, you should attempt to use Maven to resolve your dependencies, for example:

```
dependencies {
    compile 'com.squareup.okhttp:okhttp3:3.8.0'
}
```

Avoid Maven dynamic dependency resolution Avoid the use of dynamic dependency versions, such as `2.1.+` as this may result in different and unstable builds or subtle, untracked differences in behavior between builds. The use of static versions such as `2.1.1` helps create a more stable, predictable and repeatable development environment.

Use different package name for non-release builds Use `applicationIdSuffix` for *debug* build type to be able to install both *debug* and *release* apk on the same device (do this also for custom build types, if you need any). This will be especially valuable after your app has been published.

```
android {
    buildTypes {
        debug {
            applicationIdSuffix '.debug'
```

```
            versionNameSuffix '-DEBUG'
        }

        release {
            // ...
        }
    }
}
```

Use different icons to distinguish the builds installed on a device—for example with different colors or an overlaid "debug" label. Gradle makes this very easy: with default project structure, simply put *debug* icon in `app/src/debug/res` and *release*icon in `app/src/release/res`. You could also change app name per build type, as well as `versionName` (as in the above example).

1. Project guidelines

1.1 Project structure

New projects should follow the Android Gradle project structure that is defined on the Android Gradle plugin user guide.

1.2 File naming

1.2.1 Class files

Class names are written in UpperCamelCase.

For classes that extend an Android component, the name of the class should end with the name of the component; for example: `SignInActivity`, `SignInFragment`, `ImageUploaderService`, `ChangePasswordDialog`.

1.2.2 Resources files

Resources file names are written in **lowercase_underscore**.

1.2.2.1 Drawable files

Naming conventions for drawables:

Asset Type	Prefix	Example
Action bar	`ab_`	`ab_stacked.9.png`
Button	`btn_`	`btn_send_pressed.9.png`
Dialog	`dialog_`	`dialog_top.9.png`
Divider	`divider_`	`divider_horizontal.9.png`
Icon	`ic_`	`ic_star.png`
Menu	`menu_`	`menu_submenu_bg.9.png`
Notification	`notification_`	`notification_bg.9.png`
Tabs	`tab_`	`tab_pressed.9.png`

Naming conventions for icons (taken from Android iconography guidelines):

Asset Type	Prefix	Example
Icons	`ic_`	`ic_star.png`
Launcher icons	`ic_launcher`	`ic_launcher_calendar.png`
Menu icons and Action Bar icons	`ic_menu`	`ic_menu_archive.png`
Status bar icons	`ic_stat_notify`	`ic_stat_notify_msg.png`
Tab icons	`ic_tab`	`ic_tab_recent.png`
Dialog icons	`ic_dialog`	`ic_dialog_info.png`

Naming conventions for selector states:

State	Suffix	Example
Normal	`_normal`	`btn_order_normal.9.png`
Pressed	`_pressed`	`btn_order_pressed.9.png`
Focused	`_focused`	`btn_order_focused.9.png`
Disabled	`_disabled`	`btn_order_disabled.9.png`
Selected	`_selected`	`btn_order_selected.9.png`

1.2.2.2 Layout files

Layout files should match the name of the Android components that they are intended for but moving the top-level component name to the beginning. For example, if we are creating a layout for the `SignInActivity`, the name of the layout file should be `activity_sign_in.xml`.

Component	Class Name	Layout Name
Activity	`UserProfileActivity`	`activity_user_profile.xml`
Fragment	`SignUpFragment`	`fragment_sign_up.xml`
Dialog	`ChangePasswordDialog`	`dialog_change_password.xml`
AdapterView item	`---`	`item_person.xml`
Partial layout	`---`	`partial_stats_bar.xml`

A slightly different case is when we are creating a layout that is going to be inflated by an `Adapter`, e.g to populate a `ListView`. In this case, the name of the layout should start with `item_`.
Note that there are cases where these rules will not be possible to apply. For example, when creating layout files that are intended to be part of other layouts. In this case you should use the prefix `partial_`.

