Cheat Sheet: JavaScript coding standard 2019-07-25

Why we need a coding standard

It improves product quality by minimizing common mistakes and miscommunication.
It helps deliver a better product faster by facilitating team communication and encouraging code review and reuse.
It helps avoid technical debt by encouraging self-documenting code that is understood by all.

General guidelines

- **Investigate third-party code** like jQuery plugins before building a custom module - balance the cost of integration with the benefits of standardization and code consistency
- **Avoid embedding** JavaScript code in HTML; use external libraries instead
- **Minify, obfuscate, and gzip JavaScript and CSS** before release (Buildify + Superpack)

Code layout and comments

<u>Use white space for readability</u>

- **Indent two spaces** per code level
- **Use spaces, not tabs** to indent as there is not a standard for the placement of tabs stops
- **Limit code and comment lines to a maximum of 78 characters**
- **Follow a function CALL with NO space** and then its opening left parenthesis, **(**
- **Follow a function DECLARATION with ONE space** and its opening left parenthesis, **(**
- **Follow a keyword with a single space** and then its opening left parenthesis, **(**
- **Each semicolon ;** in the control part of a **for** statement should be followed with a space
- **Align like elements vertically** to aid comprehension
- **Use single quotes** to delimit string literals

<u>Organize your code in paragraphs</u>

- **Organize code in paragraphs** and place blank lines between them
- **Use at least one line for each statement or assignment;** mutliple *declarations* may be placed on a single line within a **var** statement
- **Place white space between operators** and variables so that variables are easier to spot
- **Place white space after every comma**
- **Align like operators** within paragraphs
- **Indent comments** the same amount as the code they explain
- **Place a semicolon at the end of every statement**
- **Place braces around all statements in a control structure** like `for`, `if`, and `while`

<u>Break lines consistently</u>

- **Break lines before operators** as one can easily review all operators in the left column
- **Indent subsequent lines of the statement one level** e.g. two spaces in our case
- **Break lines after commas separators**
- **If there is no closing bracket or parenthesis**, place a semicolon it on its own line

<u>Use K&R style bracketing</u>

- **Place the opening** parenthesis, brace or bracket at the end of the opening line
- **Indent the code** inside the delimiters (parenthesis, brace, or bracket) one level
- **Place the closing** parenthesis, brace or bracket on its own line with the same indentation as the opening line

<u>Comment strategically</u>

- **Align comments** to the same level as the code they explain
- **Comment frugally** and apply comments to paragraph blocks
- **Non-trivial functions should explain** the **purpose** of the function, what **arguments** it uses, what **settings** it uses, what it **returns**, and any exceptions it **throws**
- **If you disable code,** explain why with a comment of the following format: **`// TODO <YYYY-MM-DD> <username> <urgency> : <comment>`**

<u>Document function APIs in-place</u>

```
// BEGIN DOM Method /toggleSlider/
// Summary   : toggleSlider( <boolean>, [ <callback_fn> ] )
// Purpose   : Extends and retracts chat slider
// Example   : toggleSlider( true );
// Arguments : (positional)
//   0: do_extend (boolean, required).
//      A truthy value extends slider.
//      A falsey value retracts it.
//   1: callback_fn (function, optional).
//      A function that will be executed
//      after animation is complete
// Settings  :
//   * chat_extend_ms, chat_retract_ms
//   * chat_extend_ht_px, chat_retract_ht_px
// Returns   : boolean
//   * true  - slider animation successfully initiated
//   * false - slider animation not initiated
// Throws    : none
//
function toggleSlider ( do_extend, callback_fn ) { … }
// END DOM Method /toggleSlider/
```

Variable names

<u>Use common characters</u>

- **Use only a-z, A-Z, 0-9, underscore, or $**
- **Do not begin a variable name with a number**

<u>Communicate variable scope</u>

- **Use camelCase when the variable is full-module scope** (i.e. it can be accessed anywhere in a module namespace)
- **Use snake_case when the variable is not full-module scope** (i.e. variables local to a function within a module namespace)
- **Make sure all module scope variables have at least two syllables** so that the scope is clear. For example, instead of using a variable called **config** we can use the more descriptive and obviously module-scoped **configMap**
- **Avoid module scope variables.** Instead, place static values in **topCmap** ("top config map") or **topSmap** ("top state map").
- **Wrap all private key names with underscores,** e.g. topSmap._is_open_. This allows SuperPack to improve compression by 30-50% and obsfucate much better.

