Sencha's Guide to JavaScript Style and Best Practices

The following document contains a series of best practices and recommendations for building enterprise web applications with JavaScript and HTML5.

Much of this advice is used internally at Sencha and certainly applies to building applications with the Sencha frameworks -- but this JavaScript style guide can also help teams building applications with any framework, or even just vanilla JavaScript.

Mission Statement

Sencha firmly believes that high quality JavaScript is above all else consistent, and addresses four main pillars:

- **Readability:** JavaScript code should be clear and easy to understand at first glance
- **Maintainability:** JavaScript code should be consistent and cohesive
- **Error Prevention:** JavaScript code should strategically avoid common problems
- **Performance:** JavaScript code should always consider faster implementations

This style guide will attempt to provide some ground rules for building enterprise web applications with JavaScript and HTML5, and as such is intended primarily for medium-to-large development teams. These recommendations are based on our own experiences, as well as direct interaction with our enterprise customers.

Dogma vs Doctrine

Avoid obsessing about code style -- this document is merely a guide, not infallible dogma. It is not intended to be a comprehensive list of all "best practices", but rather a list of the most common areas on which people ask our opinions. Many others have created similar documents and clearly not everyone agrees on every point. The overall goal for this document is to help you evaluate what we consider to be the important aspects of "quality" code, and ultimately create your own readable, maintainable and scalable JavaScript projects. Our suggestions stand as a baseline from which your teams should implement your own strategy for building a high quality JavaScript codebase.

Outline

To best articulate Sencha's experience building both frameworks and enterprise applications, we will divide this discussion into four parts:

1. Readability
2. Maintainability
3. Preventing Errors
4. Performance

It is worth noting that many of the points we will cover might fit into more than one of these pillars.

Writing Readable JavaScript

The importance of writing "readable" JavaScript simply cannot be overstated. Be kind to the others on your team, and your future self, by writing code that is easy to digest.

In his book Facts and Fallacies of Software Engineering, Robert Glass discusses how simply "understanding the existing product" consumes roughly 30% of a developer's time. Glass frames this point within the context of the software maintenance cycle -- and while we will move our discussion to maintainability soon, Sencha recommends prioritizing the following points to make your codebase clear and coherent:

- Naming Conventions
- Comments and Documentation
- Documenting Overrides
- Spacing and White Space
- Line Length
- Block Length
- File Length

Naming Conventions

"There are only two hard things in Computer Science: cache invalidation, naming things, and off-by-one errors."

Many people only name a few things in their life: a pet, a child, or something particularly personal and meaningful. In general, naming things is difficult because there is a perceived finality to the process (e.g. you rarely re-name a pet) -- but in software development, one typically has to name things far more often than the average person.

Software developers might name thousands of new things per year: variables, classes, applications, etc. Each name attempts to capture the essence or purpose of the concept being named -- but because programs use more names than can be reasonably remembered, the names must be conceived consistently in order to help others understand, fix, and extend that same code months or years later. This idea is often referred to as "self-documenting code".

Let's examine the areas in which naming conventions should be applied and how Sencha handles each case.

Namespaces, Classes, and Constructors

Sencha always uses TitleCase when creating top-level namespaces, classes, and constructors.

```
// "MyClass" is a constructor :: new MyClass();
MyClass = function() {};
```

Intermediate namespaces should be short, descriptive and lowercase.

```
// "Foo" as the top-level namespace
// "bar" as an intermediate-level namespace
// "Baz" as the class name
Foo.bar.Baz = {};
Ext.data.reader.Json = {};
```

Functions

Sencha always uses camelCase when creating functions. We also recommend the use a leading underscore "_" when naming private functions and methods that are not encapsulated by a closure.

```
// function expression
var sortSomeStuff = function() {};

// function declaration
function findSomething() {}

// the same concept applies to object functions
var someObject = {
    objectMethod : function() {},

    _privateMethod : function() {}
};
```

Note: more information about using function expressions vs. function declarations can be found in section Functions.