1.2.2.3 Menu files

Similar to layout files, menu files should match the name of the component. For example, if we are defining a menu file that is going to be used in the `UserActivity`, then the name of the file should be `activity_user.xml`
A good practice is to not include the word `menu` as part of the name because these files are already located in the `menu` directory.

1.2.2.4 Values files

Resource files in the values folder should be **plural**,
e.g. `strings.xml`, `styles.xml`, `colors.xml`, `dimens.xml`, `attrs.xml`

1.2.2.5 Uses Styles

Almost every project needs to properly use styles, because it is very common to have a repeated appearance for a view. At least you should have a common style for most text content in the application, for example:

```
<style name="ContentText">
    <item name="android:textSize">@dimen/font_normal</item>
    <item name="android:textColor">@color/basic_black</item>
</style>
```

Applied to TextViews:

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/price"
    style="@style/ContentText"
    />
```

1.2.2.6 Split a large style file into other files.

You don't need to have a single `styles.xml` file. Android SDK supports other files out of the box, there is nothing magical about the name `styles`, what matters are the XML tags `<style>` inside the file. Hence you can have
files `styles.xml`, `styles_home.xml`, `styles_item_details.xml`, `styles_forms.xml`. Unlike resource directory names which carry some meaning for the build system, filenames in `res/values` can be arbitrary.

`colors.xml` is a color palette. There should be nothing else in your `colors.xml` than just a mapping from a color name to an RGBA value. Do not use it to define RGBA values for different types of buttons.

Don't do this:

```
<resources>
    <color name="button_foreground">#FFFFFF</color>
    <color name="button_background">#2A91BD</color>
    <color name="comment_background_inactive">#5F5F5F</color>
    <color name="comment_background_active">#939393</color>
    <color name="comment_foreground">#FFFFFF</color>
    <color name="comment_foreground_important">#FF9D2F</color>
    ...
    <color name="comment_shadow">#323232</color>
```

You can easily start repeating RGBA values in this format, and that makes it complicated to change a basic color if needed. Also, those definitions are related to some context, like "button" or "comment", and should live in a button style, not in `colors.xml`.
Instead, do this:

```
<resources>

    <!-- grayscale -->
    <color name="white"     >#FFFFFF</color>
    <color name="gray_light">#DBDBDB</color>
    <color name="gray"      >#939393</color>
    <color name="gray_dark" >#5F5F5F</color>
    <color name="black"     >#323232</color>

    <!-- basic colors -->
    <color name="green">#27D34D</color>
    <color name="blue">#2A91BD</color>
    <color name="orange">#FF9D2F</color>
    <color name="red">#FF432F</color>

</resources>
```

Ask for this palette from the designer of the application. The names do not need to be color names as "green", "blue", etc. Names such as "brand_primary", "brand_secondary", "brand_negative" are totally acceptable as well. Formatting colors as such will make it easy to change or refactor colors, and also will make it explicit how many different colors are being used. Normally for a aesthetic UI, it is important to reduce the variety of colors being used.

Treat dimens.xml like colors.xml. You should also define a "palette" of typical spacing and font sizes, for basically the same purposes as for colors. A good example of a dimens file:

```
<resources>

    <!-- font sizes -->
    <dimen name="font_larger">22sp</dimen>
    <dimen name="font_large">18sp</dimen>
    <dimen name="font_normal">15sp</dimen>
    <dimen name="font_small">12sp</dimen>
```

```
    <!-- typical spacing between two views -->
    <dimen name="spacing_huge">40dp</dimen>
    <dimen name="spacing_large">24dp</dimen>
    <dimen name="spacing_normal">14dp</dimen>
    <dimen name="spacing_small">10dp</dimen>
    <dimen name="spacing_tiny">4dp</dimen>

    <!-- typical sizes of views -->
    <dimen name="button_height_tall">60dp</dimen>
    <dimen name="button_height_normal">40dp</dimen>
    <dimen name="button_height_short">32dp</dimen>

</resources>
```

You should use the `spacing_****` dimensions for layouting, in margins and paddings, instead of hard-coded values, much like strings are normally treated. This will give a consistent look-and-feel, while making it easier to organize and change styles and layouts.

strings.xml

Name your strings with keys that resemble namespaces, and don't be afraid of repeating a value for two or more keys. Languages are complex, so namespaces are necessary to bring context and break ambiguity.

Bad