Variable Name Convention (Indicator | Local Scope | Module scope)		
Boolean type		
_bool [generic]	return_bool	returnBool
is_ (indicates state)	is_retracted	isRetracted
do_ (requests action)	do_retract	doRetract
has_ (indicates inclusion)	has_whiskers	hasWhiskers
is_ (indicates state)	is_retracted	isRetracted
String type		
_str [generic]	direction_str	directionStr
_date	email_date	emailDate
_html	body_html	bodyHtml
_id	email_id	emailId
_msg	employee_msg	employeeMsg
_name	employee_name	employeeName
_txt	email_txt	emailTxt
Integer type		
_int [generic]	size_int	SizeInt
_count	user_count	userCount

Variable Name Convention (Indicator | Local Scope | Module scope)		
_idx	user_idx	userIdx
_ms (milliseconds)	click_delay_ms	clickDelayMs
i, j, k (convention)	i	–
Number type		
_num [generic]	size_num	SizeNum
_coord	x_coord	xCoord
_px (fractional unit)	x_px, y_px	xPx
_ratio	sale_ratio	saleRatio
x,y,z	x	–
Regex type		
_rx	match_rx	matchRx
Array type		
_list [generic]	timestamp_list color_list	timestampList colorList
_table [list of lists]	user_table	userTable
Map type		
_map [generic]	employee_map receipt_map	employeeMap receiptMap
Function type		
<verb><noun>_fn [generic]	bound_fn curry_get_list_fn get_car_list_fn fetch_car_list_fn remove_car_list_fn store_car_list_fn send_car_list_fn	boundFn curryGetListFn getCarListFn fetchCarListFn removeCarListFn storeCarListFn sendCarListFn
<verb><noun>	Not recommended	makeCurryList getCarList
Object type		
_obj [generic]	employee_obj receipt_obj error_obj	employeeObj receiptObj errorObj
$ (jQuery objects)	$header $area_tabs	$Header $areaTabs
_proto (protype object)	user_proto	userProto
Unknown type		
_data	http_data socket_data arg_data data	httpData, socketData

Function verbs

- **Function variable names should always start with a verb followed by a noun**
- **Module-scoped functions should always have two syllables** or more so the scope is clear, e.g. **getRecord** or **emptyCacheMap**

Function verbs		
Verb	**Example**	**Meaning**
fn	syncFn	Generic function indicator
bound	boundFn	A curried function that has a context bound to it.
curry	curryMakeUser	Return a function as specified by argument(s)
delete	deleteUserObj	Remove data structure from memory
destroy, remove	destroyUserObj	Same as delete, but implies references will be cleaned up as well
empty	emptyUserList	Remove all members of a data structure without removing the container
get	getUserObj	Get data structure from memory
make	makeUserObj	Create a new data structure using input parameters
store	storeUserList	Store data structure in memory
update	updateUserList	Change memory data structure in-place

Variable declaration and assignment

- **Use {} or []** instead of **new `Object()`** or **new `Array()`** to create a new object, map, or array. Avoid using **new** and use object contstrutors instead.
- **Use utilities like jQuery.extend to deep copy objects and arrays**
- **Explicitly declare all variables first** in the functional scope using a single var keyword
- **Use named arguments** whenever requiring 3 or more arguments in a function, as positional arguments are not self-documenting
- **Use one line per variable assignment.** Use alphabetical order if there is no other order. Group logically related assigments into parapgraphs

Functions

- **Declare most functions like so: `function doSomething ( arg_map ) { … }`.** Notice the space after the function name. Named functions are easier to debug.
- **Use functions to provide scope**, the JavaScript 'let' statement has questionable value
- **Declare all functions before they are used**
- **Use the factory pattern for object constructors**, as it better illustrates how JavaScript objects actually works, is very fast, and can be used to provide class-like capabilities
- **Avoid pseudo classical object constructors** - those that take a **new** keyword. If you must keep such a constructor, capitalize its first letter
- **When a function is to be invoked immediately,** wrap the function in parenthesis so that it is clear that the value being produced is the result of the function
- **Use jQuery** for DOM manipulations

Namespaces and file layout

Namespace basics

- **Claim a single, short name** (2-4 letters) for your application namespace, e.g. spa
- **Subdivide the namespace per responsibility**, e.g. **`spa.data`**, **`spa.model`**, **`spa.shell`**, etc