Local Variables and Object Properties

Always use var to declare local variables -- not doing so will result in the creation of global variables, and we want to avoid polluting the global namespace. See the Constants and Global Variables section below for more details.

```
// bad
foo = true;

// good
var foo = true;
```

Sencha always uses camelCase when creating local variables and object properties. We also recommend the use a leading underscore "_" when naming private properties.

```
// local variable
var fooBar = true;

// object property
var someObject = {
    someProperty : true,
    _privateProperty : true
};
```

Variables should be given meaningful names, so that the intended purpose and functionality of the variables is clear (while also concise). Avoid single letter names; the lone exception to this rule would be an iterator.

```
// bad, too short. Not descriptive and easy to confuse with number "1"
var l = group.length;

// bad, variable name is unnecessarily long
var mainClassConfigVariableSectionOneRefreshInterval = 5000;

// good, variable name is concise yet still meaningful
var len = group.length;

// iterators are an exception
var i;
for (i = 0; i < len; i++) {}
```

Use one var declaration when creating multiple variables because it is easier to read. Sencha recommends declaring each variable assignment on a new line; declare unassigned variables last, though these can be on the same line. This helps to provide a visual cue to the person reading your code about the initial state of the variables within the current scope.

```
// bad
var foo = 1;
var bar = 2;
var baz;
var fuz;

// good
var foo = 1,
    bar = 2,
    baz, fuz;
```

Constants and Global Variables

Sencha recommends using CONSTANT_CASE when creating global variables because of the clear visual indication that the variable is special.

```
// bad
userID = '12345';
// good
USER_ID = '12345';
```

Having said that, Sencha prefers to avoid global variables and constants altogether. We feel that enterprise applications benefit from using properly-namespaced classes instead because it's always clear where a value has been defined.

```
// better
MyApp.authentication.User = {
	id : '12345'
};
```

Special Cases

Other special cases also exist -- for example, naming references to this.
As an internal convention, Sencha uses the name me when there is a need to capture a reference to this within a closure. Not everyone agrees -- Christian Johansen is a notable example -- but the greater point is to manage these special cases consistently throughout your codebase.

```
Person.logger = function() {
	var me = this; // "me" will be used consistently
	return function() {
		console.log(me);
	};
};
```

Another important thing to note is that this is a keyword and can't be compressed. In the Sencha frameworks, we abide by the rule of four: if a given scope references this four or more times, cache this using the local variable meas it will make the minified source smaller.

```
// bad
function foo() {
	this.x = 1;
	this.y = 2;
	this.z = 3;
	this.u = 4;
}
// good
function foo() {
	var me = this;
	me.x = 1;
	me.y = 2;
	me.z = 3;
	me.u = 4;
}
```

```
// comparison of minified output
function f(){this.x=1;this.y=2;this.z=3}
function f(){var e=this;e.x=1;e.y=2;e.z=3}

function f(){this.x=1;this.y=2;this.z=3;this.u=4;}
function f(){var e=this;e.x=1;e.y=2;e.z=3;e.u=4;} // 4 is now shorter!

function f(){this.x=1;this.y=2;this.z=3;this.u=4;this.v=5;}
function f(){var e=this;e.x=1;e.y=2;e.z=3;e.u=4;e.v=5;}
```

Reserved Words

Don't use reserved words as keys because they break things in older versions of Internet Explorer. Use readable synonyms in place of reserved words instead.

```
// bad
var model = {
        name    : 'Foo',
        private : true // reserved word!
};
// good
var model = {
        name   : 'Foo',
        hidden : true
};
```

Note: "readable synonyms" must actually be words.

```
// bad

var car = {
        class : 'Ford' // reserved word!
};
// bad
var car = {
        klass : 'Ford' // PLEASE don't ever do this!
};
// good
var car = {
        brand : 'Ford'
};
```

Comments and Documentation

Generally speaking, good code is supposed to be self-explanatory. However comments play two vital roles in promoting readable code: documentation, and intent (via inline comments).

Documentation

System-wide documentation is vital to developing large codebases. Using tools like JSDuck it is easy to build an API reference for your codebase, making it significantly easier for your team (and others) to digest.