```
<string name="network_error">Network error</string>
<string name="call_failed">Call failed</string>
<string name="map_failed">Map loading failed</string>
```

Good

```
<string name="error_message_network">Network error</string>
<string name="error_message_call">Call failed</string>
<string name="error_message_map">Map loading failed</string>
```

Don't write string values in all uppercase. Stick to normal text conventions (e.g., capitalize first character). If you need to display the string in all caps, then do that using for instance the attribute `textAllCaps` on a TextView.

Bad

```
<string name="error_message_call">CALL FAILED</string>
```

Good

```
<string name="error_message_call">Call failed</string>
```

Avoid a deep hierarchy of views. Sometimes you might be tempted to just add yet another LinearLayout, to be able to accomplish an arrangement of views. This kind of situation may occur:

```
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    >

    <RelativeLayout
        ...
        >

        <LinearLayout
            ...
            >

            <LinearLayout
                ...
                >

                <LinearLayout
                    ...
                    >
                </LinearLayout>

            </LinearLayout>

        </LinearLayout>

    </RelativeLayout>

</LinearLayout>
```

Even if you don't witness this explicitly in a layout file, it might end up happening if you are inflating (in Java) views into other views.
A couple of problems may occur. You might experience performance problems, because there is a complex UI tree that the processor needs to handle. Another more serious issue is a possibility of StackOverflowError.
Therefore, try to keep your views hierarchy as flat as possible: learn how to use ConstraintLayout, how to optimize your layouts and to use the `<merge>` tag.

Beware of problems related to WebViews. When you must display a web page, for instance for a news article, avoid doing client-side processing to clean the HTML, rather ask for a "*pure*" HTML from the backend programmers. WebViews can also leak memory when they keep a reference to their Activity, instead of being bound to the ApplicationContext. Avoid using a WebView for simple texts or buttons, prefer the platform's widgets.

2 Code guidelines

2.1 Java language rules

2.1.1 Don't ignore exceptions

You must never do the following:

```
void setServerPort(String value) {
    try {
        serverPort = Integer.parseInt(value);
    } catch (NumberFormatException e) { }
}
```

Acceptable alternatives (in order of preference) are:

Throw the exception up to the caller of your method.

```
void setServerPort(String value) throws NumberFormatException {
    serverPort = Integer.parseInt(value);
}
```

Throw a new exception that's appropriate to your level of abstraction.

```
void setServerPort(String value) throws ConfigurationException {
    try {
        serverPort = Integer.parseInt(value);
    } catch (NumberFormatException e) {
        throw new ConfigurationException("Port " + value + " is not
valid.");
    }
}
```

Handle the error gracefully and substitute an appropriate value in the catch {} block.

```
/** Set port. If value is not a valid number, 80 is substituted. */

void setServerPort(String value) {
    try {
        serverPort = Integer.parseInt(value);
    } catch (NumberFormatException e) {
        serverPort = 80;  // default port for server
    }
}
```

Catch the Exception and throw a new RuntimeException. This is dangerous, so do it only if you are positive that if this error occurs the appropriate thing to do is crash.

```
/** Set port. If value is not a valid number, die. */

void setServerPort(String value) {
    try {
        serverPort = Integer.parseInt(value);
    } catch (NumberFormatException e) {
        throw new RuntimeException("port " + value " is invalid, ", e);
    }
}
```

> **Note** The original exception is passed to the constructor for RuntimeException. If your code must compile under Java 1.3, you must omit the exception that is the cause.

As a last resort, if you are confident that ignoring the exception is appropriate then you may ignore it, but you must also comment why with a good reason:

```
/** If value is not a valid number, original port number is used. */
void setServerPort(String value) {
    try {
        serverPort = Integer.parseInt(value);
    } catch (NumberFormatException e) {
        // Method is documented to just ignore invalid user input.
        // serverPort will just be unchanged.
    }
```

2.1.2 Don't catch generic exception

You should **not** do this:

