Coding Style Guide

What is readability?

Code readability refers to writing code whose purpose is clear and self-explanatory, and whose syntax, layout, and structure are consistent and unsurprising.

Code with lots of inconsistencies requires more mental effort from the code reader, who must parse through random visual differences to determine what differences are actually meaningful.

Often that confused code reader is you.

Confusing yourself and struggling to parse your own code in the context of an interview/admissions assessment is Bad News. Don't make things harder for yourself!

Follow these guidelines, and you'll be ok.

Indentation

When writing any block of code that is logically subordinate, that block should be indented two spaces more than the surrounding lines.

Indent with spaces, not tabs. Do not put any tab characters anywhere in your code. You would do well to stop pressing the tab key entirely.

When a line opens a block, the next line starts 2 spaces further in than the line that opened:

Good:

```
if (condition) {

  action();

}
```

Bad:

```
if (condition) {

action();

}
```

When a line closes a block, that line starts at the same level as the line that opened the block

Good:

```
if (condition) {

  action();

}
```

Bad:

```
if (condition) {

  action();

  }
```

No two lines should ever have more or less than 2 (or 4, depending on who you ask) spaces difference in their indentation. Any number of mistakes in the above rules could lead to this, but one example would be:

Bad:

```
transmogrify({

  a: {

    b: function(){

    }

}});
```

Better:

```
transmogrify({

  a: {
```

```
    b: function(){}

  }

});
```

Variable declaration

Use a new line for each variable declaration.

Use a new `var` statement for each line you declare a variable on.

Good:

```
var ape;

var bat;
```

Bad:

```
var cat,

    dog
```

Bad:

```
var eel, fly;
```

Naming

Variable names

A single descriptive word is best. "Descriptive" preferably means it is directly describing some noun in the "domain" of your problem. Name your variables after their purpose, not their structure.

Good:

```
var animals = ['cat', 'dog', 'fish'];
```

Bad:

```
var targetInputs = ['cat', 'dog', 'fish'];
```

Bad:

```
var array = ['cat', 'dog', 'fish'];
```

Collections such as arrays and maps should have plural noun variable names.

Good:

```
var animals = ['cat', 'dog', 'fish'];
```

Bad:

```
var animalList = ['cat', 'dog', 'fish'];
```

Bad:

```
var animal = ['cat', 'dog', 'fish'];
```

Boolean names

Variables that represent boolean values should be named accordingly. Since a boolean either is true or false, typically you prefix the variable name with `is` or `are`.

E.g., `isValid` or `areAvailable`.

Good:

```
var areEqual = true;
```

Bad:

```
var pass = true;
```

Function names

Function names should start with a verb -- in the form "{verbObject}", like "calculateTotal" or "listInventory" etc. That makes it self-describing, telling the code reader at a glance what the function's input and output are as well as a summary of the processing that transforms the former to the latter.

Bad:

```
var waterBlocks = function() {

  // count how many blocks of water are collected between each tower

}
```

Good:

```
var countWaterBlocks = function() {

  // do stuff

}
```

Good:

```
var countWaterBlocksBetweenTowers = function() {

  // do stuff

}
```

Capital letters in variable names

- Most people choose to use capitalization of the first letter in their variable names to indicate that they contain a class .
- Optionally, some people use a capital letter only on functions that are written to be run with the keyword `new`.
- Use all-caps for constant variables (ones that will not change throughout the life of your program).

```
// Example of a capitalized class constructor function name.

function Animal() {



}
```

```
// Example of an all-caps constant variable name.

const MAX_ITEMS_IN_QUEUE = 100;
```

Symbols / punctuation

Don't omit braces (even if you can)

Never omit braces for statement blocks (although they are sometimes optional).

Good:

```
for (key in object) {

  alert(key);

}
```

Bad:

```
for (key in object)

  alert(key);
```

Quoting

Having a standard of any sort is preferable to a mix-and-match approach.

Good:

```
var firstAnimal = 'dog';

var secondAnimal = 'cat';
```

Good:

```
var firstAnimal = "dog";

var secondAnimal = "cat";
```

Bad:

```
var firstAnimal = 'dog';

var secondAnimal = "cat";
```

Semicolons

Don't forget semicolons at the end of lines.

Good:

```
alert('hi');
```

Bad:

```
alert('hi')
```

Semicolons are not required at the end of statements that include a block--i.e. `if`, `for`, `while`, etc.

Good:

```
if (condition) {

  response();

}
```

Bad:

```
if (condition) {

  response();

};
```

A function may be used at the end of a normal assignment statement, and would require a semicolon (even though it looks rather like the end of some statement block).

Good:

```
var greet = function () {

  alert('hi');

};
```

Bad:

```
var greet = function () {

  alert('hi');

}
```

Operators and keywords

Use strict comparison operators

Always use `===` and `!==`, since `==` and `!=` will automatically convert types in ways you're unlikely to expect.

Good:

```
// this comparison evaluates to false, because the number zero is not the same as the empty string.

if (0 === '') {

  alert('looks like they\'re equal');

}
```

Bad:

```
// This comparison evaluates to true, because after type coercion, zero and the empty string are equal.

if (0 == '') {
```

```
  alert('looks like they\'re equal');

}
```

Use of the ternary operator (x ? y : z)

The ternary operator is the form:

```
x ? y : z;
```

It evaluates to `y` if `x` is true, otherwise it evaluates to `z`.

The ternary operator makes for compact code, but it can be hard to read.

Which of the following do you think is easier to read?