Sencha uses JSDuck internally, which follows the JavaDoc style for block comments. See the JSDuck wiki for more information.

```
/**
  * @class MyApp.foo.Bar
  */
MyApp.foo.Bar = function() {
        var baz = true;

        return {
                /**
                  * @method
                  */
                utilityMethod : function() {
                        return baz;
                }
        };
};
```

Inline Comments

Many developers feel that code ought to be "self-documenting" and therefore inline comments are to be avoided. Sencha doesn't necessarily agree with the rigidness of that mindset; we believe that comments should always be added when the intent or purpose of any code isn't completely explicit, but the code itself ought to be clear enough to follow logically.

```
// In a majority of cases, the controller ID will be the same as the name.
// However, when a controller is manually given an ID, it will be keyed
// in the collection that way. So if we don't find it, we attempt to loop
// over the existing controllers and find it by classname
if (!controller) {
    all = controllers.items;
    for (i = 0, len = all.length; i < len; ++i) {
        cls = all[i];
        className = cls.getModuleClassName();
        if (className && className === name) {
            controller = cls;
            break;
        }
    }
}
```

Regular expressions should also always be explained with a comment because of their inherently confusing syntax.

```
// match Roman Number input
var romanNums = /^M{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$/;
```

Commenting out entire blocks of code should be generally avoided because they serve no purpose and create bloated code.

```
// Why leave the following code in production?
items    : [
    //{
    //    xtype        : 'booleancolumn',
    //    width        : 5,
    //    resizable    : false,
    //    defaultWidth : 5,
    //    sortable     : false,
    //    dataIndex    : 'isOwn',
    //    groupable    : false,
    //    hideable     : false,
    //    lockable     : false,
    //    tdCls        : 'indicator',
    //    falseText    : ' ',
    //    trueText     : ' '
    //},
    //{
    //    xtype     : 'gridcolumn',
        //    dataIndex : 'key',
        //    text      : 'Binding Key',
        //    flex      : 1
        //},
        {
            xtype     : 'templatecolumn',
            dataIndex : 'boundTo',
            text      : 'Bound To',
            flex      : 1,
            tpl       : '\\{{boundTo}\\}'
        },
        {
            xtype     : 'gridcolumn',
            dataIndex : 'value',
            text      : 'Value',
            flex      : 1,
            renderer  : function (value, metaData, record, rowIndex, colIndex, store, view) {
                var v = value;

                if (value === null) {
                    v = 'null';
                }

                if (record.data.text === 'undefined') {
                    v = 'undefined';
                }

                return '<span class="highlight ' + record.get('type') + ' ' + v + '">' + v +
'</span>';
            }
        }
    ]
```

Documenting Overrides

In cases where you need to override default or inherited functionality, both inline and block comments are actively encouraged so that the changes are perfectly clear.

```
// OVERRIDE for bug EXTJS-12345
Ext.define('MyApp.override.CustomNumberField', {
        override : 'Ext.form.field.Number',

        initComponent: function() {
                var me = this,
                        allowed;

                me.callParent();

                me.setMinValue(me.minValue);
                me.setMaxValue(me.maxValue);

                // Build regexes for masking and stripping based on the configured options
                if (me.disableKeyFilter !== true) {
                        allowed = me.baseChars + '';
                        if (me.allowDecimals) {
                                //OVERRIDE THIS LINE...
                                //allowed += me.decimalSeparator;
                                allowed += ',.';
                        }
                }
        }
});
```

This documentation will often come in handy during an upgrade process. For example, the bug EXTJS-12345 might have been fixed in the latest version -- so this override could be removed completely.

Spacing and White Space

Many developers have strong opinions on the topic of tabs-vs-spaces for spacing. Sencha advocates the use of four spaces in our own code because tab sizes are unpredictable; the only way we can guarantee readable code is to enforce the use of spaces. Ultimately the goal is just to have consistency, so whatever your choice don't mix them!

```
// bad
function doSomething(isTrue) {
  // < 1 space in
  if (isTrue) {
        // <<< 3 spaces in?
}// now you're just being confusing...
}

// good
function doSomething(isTrue) {
        // <<<< 4 spaces in!
        if (isTrue) {
                // <<<< 4 spaces in again!
        }
}
```

On the other hand, Sencha also advocates for using as much white space as necessary to make your code easier to read.