```
try {
    someComplicatedIOFunction();        // may throw IOException
    someComplicatedParsingFunction();   // may throw ParsingException
    someComplicatedSecurityFunction();  // may throw SecurityException
```

```
    // phew, made it all the way
} catch (Exception e) {                 // I'll just catch all exceptions
    handleError();                      // with one generic handler!
}
```

Alternatives to catching generic Exception:

- Catch each exception separately as separate catch blocks after a single try. This can be awkward but is still preferable to catching all Exceptions. Beware repeating too much code in the catch blocks.
- Refactor your code to have more fine-grained error handling, with multiple try blocks. Split up the IO from the parsing, handle errors separately in each case.
- Rethrow the exception. Many times you don't need to catch the exception at this level anyway, just let the method throw it.

2.1.3 Fully qualify imports

This is bad: `import foo.*;`
This is good: `import foo.Bar;`
See more info here

2.2 Java style rules

2.2.1 Fields definition and naming

Fields should be defined at the **top of the file** and they should follow the naming rules listed below.

- Private, non-static field names start with **m**.
- Private, static field names start with **s**.
- Other fields start with a lower-case letter.
- Static final fields (constants) are ALL_CAPS_WITH_UNDERSCORES.

Example:

```
public class MyClass {
    public static final int SOME_CONSTANT = 42;
    public int publicField;
    private static MyClass sSingleton;
    int mPackagePrivate;
    private int mPrivate;
    protected int mProtected;
}
```

2.2.3 Treat acronyms as words

Good	Bad
XmlHttpRequest	XMLHTTPRequest
getCustomerId	getCustomerID
String url	String URL
long id	long ID

2.2.4 Use spaces for indentation

Use **4 space** indents for blocks:

```
if (x == 1) {
    x++;
}
```

Use **8 space** indents for line wraps:

```
Instrument i =
        someLongExpression(that, wouldNotFit, on, one, line);
```

2.2.5 Use standard brace style

Braces go on the same line as the code before them.

```
class MyClass {
    int func() {
        if (something) {
            // ...
        } else if (somethingElse) {
            // ...
        } else {
            // ...
        }
    }
}
```

Braces around the statements are required unless the condition and the body fit on one line.

If the condition and the body fit on one line and that line is shorter than the max line length, then braces are not required, e.g.

```
if (condition) body();
```

This is **bad**:

```
if (condition)
    body();  // bad!
```

2.2.6 Annotations

2.2.6.1 Annotations practices

According to the Android code style guide, the standard practices for some of the predefined annotations in Java are:

- `@Override`: The @Override annotation **must be used** whenever a method overrides the declaration or implementation from a super-class. For example, if you use the @inheritdocs Javadoc tag, and derive from a class (not an interface), you must also annotate that the method @Overrides the parent class's method.
- `@SuppressWarnings`: The @SuppressWarnings annotation should only be used under circumstances where it is impossible to eliminate a warning. If a warning passes this "impossible to eliminate" test, the @SuppressWarnings annotation must be used, so as to ensure that all warnings reflect actual problems in the code. 2.2.6.2 Annotations style

Classes, Methods and Constructors
When annotations are applied to a class, method, or constructor, they are listed after the documentation block and should appear as **one annotation per line** .

```
/* This is the documentation block about the class */
@AnnotationA
@AnnotationB
public class MyAnnotatedClass { }
```

Fields
Annotations applying to fields should be listed **on the same line**, unless the line reaches the maximum line length.

