### JavaScript Coding Standards

### Comments

* Comments outside functions **should** use the C syntax, not the C++ syntax.

/\* This is the recommended comment format outside a function block. \*/

Multi-line comments should be aligned with existing indentation and each line should start with an asterisk.

/\*\*

\* This is a multi-line comment.

\* Note the alignment of the asterisks.

\*/

* Comments inside function blocks can be of either comment style.

/\* Should use this commenting style because this is not within a function. \*/

function foo()

{

// Valid comment format within a function block.

}

### Functions

* Function names are defined in camelCase.
* Private functions are prefixed with an \_underscore.
* Event handler functions should be prefixed with the word *on*.
* Function braces go on the line following the function name.

function myFunction(one, two)

{

...

}

* Inline functions **should** be named to aid debugging. Inline function names are not referenced in code.

var MyObj =

{

myFunction: function MyObj\_myFunction(one, two)

{

...

}

}

MyObj.myFunction(123, 456);

* Functions to be prefixed with header block consisting of the following parts:
  + *Function description* can include <pre> tags for sample code.
  + @method *functionName*
  + @static (*if the function is defined as static*)
  + @param *paramName* {*type*} *description*
  + @return {*type*} *description*
  + @private (**if the function is private**)

/\*\*

\* Adds three numbers together and returns sum.

\* <pre>

\* addNumbers(1, 2, 3);

\* </pre>

\* @method addNumbers

\* @param one {int} the first number to be added

\* @param two {int} the second number in the addition

\* @param three {int} the third number used in the sum

\* @return {int} the sum of parameters one, two and three

\*/

addNumbers: function Util\_addNumbers(one, two, three)

{

return (one + two + three);

}

### Callback functions

* Common structure for specifying optional callback functions

myCallback =

{

fn: function MyCallback(obj)

{

...

},

obj: myArbitraryObject,

scope: this

}

* Modules which support callbacks can either pre-define an empty function, and/or perform type checking before attempting the call

callbackFunction1 = null,

callbackFunction2 =

{

fn: function(obj){},

obj: null,

scope: this

}

...

if (callbackFunction1 && (typeof callbackFunction1.fn == "function")

{

callbackFunction1.fn.call(callbackFunction1.scope ? callbackFunction1.scope : this, callbackFunction1.obj);

}

callbackFunction2.fn.call(callbackFunction2.scope, callbackFunction2.obj);

### Overridable Functions

* Note the callbackFunction2 pattern in the example above can also be used for overridable abstract functions. In this case, the function **should** start with "do". e.g. doBeforeFormSubmit.
* A more useful overridable function pattern expects a returned parameter, e.g. when processing a data block or optionally halting further progress.

doBeforeSubmission =

{

fn: function(dataObj, obj)

{

return dataObj;

},

obj: null,

scope: this

}

...

var dataObj = { ... some data ... };

dataObj = doBeforeSubmission.fn.call(doBeforeSubmission.scope, dataObj, doBeforeSubmission.obj);

### Constants

* Constants **should** be defined in ALL\_CAPS.

### Tabs and Spaces

* Tab characters should not be used in source files.
* Indentations are 3 spaces per nesting level.
* Line lengths **should** be shorter than 80 characters where possible.
* Do not use spaces between a function and it's parameter list.
* Do use a space after a comma.

var total = addNumbers(1, 2, 3);

* Use a space between keywords and parentheses

if (condition)

{

...

}

### Indentation and Braces { }

* Braces **must** be used around all *if*, *do*, *switch*, etc. statements
* Braces **should** appear on their own line.

if (some condition)

{

...

}

else

{

...

}

* Do not put compound statements on one line

if (condition) return; /\* DO NOT USE \*/

if (condition) /\* OK \*/

{

return;

}

* **NOTE** the special format require when *return*ing an object

return /\* DO NOT USE \*/

{

param1: "foo",

param2: "bar"

}

return { /\* BETTER \*/

param1: "foo",

param2: "bar"

}

return ( /\* PREFERRED \*/

{

param1: "foo",

param2: "bar"

});

### JavaScript Features

* Use strict equality testing where possible, **must** be used when testing for *null*
  + This is because *!=* and *==* perform type coercion, which may or may not be intended.

if (x === null)

{

...

}

if (y !== null)

{

...

}

* **Do not** assign variables in *if* statements

if (a = b) /\* INTENTIONAL ASSIGNMENT, OR BUG? \*/

* Create arrays with []s rather than *new Array(length)*
* Create objects with {}s rather than *new Object()*
* Don't compare booleans to *true* or *false*

if (Service.isAvailable)

**Sample JavaScript WebScript Code**

Create Backup of a Document and Log Doc Properties

// find the backup folder - create if not already exists

var backupFolder = space.childByNamePath("Backup");

if (backupFolder == null && space.hasPermission("CreateChildren"))

{

// create the folder for the first time

backupFolder = space.createFolder("Backup");

}

if (backupFolder != null && backupFolder.hasPermission("CreateChildren"))

{

// copy the doc into the backup folder

var copy = document.copy(backupFolder);

if (copy != null)

{

// change the name so we know it's a backup

var backupName = "Backup of " + copy.name;

copy.name = backupName;

copy.save();

}

// record the time of the backup to a log file

var logFile = backupFolder.childByNamePath("backuplog.txt");

if (logFile == null)

{

logFile = backupFolder.createFile("backuplog.txt");

}

if (logFile != null)

{

logFile.content += "File: " + backupName +

"\tDate: " + new Date().toUTCString() +

"\tSize: " + copy.size + "\r\n";

}

}