Line Length

Not everyone agrees with the specific limit for characters-per-line, but Sencha generally tries to limit line length. This limit can be arbitrary (e.g. 80 or 100 characters) and not rigidly enforced, but the goal is to reduce the amount of horizontal scrolling for the developer.

Strings longer than the decided limit should be written across multiple lines using string concatenation.

Method and Block Length

How long can a method or code block get before you consider breaking functionality into smaller utility methods?

A good rule-of-thumb is to limit the length of method and code blocks (e.g. 50 or 100 lines) so that they are not trying to do too much. Shorter methods are easier to test, and smaller sections of code are more quickly comprehended by developers.

File Length

How long should a file be before you consider breaking functionality into mixins, modules or other utility classes?

As with method/block length, comments can easily impact the length of a file. Abstract classes might also be longer than usual because they define interfaces and baseline functionality. Nevertheless, defining an arbitrary file length (e.g. 500 or 1000 lines) might give you an indication of whether-or-not a class might need to be refactored.

Preventing JavaScript Errors

Defensive Programming is an approach to building software which strives to reduce the likelihood of common problems. Because JavaScript syntax contains many gotchas and edge cases, adopting a defensive programming mindset can help to prevent many runtime errors in large codebases.

- Removing Lint
- Using Semicolons
- Trailing Commas
- Brackets and Blocks
- Equality
- Prototypes
- Functions

Removing Lint

JavaScript "lint" tools read your source code to help identify common mistakes -- things as subjective as multiple varstatements in a function, or as objective as flagrant syntax errors. These mistakes are considered "lint" and should be removed or restructured.
Using tools such as JSLint, ESLint, JSHint, etc is always recommended during development. Many popular IDEs even have direct integration with these tools.

However, you should also consider making these tools part of your automated testing or build processes. IDE integration will only warn the developer about errors - it does not force them to correct problems, and it will not prevent the bad code from being committed to your repo.

Using Semicolons

Automatic Semicolon Insertion (ASI) is not a feature. Don't rely on it. Crockford recommends putting a semicolon at the end of every simple statement because JavaScript allows any expression to be used as a statement, which can mask some tricky errors.

```
// bad
var a = obj
[a].forEach(logProp);
// Because a semicolon isn't used that code behaves like this:
// var a = obj[a].forEach(logProp)

// good
var a = obj;
[a].forEach(logProp); // this works fine
// bad
var example = function() {
	// because of the line break, ASI returns "undefined"
	return
	{
		foo: 123
	};
};

// good
var example = function() {
	return {
		foo: 123
	};
};
```

Trailing Commas

Trailing commas have caused more headaches in JavaScript development over the years than perhaps anything else.

```
var myObject = {
	foo : 1,
	bar : 2, // trailing comma
};
var myArray = [ 1, 2, 3, ]; // trailing comma
```

Although the current ECMAScript 5 specification allows for trailing commas in Object and Array literals, older browsers (particularly IE <9) encounter unexpected behavior: trailing commas in object literals would throw runtime errors, while trailing commas in array literals would return inaccurate results for Array.length.

Therefore, as a best practice, Sencha discourages developers from using them.

On a related note, some developers prefer to use leading commas to avoid this problem. Sencha doesn't feel that solution adequately solves the issue, and furthermore we believe it reduces the readability of the code.

Brackets and Blocks

Always use brackets when creating code blocks of any kind. Every block, even if it is only one line, needs to have its own curly braces in order to avoid confusion.

```
// bad
if (foobar) doSomething();

// good
if (foobar) {
	doSomething();
}
```

In many cases, the use of guard clauses makes good sense as they highlight exceptions to the "normal" execution path:

```
// bad
function getPayAmount() {
	var result;

	if (_isDead) { result = deadAmount(); }
	else {
		if (_isSeparated) { result = separatedAmount(); }
		else {
			if (_isRetired) { result = retiredAmount(); }
			else { result = normalPayAmount(); }
		};
	}
	return result;
};
```

```
// good
function getPayAmount() {
	if (_isDead) {
		return deadAmount();
	}
	if (_isSeparated) {
		return separatedAmount();
	}
	if (_isRetired) {
		return retiredAmount();
	}

	return normalPayAmount();
};
```