```
@Nullable @Mock DataManager mDataManager;
```

2.2.7 Limit variable scope

The scope of local variables should be kept to a minimum (Effective Java Item 29). By doing so, you increase the readability and maintainability of your code and reduce the likelihood of error. Each variable should be declared in the innermost block that encloses all uses of the variable.

Local variables should be declared at the point they are first used. Nearly every local variable declaration should contain an initializer. If you don't yet have enough information to initialize a variable sensibly, you should postpone the declaration until you do.

2.2.8 Order import statements

If you are using an IDE such as Android Studio, you don't have to worry about this because your IDE is already obeying these rules. If not, have a look below.

The ordering of import statements is:

1. Android imports
2. Imports from third parties (com, junit, net, org)
3. java and javax
4. Same project imports

To exactly match the IDE settings, the imports should be:

- Alphabetically ordered within each grouping, with capital letters before lower case letters (e.g. Z before a).
- There should be a blank line between each major grouping (android, com, junit, net, org, java, javax).

2.2.9 Logging guidelines

Use the logging methods provided by the `Log` class to print out error messages or other information that may be useful for developers to identify issues:

- `Log.v(String tag, String msg)` (verbose)
- `Log.d(String tag, String msg)` (debug)
- `Log.i(String tag, String msg)` (information)
- `Log.w(String tag, String msg)` (warning)
- `Log.e(String tag, String msg)` (error)

As a general rule, we use the class name as tag and we define it as a `static final` field at the top of the file. For example:

```
public class MyClass {
    private static final String TAG = MyClass.class.getSimpleName();

    public myMethod() {
```

```
        Log.e(TAG, "My error message");
    }
}
```

VERBOSE and DEBUG logs **must** be disabled on release builds. It is also recommended to disable INFORMATION, WARNING and ERROR logs but you may want to keep them enabled if you think they may be useful to identify issues on release builds. If you decide to leave them enabled, you have to make sure that they are not leaking private information such as email addresses, user ids, etc.

To only show logs on debug builds:

```
if (BuildConfig.DEBUG) Log.d(TAG, "The value of x is " + x);
```

2.2.10 Class member ordering

There is no single correct solution for this but using a **logical** and **consistent** order will improve code learnability and readability. It is recommendable to use the following order:

1. Constants
2. Fields
3. Constructors
4. Override methods and callbacks (public or private)
5. Public methods
6. Private methods
7. Inner classes or interfaces

Example:

```
public class MainActivity extends Activity {

    private static final String TAG = MainActivity.class.getSimpleName();

    private String mTitle;
    private TextView mTextViewTitle;

    @Override
    public void onCreate() {
        ...
    }

    public void setTitle(String title) {
        mTitle = title;
    }

    private void setUpView() {
        ...
    }
```

```
    static class AnInnerClass {

    }

}
```

If your class is extending an **Android component** such as an Activity or a Fragment, it is a good practice to order the override methods so that they **match the component's lifecycle**. For example, if you have an Activity that implements `onCreate()`, `onDestroy()`, `onPause()` and `onResume()`, then the correct order is:

```
public class MainActivity extends Activity {

        //Order matches Activity lifecycle
    @Override
    public void onCreate() {}

    @Override
    public void onResume() {}

    @Override
    public void onPause() {}

    @Override
    public void onDestroy() {}

}
```

2.2.11 Parameter ordering in methods

When programming for Android, it is quite common to define methods that take a `Context`. If you are writing a method like this, then the **Context** must be the **first** parameter.
The opposite case are **callback** interfaces that should always be the **last** parameter.

Examples:

```
// Context always goes first
public User loadUser(Context context, int userId);

// Callbacks always go last
public void loadUserAsync(Context context, int userId, UserCallback callback);
```

2.2.13 String constants, naming, and values

Many elements of the Android SDK such as `SharedPreferences`, `Bundle`, or `Intent` use a key-value pair approach so it's very likely that even for a small app you end up having to write a lot of String constants.
When using one of these components, you **must** define the keys as a `static final` fields and they should be prefixed as indicated below.

Element	Field Name Prefix
SharedPreferences	`PREF_`
Bundle	`BUNDLE_`
Fragment Arguments	`ARGUMENT_`
Intent Extra	`EXTRA_`
Intent Action	`ACTION_`

Note that the arguments of a Fragment - `Fragment.getArguments()` - are also a Bundle. However, because this is a quite common use of Bundles, we define a different prefix for them.
Example:

```
// Note the value of the field is the same as the name to avoid duplication issues
static final String PREF_EMAIL = "PREF_EMAIL";
static final String BUNDLE_AGE = "BUNDLE_AGE";
static final String ARGUMENT_USER_ID = "ARGUMENT_USER_ID";

