JavaScript Coding Standards and Guidelines

**Introduction**

The purpose of coding standards and coding guidelines is to improve productivity and quality.  If everyone on a development team follows the same standards, the result is source code that is easier to read by all members of the team because it is written in a consistent style.  Easy to read code is then easier to debug and maintain by all members of the development team.  If a standard doesn't exist for the type of code you are writing, then at a minimum it should be coded in a consistent style throughout the program.

Source code readability is also the best indication of program quality.  If the source code is easy to read, it is probably a good program.  If it is hard to read, it probably is not.

* 1. **Naming Conventions – Variables**

Use meaningful, descriptive, English-like variable names, with capital letters at the start of each word or word segment, and lower case otherwise (i.e. camel-back notation).

Local variables begin with a lower case letter. Global variables should begin with a ‘g’ character and followed by the variable name in camel case. Even for a short-lived variable that may appear in only a few lines of code, still use a meaningful name. Use single-letter variable names, such as i, or j, for short-loop indexes only.

Example, do this:

var birthDate; // Local variable declaraction

var gStartDate; // Global variable declaration

Not this:

var BirthDate; // Local variable declaraction

var StartDate; // Global variable declaration

var msg; // Vague local variable declaration

**1.2    Naming Conventions - Consistency**

Data items should be named consistently throughout all parts of a program.

For example, suppose you are writing a program that processes information about books and one data item the program uses is the date the book was published.  Decide on a name for date published and use that same name whenever the data item is used.  This includes variable names, user-interface object names, and column names in a database.  The only time you would use anything different would be to add a prefix or change the capitalization of the name.

Do this:

DateTime datePublished;  
…  
datePublished = DateTime.Parse(txtDatePublished.value);  
…  
mBooksArray[index].DatePublished = datePublished;  
…  
datePublished = book.DatePublished;

Not this:

DateTime dtPub;  
…  
dtPub = DateTime.Parse(txtPubDate.value);  
…  
mBooksArray[index].published = dtPub;  
…  
pubDate = book.DatePublshd;

Use consistent abbreviates for the same word.  For example, if you decide to abbreviate Amount as Amt, then it should be abbreviated as Amt everywhere it is used.

Example, do this:

var hourlyPayAmt;  
var weeklyPayAmt;  
var taxAmt;

Not this:

var hourlyPayAmount;  
var weeklyPayAmt;  
var taxAmnt;

**1.4 Naming Conventions - Local Functions**

Use meaningful, descriptive and English-like method names. Method names should describe what the method does and not how it does it. Method names generally begin with a verb to indicate action, followed by a noun to indicate what they perform that action on. Local functions should start with a lowercase letter followed by camel case notation (like local variables).

Example, do this:

function showErrorMessage()  
{  
...  
}

Not this:

Function CalculateInvoiceTotal()  
{  
...  
}

**1.5 Naming Conventions - Class Functions**

Use meaningful, descriptive and English-like method names. Method names should describe what the method does and not how it does it. Method names generally begin with a verb to indicate action, followed by a noun to indicate what they perform that action on. Since in-built functions in javascript begin with a lowercase letter, followed by camel-back notation, user-defined methods should start with an uppercase letter followed by camel-back notation.

Example, do this:

TimeLogEntry.prototype.Serialize = function()  
{  
...  
}

When the function is called:

var timeLogEntry = new TimeLogEntry(); …  
var.Serialize();

Not this:

TimeLogEntry.prototype.serialize = function()  
{  
...  
}

When the function is called:

var timeLogEntry = new TimeLogEntry(); …  
var.serialize(); // looks similar to an in-built javascript function

**1.6    Naming Conventions - Source Code Files**

A source code file should begin with an uppercase letter and follow camel-back notation. It is a good idea to call the supporting HTML file and same name as the JavaScript file.

Example, do this:

TimeCardv10.html

TimeCardv10.js

Not this:

Index.html

Logic.js

**2    Comments**

Comments should be included in source code to indicate what the code does and to break up large sections of code.

Comments lines should begin with // indented at the same level as the code they are documenting.

Do not use /\* ... \*/ blocks for comments.

Indent comments at the same level of indentation as the code you are documenting

Source code comments should be written in clear, concise, English with correct spelling and grammar.

Comments should not include humorous remarks.  Your comments may seem funny to you when you type them, but they won't to the person who has to understand and fix your code during a late-night debugging session.

Comments should not be used to teach the reader how to program.  Assume the reader knows as much about the programming language as you do.

Comments should not be used to describe obvious code.  The purpose of comments is to increase your code readability.  Often using good variable and method names make the code easier to read than if too many comments are present.

Never "comment out" code without also including a description of why you did so.  If the code isn't necessary, it should be deleted.

Do not use "clutter" comments, such as an entire line of asterisks.  Instead, use a single blank line white space line to separate comments from code.

Do not "draw flower boxes" around comments using rows of asterisks or other characters.

**3.1    Format – End line with a semi-colon**

Always end a line of code with a semi-colon (althought this is not a requirement in JavaScript). This enhances readability.

Example, do this:

**3.2    Format - White Space**

Use a space before and after most operators.  Exceptions include ++ and --.  Semicolons should immediately follow the last character in the statement.

Example, do this:

function addTwoNumebrs(number1, number2)  
{  
    return number1 + number2;   
}

Not this:

function addTwoNumebrs(number1, number2)  
{  
    return number1+number2;   
}

**4.2    Format - One Statement Per Line**

Each line of code should not contain more than one statement.  Stacking several statements onto one line of code makes your code harder to read.