Equality

Always favor the === and !== operators over == and != unless you have specific reasons not to.
By using the "strict equality operators", we can accurately compare the value and type of the variables being compared.

```
// bad
var result = (0 == false); //returns TRUE

// good
var result = (0 === false); //returns FALSE
```

The non-strict equality operators (== and !=) will attempt to cast the operands into the same value type, returning truthy or falsy results which may not be expected.
The same problem exists when comparing native types in an if statement:

```
function compare(val) {
	return (val) ? true : false;
}

compare({}); // evaluates to true
compare([]); // evaluates to true, because Array is an Object

compare(undefined); // evaluates to false
compare(null);      // evaluates to false

compare(true);  // evaluates to true
compare(false); // evaluates to false

compare(0);   // +0, -0 evaluate to false
compare(NaN); // evaluates to false
compare(1);   // all other positive numbers evaluate to true
compare(-1);  // all other negative numbers evaluate to true

compare('');    // empty string evaluates to false
compare('foo'); // all other strings evaluate to true
```

In short, you need to be very careful when testing the equality of any variables when not using the strict equality operators!

However, there are some situations in which using truthy or falsy values without direct comparison are acceptable -- but again, developers should always be cautious about the values they expect.

```
if (!disabled) {
        // ...
}

// or
if (enabled) {
        // ...
}
```

Prototypes

JavaScript is a prototype-based language -- all objects inherit directly from other objects, and formal "classes" do not exist. Prototypal inheritance is conceptually similar to classical inheritance, but Crockford points out that "JavaScript is conflicted about its prototypal nature" because "the prototype mechanism is obscured by some complicated syntactic business that looks vaguely classical".

In short, understanding the nuances of prototypal inheritance is key to avoiding errors.

Native Prototypes

Hacking native prototypes should be avoided. It increases the possibility of naming collisions and incompatible implementations, inevitably causing headaches and hard-to-find bugs.

Instead, create utility classes/methods to implement the desired behavior:

```
// bad
Array.prototype.each = function(functionToCall) {
        //loop over the items in the array
};

// good
Ext.define('Ext.Array', {
        singleton : true,
        each : function(arrayToIterate, functionToCall) {
                //loop over the items in the array
        }
}};
```

Note: in some situations, polyfilling native prototypes may be acceptable to add standard behavior to older browsers. For example, Ext JS 5 pollyfills Function.bind() in IE8.

Functions

Function Hoisting

When defining JavaScript functions, beware of hoisting. Function declarations are evaluated at parse-time (when the browser first downloads the code):

```
// FUNCTION DECLARATION (preferred)
function sum(x, y) {
    return x + y;
}
```

Because the declaration is hoisted to the top of its scope at parse-time, it doesn't matter when the function is defined:

```
sum(1,2); // returns 3
// FUNCTION DECLARATION (preferred)
function sum(x, y) {
    return x + y;
}
```

Function expressions are evaluated at run-time (when the call stack physically hits a line of code), just like any other variable assignment:

```
// FUNCTION EXPRESSION
var sum = function(x, y) {
	return x + y;
};
```

Because function expressions are NOT hoisted at parse-time, it DOES matter when the function is defined:

```
sum(1,2); //throws an error "undefined is not a function"

// FUNCTION EXPRESSION
var sum = function(x, y) {
	return x + y;
};
```

Anonymous Functions

Anonymous functions can be very convenient, but poorly constructed code can easily lead to memory leaks. Consider the following example:

```
function addHandler() {
	var el = document.getElementById('el');

	el.addEventListener(
		'click',
		function() { // anonymous function
			el.style.backgroundColor = 'red';
		}
	);
}
```

There are two issues caused by using an anonymous function:

1. We don’t have a named reference to the click handler function, so we can’t remove it via removeEventListener()

2. The reference to el is inadvertently caught in the closure created for the inner function, and therefore cannot be garbage collected. This creates a circular reference between JavaScript (the function) and the DOM (el).

To avoid the first problem, always use named functions when adding event listeners to

DOM elements.