// Intent-related items use full package name as value
static final String EXTRA_SURNAME = "com.myapp.extras.EXTRA_SURNAME";
static final String ACTION_OPEN_USER = "com.myapp.action.ACTION_OPEN_USER";
```

2.2.14 Arguments in Fragments and Activities

When data is passed into an `Activity` or `Fragment` via an `Intent` or a `Bundle`, the keys for the different values **must** follow the rules described in the section above.
When an `Activity` or `Fragment` expects arguments, it should provide a `public static` method that facilitates the creation of the relevant `Intent` or `Fragment`.
In the case of Activities the method is usually called `getStartIntent()`:

```
public static Intent getStartIntent(Context context, User user) {
        Intent intent = new Intent(context, ThisActivity.class);
        intent.putParcelableExtra(EXTRA_USER, user);
        return intent;
}
```

For Fragments it is named `newInstance()` and handles the creation of the Fragment with the right arguments:

```
public static UserFragment newInstance(User user) {
        UserFragment fragment = new UserFragment();
        Bundle args = new Bundle();
        args.putParcelable(ARGUMENT_USER, user);
        fragment.setArguments(args)
        return fragment;
}
```

Note 1: These methods should go at the top of the class before `onCreate()`.
Note 2: If we provide the methods described above, the keys for extras and arguments should be `private` because there is not need for them to be exposed outside the class.

2.2.15 Line length limit

Code lines should not exceed **100 characters**. If the line is longer than this limit there are usually two options to reduce its length:

- Extract a local variable or method (preferable).
- Apply line-wrapping to divide a single line into multiple ones.

There are two **exceptions** where it is possible to have lines longer than 100:

- Lines that are not possible to split, e.g. long URLs in comments.
- `package` and `import` statements.

2.2.15.1 Line-wrapping strategies

There isn't an exact formula that explains how to line-wrap and quite often different solutions are valid. However there are a few rules that can be applied to common cases.

Break at operators When the line is broken at an operator, the break comes **before** the operator. For example:

```
int longName = anotherVeryLongVariable + anEvenLongerOne - thisRidiculousLongOne
        + theFinalOne;
```

Assignment Operator Exception

An exception to the `break at operators` rule is the assignment operator `=`, where the line break should happen **after** the operator.

```
int longName =
        anotherVeryLongVariable + anEvenLongerOne - thisRidiculousLongOne +
theFinalOne;
```

Method chain case

When multiple methods are chained in the same line - for example when using Builders - every call to a method should go in its own line, breaking the line before the `.`

```
Picasso.with(context)
        .load("http://ribot.co.uk/images/sexyjoe.jpg")
        .into(imageView);
```

Long parameters case

When a method has many parameters or its parameters are very long, we should break the line after every comma `,`

```
loadPicture(context,
        "http://ribot.co.uk/images/sexyjoe.jpg",
        mImageViewProfilePicture,
        clickListener,
        "Title of the picture");
```

2.2.16 RxJava chains styling

Rx chains of operators require line-wrapping. Every operator must go in a new line and the line should be broken before the `.`

```
public Observable<Location> syncLocations() {
    return mDatabaseHelper.getAllLocations()
            .concatMap(new Func1<Location, Observable<? extends Location>>() {
                @Override
                 public Observable<? extends Location> call(Location location) {
                     return mRetrofitService.getLocation(location.id);
                 }
            })
            .retry(new Func2<Integer, Throwable, Boolean>() {
                 @Override
                 public Boolean call(Integer numRetries, Throwable throwable) {
                     return throwable instanceof RetrofitError;
                 }
            });
}
```

2.3.1 Use self closing tags

When an XML element doesn't have any contents, you **must** use self closing tags.

This is **good:**

```
<TextView
        android:id="@+id/text_view_profile"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
```

This is **bad** :

```
<!-- Don\'t do this! -->
<TextView
    android:id="@+id/text_view_profile"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" >
</TextView>
```

2.3.2 Resources naming

Resource IDs and names are written in **lowercase_underscore**.

2.3.2.1 ID naming

IDs should be prefixed with the name of the element in lowercase camelcase. For example:

Element	Prefix
TextView	tvVariableName
ImageView	imVariableName
Button	btnVariableName
Menu	menuVariableName

Image view example:

