SuperPhy CoffeeScript Style Guide

This is a guide for the best practices and conventions for writing in CoffeeScript for the SuperPhy project. This guide draws from Polar Mobile's CoffeeScript Style Guide. It may be modified in the future, so let me know if you think something should be added/removed.

Code Layout

Tabs/Spacing

Use tabs for each level of indentation. Do not use spaces.

White Space

Do not leave trailing white space on any lines.

Do not include extraneous white space immediately inside parentheses or before commas:

```
($ 'example') # Yes
( $ 'example' ) # No
```

```
console.log x, y # Yes
console.log x , y # No
```

Use a single space on either side of binary operators like +, –, =, etc.

Blank Lines

Separate method and class definitions with a single blank line. Use a single blank line within code blocks to logically separate sections and improve readability.

Optional Commas

Do not use commas between properties/elements of objects/arrays when they are listed on separate lines.

Comments

When modifying code that is currently described by comments, update the comments to reflect the changes.

Capitalize the first word of comments, unless the first word is an identifier that begins with a lower-case letter.

Block Comments

Block comments are used to apply to the block of code that follows them.

Each line of a block comment starts with a `#` followed by a single space. They should be indented to the same level of the described code.

Paragraphs should be separated by a line containing a `#`.

```
# This is an example of a block comment.  It is used to describe
# subsequent code.
#
# This is the first line of the second paragraph of the block
# comment.  It follows a blank line.
```

In-line Comments

Single-line comments should start with a `#` followed by a single space.

Comments describing specific lines of code should be placed immediately above the line in question. If the comment is sufficiently short, it may be placed on the same line as the code, separated by a single space.

Use in-line comments judiciously; they should be useful in describing the behaviour of a specific line in the context of the goal of the statement, as opposed to stating the obvious.

Annotations

Lines of code that require specific actions should be annotated above the line in question using a annotation keyword, followed by a colon and space, then a descriptive note. Extra lines should be tab-indented:

```
# TODO: Description of functionality to be implemented.
#   This is an extra line that has been tab-indented.
```

Annotation keywords:

- `TODO`: Describes missing functionality that needs to be implemented at a later date.
- `FIX`: Describes broken code that needs to be fixed.
- `OPTIMIZE`: Describes code that is inefficient.
- `REVIEW`: Describes code that should be reviewed to confirm implementation.

If custom annotation keywords are required, they should be added to the guide.

Naming Conventions

When naming variables, use a leading lower-case character, as in `variableName`. If there are multiple words within the name, capitalize subsequent words.

When naming classes, use a leading upper-case character, as in `ClassName`. If there are multiple words within the name, capitalize subsequent words.

Variables and methods that are intended to be "private" should begin with an underscore, as in `_privateMethod: ->`.

Files

CoffeeScript files should be named as `superphy_filename.coffee`.

Each file should be begin with comments describing its name, its purpose, its author, and the date of creation:

```
###

File: superphy_filename.coffee
Desc: Description of superphy_filename.coffee
Author: Author's Name (authors.name@email.com)
Date: May 11th, 2015

###
```

Classes

Classes should begin with comments describing its name and its purpose:

```
###

CLASS ClassName

Description of the name of the class

Description of the purpose of the class

###
```

Functions

Functions should be described by a block comment, describing the function, its parameters, and what it returns:

```
# FUNC functionName
# Description of the function
#
# PARAMS
# Param1 and description
# Param2 and description
#
```

```
# RETURNS
# Return type
#
```

Functions that take arguments should be declared using a single space after the closing parenthesis of the arguments list:

```
function = (arg1, arg2) ->
```

Functions that do not take any arguments should not use parentheses in their declarations:

```
function = ->
```

When method calls are being chained and the code does not fit on a single line, call each method on a new, tab-indented line

```
Object
    .function1()
    .function2()
    .function3()
```

Strings

Use string interpolation as opposed to string concatenation:

```
"This is an #{adjective} string" # Yes
"This is an" + adjective + " string" # No
```

Use single quotes wherever possible.

Conditionals

Use `unless` instead of `if` for negative conditions.

Use `if...else` instead of `unless...else`.

Multi-line if/else clauses should use indentation:

```
if true
    ...
else
    ...
```

Looping

Use comprehensions wherever possible:

```
# Yes
result = (item.name for item in array)

# No
results = []
for item in array
    results.push item.name
```

To filter:

```
result = (item for item in array when item.name is "test")
```

To iterate over the keys and values of object:

```
object = key1: val1, key2: val2
alert("#{key} = #{value}") for key, value of object
```

Miscellaneous

English operators should be used in place of symbolic operators:

`and` is preferred over `&&`.

`or` is preferred over `||`.

`is` is preferred over `==`.

`isnt` is preferred over `!=`.

`not` is preferred over `!`.

Use `@.property` in place of `this.property`, but avoid using `@` on its own:

```
return this # Yes
return @ # No
```

Use `return` only when its use increases clarity.

Use splats `...` when functions can accept variable numbers of arguments.