JavaScript files

- **Include third-party JavaScript** files first in our HTML so their functions may be evaluated and made ready to our application
- **Include our JavaScript** files in order of namespace. You cannot load namespace **spa.shell**, for example, if the root namespace, **spa**, has not yet been loaded
- **Give all JavaScript files a .js suffix**
- **Store all Static JavaScript files** under a directory called **js**
- **Use the template** to start any JavaScript module file
- **Name JavaScript files** according to the namespace they provide, one namespace per file proceeded by the software layer number. Examples includie spa.00_rootjs, spa.07_shell.js, spa.06_chat.js.

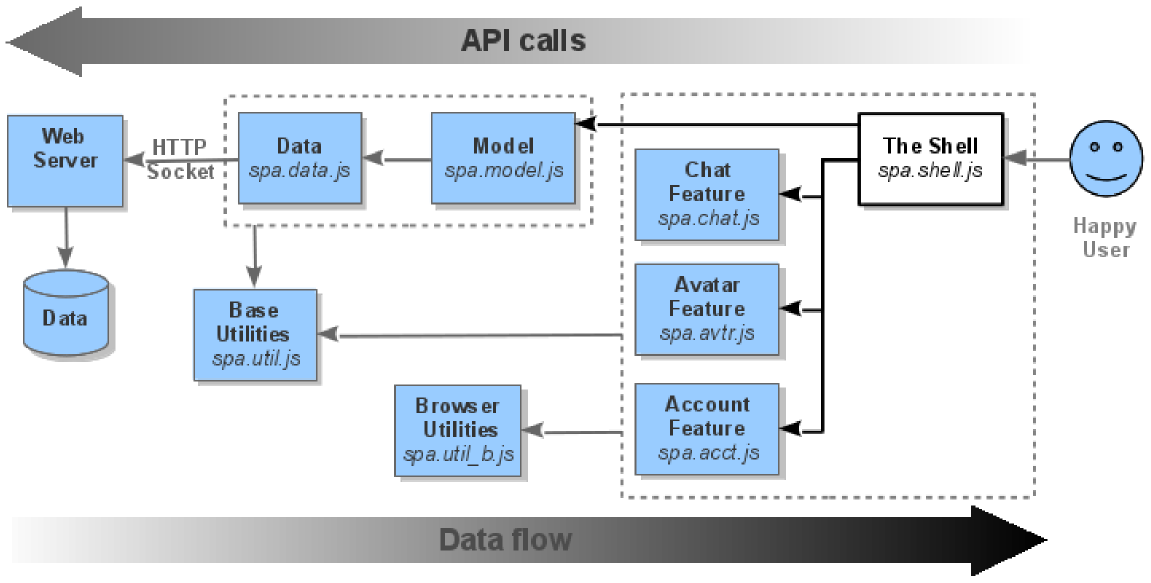
CSS files

- **Prefer PowerCSS,** replacing css files with corresponding JS files like `spa.07_01_css_shell.js`
- **A a PowerCSS or static CSS file should be created for each JavaScript file that generates HTML.** Examples: spa.07_01_css and spa.07_01_chat
- **Store all CSS files** under a **`css`** directory and use a **`css`** file extension.

- **CSS id's and class names** should be prefixed according to the name of the module they support. Examples: **spa.05_01_css_base.js** should define `#spa, .spa-x-clearall` while **`spa.07_01_shell_css.js`** should defines `#spa-shell-header, #spa-shell-footer, and .spa-shell-main`
- **Use an application prefix for all classes and id's** to avoid unintended interaction with third-party modules
- **Use <namespace>-x-<descriptor>** for state-indicator and other shared class names Examples include **spa-x-select** and **spa-x-disabled** and defined in the **spa.css** file

Code validation

- On commit or build these tests should be run on all revised or new code. This is built into hi_score: ESLint, TODO review, Unit and regression tests in test.d, coverage test, whitespace check.
- Always use `js/xhi/xhi-module-tmplt.js` as a starter file.


API calls
Web Server
HTTP Socket
Data spa.data.js
Model spa.model.js
The Shell spa.shell.js
Chat Feature spa.chat.js
Avatar Feature spa.avtr.js
Account Feature spa.acct.js
Data
Base Utilities spa.util.js
Browser Utilities spa.util_b.js
Happy User
Data flow