```
<ImageView
    android:id="@+id/imProfile"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

Menu example:

```
<menu>
        <item
        android:id="@+id/menuDone"
        android:title="Done" />
</menu>
```

2.3.2.2 Strings

String names start with a prefix that identifies the section they belong to. For example `registration_email_hint` or `registration_name_hint`. If a string **doesn't belong** to any section, then you should follow the rules below:

Prefix	Description
`error_`	An error message
`msg_`	A regular information message
`title_`	A title, i.e. a dialog title
`action_`	An action such as "Save" or "Create"

2.3.2.3 Styles and Themes

Unlike the rest of resources, style names are written in **UpperCamelCase**.

2.3.3 Attributes ordering

As a general rule you should try to group similar attributes together. A good way of ordering the most common attributes is:

1. View Id
2. Style
3. Layout width and layout height
4. Other layout attributes, sorted alphabetically
5. Remaining attributes, sorted alphabetically

2.4.1 Unit tests

Test classes should match the name of the class the tests are targeting, followed by `Test`. For example, if we create a test class that contains tests for the `DatabaseHelper`, we should name it `DatabaseHelperTest`.
Test methods are annotated with `@Test` and should generally start with the name of the method that is being tested, followed by a precondition and/or expected behaviour.

- Template: `@Test void methodNamePreconditionExpectedBehaviour()`
- Example: `@Test void signInWithEmptyEmailFails()`

Precondition and/or expected behaviour may not always be required if the test is clear enough without them.

Sometimes a class may contain a large amount of methods, that at the same time require several tests for each method. In this case, it's recommendable to split up the test class into multiple ones. For example, if the `DataManager` contains a lot of methods we may want to divide it into `DataManagerSignInTest`, `DataManagerLoadUsersTest`, etc. Generally, you will be able to see what tests belong together because they have common test fixtures.

3 Additional guidelines

3.1 Structuring Project

For naming folder in project used an lowercase letter, for example view, model, util. We give example for project structure in our drive here.

3.2 API and Data Manager

We take any library and methods to get API, there's no limitation to use library such as volley, retrofit, or other library. What must be implemented in project is data manager to control each API so we can maintain it better. For the example of Data Manager, we will give in our example project.

3.3 Region (Code Marking)

For any method that has same purpose in one activity we need to give them code marking, in android we used region. With region we can divide our code and easily track our method that has same used in our activity. The usage of comment is needed too for describing the method that we made.

i. Copyright Statement

The following copyright statement should be included at the top of every source file:

```
//
// Class Name
// ecash
//
// Created by Reza on 1/15/18.
// Copyright © 2018 DIGITAL ARTHA MEDIA, PT. All rights reserved.
//
```

ii. References

- https://github.com/ribot/android-guidelines/blob/master/project_and_code_guidelines.md#1-project-guidelines
- https://gradle.org/
- https://developer.android.com/studio/build/index.html
- https://developer.android.com/studio/build/index.html#sourcesets
- https://developer.android.com/studio/build/index.html
- https://developer.android.com/about/dashboards/index.html#Platform
- https://docs.gradle.org/current/userguide/userguide_single.html#N10CBF
- https://developer.android.com/studio/build/gradle-tips.html
- https://developer.android.com/studio/build/index.html
- https://stackoverflow.com/questions/24785270/how-to-change-app-name-per-gradle-build-type
- https://en.wikipedia.org/wiki/Camel_case
- https://material.io/guidelines/style/icons.html
- https://en.wikipedia.org/wiki/Test_fixture
- https://repo.ptdam.com/ayatuna/android-project-guide.git

5. Monitoring / Progress Report	
Status of each deliverable (based on timeline)	Progress report once a week by e-mail, through face-to-face meeting (if necessary).
Risks / Issues	Not Applicable
Resources	Not Applicable
Next Action Plan (to solve issues occurred during the project)	Not Applicable
Request For Change (RFC)	Minor CR max 7man-days, @ Rp 2.200.000 Major CR max.21 man-days, @ Rp 1.700.000