Example, do this:

var address;  
var city;  
var state;  
var zip;  
...  
address = txtAddress.value;  
city = txtCity.value;  
state = txtState.value;  
zip = txtZip.value;

Not this:

var address, city, state, zip;  
...  
address = txtAddress.value; city = txtCity.value; state = txtState.value; zip = txtZip.value;

**4.3    Format - Indenting**

You should indent all statements inside classes, methods, loops, *if* statements, switch statements, try/catch blocks, etc.  Statements should be indented in four-space increments (4, 8, 12, 16, 20, etc.).

Example, do this:

function httpStateChange() {

if (httpRequest.readyState === 4) {

clearTimeout(timeoutFunction);

if (httpRequest.status === 200) {

if (httpRequest.responseText.substring(0, 6) === "error\n") {

gServerErrorMsg = httpRequest.responseText.substring(6);

}

callbackFunction(httpRequest.responseText);

}

else {

if (aborted === false) {

gServerErrorMsg = "Error received from the server.";

}

}

}

}

Not this:

function httpStateChange() {

if (httpRequest.readyState === 4) {

clearTimeout(timeoutFunction);

if (httpRequest.status === 200) {

if (httpRequest.responseText.substring(0, 6) === "error\n") {

gServerErrorMsg = httpRequest.responseText.substring(6);

}

callbackFunction(httpRequest.responseText);

}

else {

if (aborted === false) {

gServerErrorMsg = "Error received from the server.";

}

}

}

}

**4.4    Format - Brace Alignment**

Open braces should be on the same line as the code while the closing braces should be on a line of their own.

Example, do this:

for (var i = 0; i < 100; i++) {

if (totPurchaseAmt < 0) {

...

}

}

Not this:

for (var i = 0; i < 100; i++)

{

if (totPurchaseAmt < 0)

{

...

}

}

**5.1 Decisions: Eqaulity operator**

Always use “===” to check equality rather than “==”.

Example, do this:

if (itemFound == true) {  
    itemCounter++;  
}

Not this:

if (itemFound == true) {  
    itemCounter++;  
}

**5.2   Decisions with single Statements**

Always use a curly brace scope in an if statement, even if if a true condition executes a single statement.

Example, do this:

if (itemFound === true) {  
    itemCounter++;  
}

Not this:

if (itemFound === true)  
    itemCounter++;

**5.3  Decisions with multiple conditions**

When the logical expressions of a decision statement include more than two expressions, code each condition on a line by itself and begin the line with the logical operator.

Example, do this:

if (txtID.value.length == 5   
 && txtFirstName.value != ""   
 && txtLastName.value != ""   
 && txtStartDate.value != ""   
 && txtClass.value.length == 2){  
    return false;  
}

Not this:

if (txtID.value.length == 5 && txtFirstName.value != "" && txtLastName.value !=   
"" && txtStartDate.value != "" && txtClass.value.length == 2)  
{  
    return false;  
}

**6    f*or* Loops or *while* Loops**

When should you use a *for* loop instead of a *while* loop?

Basically, you use a *for* loop if you know how many times you are going to go through the loop (before you start looping).  If you are looping based on a constant value or a user inputted value, then the program will "know" how many times it will loop before it starts looping even if you don't know what the user will input when you are writing your loop.

*for* loops are preferred because the syntax automatically initializes the counter variable, increments it, and checks to see if the end of the number of iterations has been reached.

If you are going to loop until something happens (such as read each line in a file until you reach the end of the file) then you will want to use a *while*.

**7  Classes**

Although it is possible to add fields/ methods to objects in Javascript at any point in the code, it is better to define these separately in a class file of its own.

**7.1  Classes – Constructors**

**7.1.1  Empty constructors:**

Since there is not a clear concept of class variables in JavaScript, it is a good idea to give the variable declarations some default value to make it explicit for someone reading the code what exactly are all the class variables. The class variables should start with an uppercase letter and follow camelback notation.

Example, do this:

function Contact() {

this.Name = “”;

this.BirthDate = “”;

this.NumberOfFamilyMembers = null;

}

Not this:

function Contact() {

this.name = “”;

this.birthDate = “”;

this.numberOfFamilyMembers = null;

}

**7.1.1  Constructors with parameters:**

The class variables should start with an uppercase letter and follow camelback notation. The constructor variables should use the same name as the class variables however they should start with a lowercase letter instead.

Example, do this:

function Contact(name, birthDate, numberOfFamilyMembers) {

this.Name = name;

this.BirthDate = birthDate;

this.NumberOfFamilyMembers = numberOfFamilyMembers;

}

Not this:

function Contact(Name, BirthDate, NumberOfFamilyMembers) {

this.Name = Name;

this.BirthDate = BirthDate;

this.NumberOfFamilyMembers = NumberOfFamilyMembers;

}

**7.2  Constructor Methods**

Explicitly define all constructor methods whose names should start with an uppercase letter followed by camel case notation. This provides a distinction with the in-built javascript functions.

Example, do this:

Contact.prototype.Deserialize = function (serializedEntry) {

var entry = serializedEntry.split("\t");

this.Name = entry[0];

this.BirthDate = entry[1];

this. NumberOfFamilyMembers = parseInt(entry[2]);

}

Example, do this:

Contact.prototype.deserialize = function (serializedEntry) {

var entry = serializedEntry.split("\t");

this.Name = entry[0];

this.BirthDate = entry[1];

this. NumberOfFamilyMembers = parseInt(entry[2]);

}