Uses ternary:

```
return (actual === expected) ? 'passed' : 'FAILED ['+ testName + '] Expected
"'+expected+'",but got '+'"'+actual+'"';
```

Uses simple if-statement:

```
if (actual === expected) {

  return 'passed';

} else {

  return 'FAILED ' + testName + ': Expected ' + expected + ', but got ' + actual;
```

```
}
```

Only use the ternary operator if it's extremely clear and short to do so. Don't just use it to be clever.

Use of the not-operator (!)

The idiom is to keep the not-operator right next to the item it is negating:

Bad:

```
if (! isEqual) {



}
```

Good:

```
if (!isEqual) {



}
```

Flow control

Switch statements

Avoid use of `switch` statements altogether.

They are prone to error due to missing `break` statements. See this article for more detail.

Brevity

Write the least code you can that is still completely clear.

Not as good:

```
function square(n) {

  var squaredN = n * n;

  return squaredN;

}
```

Good:

```
function square(n) {

  return n * n;

}
```

The principle is to write just enough code to be self-describing. The aim is for any reasonable code reader to be able to glance at a given piece of code and understand what it's doing, with no comments, without asking anybody for an explanation, and without having to pore over the surrounding code for clues.

Avoid negation if possible

Whenever you find yourself with lots of negation, you might have an opportunity to increase clarity by converting it to positive.

A bit confusing to work out:

```
if (!equalSizes || !equalValues) {

  // negative outcome

} else {

  // positive outcome

}
```

More straightforward:

```
if (equalSizes && equalValues) {

  // positive outcome

} else {

  // negative outcome

}
```

Return boolean results directly

Whenever you find yourself returning true or false from within a conditional, you might have an opportunity for brevity.

Verbose:

```
if (charSet.size < text.length) {

  return true;

} else {

  return false;

}
```

Concise:

```
return charSet.size < text.length;
```

Spacing

Code density

- Conserve vertical space by minimizing the number of lines you write. The more concisely your code is written, the more context can be seen in one screen.

Good:

```
function square(n) {

  return n * n;

}
```

```
function assertEqual(actual, expected, testName) {

  // compare actual and expected

}
```

Bad:

```
function square(n) {

  return n * n;

}
```

```
function assertEqual(actual, expected, testName) {

  // compare actual and expected

}
```

- Conserve line length by minimizing the amount of complexity you put on each line. Long lines are difficult to read. Try to make it easily read in one glance.

- This goal is in conflict with the line quantity goal, so you must do your best to balance them.

  ### Padding & additional whitespace

  Generally, we don't care where you put extra spaces, provided they are not distracting.

  You may use it as padding for visual clarity. If you do though, make sure it's balanced on both sides.

  Good:

  ```
  alert('I chose to put no visual padding around this string');
  ```

  Good:

  ```
  alert( 'I chose to put visual padding around this string' );
  ```

  Bad:

  ```
  alert( 'I only put visual padding on one side of this string');
  ```

  You may use it to align two similar lines, but it is not recommended. This pattern usually leads to unnecessary edits of many lines in your code every time you change a variable name.

  ```
  // discouraged:

  var firstItem  = getFirst();

  var secondItem = getSecond();
  ```

```
// encouraged:

var firstItem = getFirst();

var secondItem = getSecond();
```

Put `else` and `else if` statements on the same line as the ending curly brace for the preceding `if` block

Good:

```
if (condition) {

  response();

} else {

  otherResponse();

}
```

Bad:

```
if (condition) {

  response();

}
```

```
else {

  otherResponse();

}
```

Spacing between commas

Bad:

```
assertEqual(Math.pow(3,2),9,'Math.pow squares properly');
```

Good:

```
assertEqual(Math.pow(3, 2), 9, 'Math.pow squares properly');
```

Spacing around operators

Bad:

```
'Failed ['+testName+']'...
```

Good:

```
'Failed [' + testName + ']'...
```

Bad:

```
if(actual===excepted){

  // action
```

```
}else{

  // alternate action

}
```

Good:

```
if (actual === expected) {

  // action

} else {

  // alternate action

}
```

Code comments

- Code comments are usually less effective than good variable names and function names.
- Having to write comments suggests that your code structure and naming are not sufficient to convey the "story" (the flow of data and processing) of your code by themselves. That's a bad sign.
- Comments make a file longer and drift out of sync with the code they annotate.
- If you do comment, then comment on the purpose of the code (the "why"), not the mechanics of implementing it (the "how").

- Do not leave stray comments (aka "cruft", or leftover junk) sitting in your code. Just delete unnecessary / outdated / temporary-scaffolding type of comments.

Avoid for...in for Arrays

Do not use `for...in` statements with the intent of iterating over a list of numeric keys. Use a generic for loop instead.

Good:

```
var letters = ['a', 'b', 'c']

for (var i = 0; i < letters.length; i++) {

  alert(list[i]);

}
```

Bad:

```
var letters = ['a', 'b', 'c']

for (var i in letters) {

  alert(list[i]);

}
```

Snake vs. Camel Casing

In Javascript, the convention is generally to use 'Camel Casing' when naming variables. This is opposed to the convention called 'Snake

Casing', which is used in other programming languages like Python or Ruby.

Good:

```
var camelCased = 'Used in javascript';
```

Bad:

```
var snake_cased = 'Used in other languages, but also sometimes in JS';
```

Under certain circumstances, you will see people use snake-casing for constants in Javascript.

```
const MAX_ITEMS_IN_QUEUE = 100;
```