To avoid the second problem, carefully craft your scopes to prevent leaks and promote garbage collection:

```
function clickHandler() {
	this.style.backgroundColor = 'red';
}

function addHandler() {
	var el = document.getElementById('el');
	el.addEventListener('click', clickHandler);
}
```

One final note: the risks exposed by the anonymous function pattern are often mitigated by following the paradigms of the Sencha class system, where the framework manages much of this scoping and binding for you.

Keeping an Eye on Performance

"Performance is only a problem if performance is a problem."
Every developer should care about performance -- but developers also shouldn't spend time optimizing minute sections of code without first proving such efficiencies are necessary. For example, optimizations that make sense in a JavaScript library/framework may have little impact in application code.

Given our experience building the Ext JS and Touch frameworks and our exposure to customer applications, Sencha has identified the following techniques as proven methods for improving performance:

Library or Framework Code

- Loops
- Try \ Catch
- Page Reflow
- Function-based Iteration

Application Code

The Yahoo team put together a document of exceptional performance techniques that is so thorough it’s hard to add anything.

Nevertheless, Sencha applications should utilize Sencha Cmd as part of their build process in order to compress the code to the smallest possible size. Be sure to follow our Compiler-Friendly Code Guidelines!

Loops

Don't declare functions (and for that matter, other re-usable things) inside of loops. It wastes processing time, memory, and garbage collection cycles:

```
// bad
var objectPool = [];

for (i = 0; i < 10; i++) {
        objectPool.push({
                foo : function() {}
        });
}

// good
var objectPool = [];

function bar() {}

for (i=0; i<10; i++) {
        objectPool.push({
                foo : bar
        });
}
```

Calculate array length only once upfront and assign its value to a variable. This will prevent measuring it for every iteration.

```
// bad
var i;
for(i=0; i<items.length; i++){
        // some code
}
```

```
// good
var i, len;
for(i=0, len=items.length; i<len; i++){
        // some code
}

// bad
var i;
for(i=0; i<items.getCount(); i++){
        // some code
}

// good
var i, len;
for(i=0, len=items.getCount(); i<len; i++){
        // some code
}
```

Whenever possible avoid for/in type of loop as they are known to negatively impact performance.

Try \ Catch

Avoid try/catch statements when possible as they cause significant drags on performance.

Page Reflow

Avoid patterns that cause unnecessary page reflows.

A reflow involves changes that affect the CSS layout of a portion or the entire HTML page. Reflow of an element causes the subsequent reflow of all child and ancestor elements, as well as any elements following it in the DOM.

The browser will automatically keep track of DOM and CSS changes, issuing a "reflow" when it needs to change the position or appearance of something.

Unwieldy JavaScript code can force the browser to invalidate the CSS layout -- for example, reading certain results from the DOM (e.g. offsetHeight) can cause browser style recalculation of layout. Therefore developers must be incredibly careful to avoid causing multiple page reflows as they will cause application performance to noticeably lag.

```
// bad
elementA.className = "a-style";       // style change invalidates the CSS layout
var heightA = elementA.offsetHeight;  // reflow to calculate offset
elementB.className = "b-style";       // invalidates the CSS layout again
var heightB = elementB.offsetHeight;  // reflow to calculate offset

// good
elementA.className = "a-style";       // style change invalidates the CSS layout
elementB.className = "b-style";       // CSS layout is already invalid; but no reflow yet
var heightA = elementA.offsetHeight;  // reflow to calculate offset
var heightB = elementB.offsetHeight;  // CSS layout is up-to-date; no second reflow!
```

Function-Based Iteration

Function-based iteration, while convenient, will always be slower than using a loop.

```
var myArray = [ 1, 2, 3 ];

// jQuery
$.each(myArray, function(index, value) {
       console.log(index + ": " + value);
});

// Ext JS 5
Ext.each(myArray, function(value, index) {
       console.log(index + ": " + value);
});

// BETTER PERFORMANCE!
var len = myArray.length,
       i, prop;
for (i=0; i<len; i++) {
       console.log(i + ": " + myArray[i]);
}
```

Every function invocation creates a new execution context (scope chain). Function calls and returns require state preservation and restoration, as well as garbage collection -- while iteration simply jumps to another point in the existing context.