**Coding Guidelines | Kotlin**

If in doubt default to the Java Coding Conventions such as:

* use of camelCase for names (and avoid underscore in names)
* types start with upper case
* methods and properties start with lower case
* use 4 space indentation
* public functions should have documentation such that it appears in Kotlin Doc ## Colon

There is a space before colon where colon separates type and supertype and there's no space where colon separates instance and type:

```

interface Foo<out T : Any> : Bar { fun foo(a: Int): T

}

```

## Lambdas

In lambda expressions, spaces should be used around the curly braces, as well as around the arrow which separates the parameters from the body. Whenever possible, a lambda should be passed outside of parentheses.

```

list.filter { it > 10 }.map { element -> element \* 2 }

```

In lambdas which are short and not nested, it's recommended to use the it convention instead of declaring the parameter explicitly. In nested lambdas with parameters, parameters should be always declared explicitly.

## Unit

If a function returns Unit, the return type should be omitted:

```

fun foo() { // ": Unit" is omitted here

}

```

## Functions vs Properties

In some cases functions with no arguments might be interchangeable with read-only properties. Although the semantics are similar, there are some stylistic conventions on when to prefer one to another.

Prefer a property over a function when the underlying algorithm:

* does not throw
* has a O(1) complexity
* is cheap to calculate (or caсhed on the first run)
* returns the same result over invocations ## Vokal Specific Standards

### Naming

In Java, it is common to name variables based on their scope. E.g. private `String mSomeString;`. This syntax does not lend itself well to Kotlin. Instead all variables should be named as lower-camelcase. This makes the variable more concise and readable.

### Types

When declaring a variable's type, always put a space between the `:` and the type name but not the variable name.

Do this:

```

val name: String

```

Not this:

```

val name : String

```

If at all possible, ignore explicit usage of the `Unit` type. Most commonly this would be in the case of a function that returns `Unit`:

Do this:

```

fun doSomething()

```

Not this:

```

fun doSomething(): Unit

```

### Naming

Use camelcase for variable naming. This will make for cleaner java interop if necessary. Do not use Hungarian notation or prefix your variables.

```

val myVal

``` becomes

``` myClass.getMyVal()

```

```

val my\_val

```

becomes

``` myClass.getMy\_val()

```

### Declaration

Types can be inferred by the right hand side of a variable assignment in Kotlin. This should be used unless otherwise necessary.

For example:

```

val a = 100 // Int is inferred

```

rather than:

```

val a: Int = 100 // Int is redundant

```

You should only specify the type of a variable when declaring without an initializer, which is enforced by the compiler:

```

val a: Int

// Do something

a = getIntResult()

```

When defining a class always put a space between the closing parenthesis of the primary constructor and the opening brack of the class body. Parameters in the default constructor should each appear on their own line:

Do this:

```

class User(

public open var firstName: String, public open var lastName: String

) {}

```

Not this:

```

class User(public open var firstName: String, public open var lastName: String){}

```

If your class is either a subclass or implements an interface the standard formatting rules for type declarations still apply. There should be a space between the `:` and name of the parent class, but not between the primary constructor and the `:`:

```

class User(

public open var firstName: String, public open var lastName: String

): RealmObject() {}

```

## Getters / Setters

In Java it is important to provide getters and setters for your member variables. This provides protection from rework if the implementation changes from being a direct access to the variable to something more complicated. Say, for instance, your timestamp format changes and you wish to do the conversion under the hood so as not to have to re-implement much of the processing code. Kotlin, however, uses the concept of "properties". All member variables automatically have the ability to modify their getters and setters. It is unnecessary to implement getters and setters for your properties.

```

private String mMyString;

public String getMyString() {

// Some special getter implementation

}

public void setMyString(String str) { mMyString = str;

}

``` becomes:

```

public var myString: String = ""

get() = // some special getter implementation

// Implicit standard setter

```

## When Statements

Like switch statements in Java, `when()` bodies should be concise and single line if possible.

Do this:

```

return when(myValue) {

is String -> myValue + "test"

is Number -> String.valueOf(myValue) else -> null

}

```

Not this:

```

return when(myValue) { is String -> {

var value = myValue + "test" value

}

```

}

is Number -> String.valueOf(myValue) else -> null

If a multi-line body is necessary, break it off into a separate method so that the

`when` can remain clean and concise. ## Nullability

As a rule of thumb, `!!` should never be used and `?` should be used rarely. Theses checks can often be avoided and doing so will provide for a more stable code-base. Whenever possible, objects and properties should be made to be not nullable.

## Strings

Always prefer string interpolation if possible:

Do this:

```

val fullName = "${user.firstName} ${user.lastName}"

```

Not this:

```

val fullName = user.firstName + " " + user.lastName

```

A NOTE ABOUT LOGGING: When using Timber, it is recommended to use the varargs as this string substitution will not be done on Release builds saving time on complex logs.

## Activities / Fragments

Always import the synthetic layouts for your activity or fragment. This cuts down on an immense amount of boilerplate and keeps your code clean.

Always prefer Kotlin-esque direct property accesses whenever available: Do this:

```

myTextView.text = "Hello, ${user.firstName}!" myTextView.visibility = View.VISIBLE

```

Not this:

```

myTextView.setText("Hello, ${user.firstName}!") myTextView.setVisibility(View.VISIBLE)

```

When referencing a parent activity from a fragment, do not call `getActivity()` like you would in Java:

Do this:

```

val typefaceSpan = CalligraphyTypefaceSpan( TypefaceUtils.load(activity.assets, "fonts/BooterFiveZero.ttf"))

```

Not this:

```

val typefaceSpan = CalligraphyTypefaceSpan( TypefaceUtils.load(getActivity().getAssets(), "fonts/BooterFiveZero.ttf"))

When defining a class-level constant, normal naming conventions apply:

```

Do this:

```

class MyFragment: Fragment() { private val TypeViewHeader = 0 private val TypeViewFooter = 1

}

```

Not this:

```

class MyFragment: Fragment() { private val TYPE\_VIEW\_HEADER = 0 private val TYPE\_VIEW\_FOOTER = 1

}

```

## Companion objects

Companion objects should always be defined immediately at the top of the class. Naming conventions for `val` within a companion object should follow our Java coding standards:

Do this:

```

class MyFragment: Fragment() { companion object {

const val TYPE\_VIEW\_HEADER = 0 const val TYPE\_VIEW\_FOOTER = 1

}

}

```

Not this:

```

class MyFragment: Fragment() { companion object {

val TypeViewHeader = 0 val TypeViewFooter = 1

}

}

```

## Views

View click listeners should call `setOnClickListener` directly along with lambda syntax. This avoids the boilerplate of declaring the right hand side of the assignment and keeps your code clean:

Do this:

``` myButton.setOnClickListener {

myTextView.text = "Hello, ${user.firstName}!"

}

```

Not this:

```

myButton.setOnClickListener = (object : View.OnClickListener() { myTextView.text = "Hello, ${user.firstName}!"

})

```

## Annotations

Annotations should appear above a class definition, and inline if it is annotating a field in a class:

Do this:

``` @Root(strict=false)

public open class User (

@field:Element public open var firstName: String? = "", @field:Element public open var lastName: String? = "",

) {}

```

Not this:

```

@Root(strict=false) public open class User ( @field:Element

public open var firstName: String? = "", @field:Element

public open var lastName: String? = "",

) {}

```

## Extensions

Extension functions should only be used if absolutely necessary. Given the option to define a class method vs an extension function, you should always prefer a class method.