原本

Airbnb JavaScript スタイルガイド

形(Types)

- 1.1 **Primitives**: When you access a primitive type you work directly on its value.
- 1.1 **Primitives**: primitive typeはその値を直接変えます。
  - `string`
  - `number`
  - `boolean`
  - `null`
  - `undefined`

```
 const foo = 1;
let bar = foo;

bar = 9;

console.log(foo, bar); // => 1, 9
```

- 1.2 **Complex**: When you access a complex type you work on a reference to its value.
- 1.2 **参照形**: 参照形(Complex)は参照で値を変えます。
  - `object`
  - `array`
  - `function`

```
 const foo = [1, 2];
const bar = foo;

bar[0] = 9;

console.log(foo[0], bar[0]); // => 9, 9
```

参照(References)

- 2.1 Use `const` for all of your references; avoid using `var`.
- 2.1 全参照は `const` を使います。 `var` は使いません

> Why? This ensures that you can't reassign your references, which can lead to bugs and difficult to comprehend code.
>
> なぜ？参照をまた割り当てすることはできないので、bugを起こす恐れがある。

```
 // bad
var a = 1;
var b = 2;

// good
const a = 1;
const b = 2;
```

- 2.2 If you must reassign references, use `let` instead of `var`.
- 2.2 参照をまた割り当てしたい時は `var` ではなく `let` を使ってください。

> Why? `let` is block-scoped rather than function-scoped like `var`.
>
> なぜ? `var` は関数scope `let` はブロックscope

```
// bad
var count = 1;
if (true) {
  count += 1;
}

// good, use the let.
let count = 1;
if (true) {
  count += 1;
}
```

- 2.3 Note that both `let` and `const` are block-scoped.
- 2.3 `let` と `const` はブロックscopeです。

```
// const and let only exist in the blocks they are defined in.
// const と let はブロックの中でしか存在しません。
{
  let a = 1;
  const b = 1;
}
console.log(a); // ReferenceError
console.log(b); // ReferenceError
```

オブジェクト(Objects)

- 3.1 Use the literal syntax for object creation.
- 3.1 オブジェクトを 作成する時は, literalを使ってください。

```
// bad
const item = new Object();

// good
const item = {};
```

- 3.2 If your code will be executed in browsers in script context, don't use reserved words as keys. It won't work in IE8. More info. It's OK to use them in ES6 modules and server-side code.
- 3.2 codeがbrowserのscriptとして実行される時は予約語を使わないでください。. IE8で動きません。

```
// bad
const superman = {
  default: { clark: 'kent' },
  private: true,
};

// good
const superman = {
  defaults: { clark: 'kent' },
  hidden: true,
};
```

- 3.3 Use readable synonyms in place of reserved words.
- 3.3 予約語を避けて、同義語を使ってください。

```
// bad
const superman = {
  class: 'alien',
};

// bad
const superman = {
  klass: 'alien',
};

// good
const superman = {
  type: 'alien',
};
```

- 3.4 Use computed property names when creating objects with dynamic property names.
- 3.4 動的propertyを持つオブジェクトを作成するときはcomputed property namesを使ってください。

> Why? They allow you to define all the properties of an object in one place.
>
> なぜ? オブジェクトの全propertyを一か所で定義できる。

```
function getKey(k) {
  return a `key named ${k}`;
}

// bad
const obj = {
  id: 5,
  name: 'San Francisco',
};
obj[getKey('enabled')] = true;

// good
const obj = {
  id: 5,
  name: 'San Francisco',
  [getKey('enabled')]: true
};
```

- 3.5 Use object method shorthand.
- 3.5 methodの略を使ってください。

```
// bad
const atom = {
  value: 1,

  addValue: function (value) {
    return atom.value + value;
  },
};

// good
const atom = {
  value: 1,

  addValue(value) {
    return atom.value + value;
  },
};
```

- 3.6 Use property value shorthand.
- 3.6 propertyの略を使ってください。

> Why? It is shorter to write and descriptive.
>
> なぜ? 記述と説明が簡単になります。

```
const lukeSkywalker = 'Luke Skywalker';

// bad
const obj = {
  lukeSkywalker: lukeSkywalker,
};

// good
const obj = {
  lukeSkywalker,
};
```

- 3.7 Group your shorthand properties at the beginning of your object declaration.
- 3.7 property略文は オブジェクト 宣言の始まりにグループ化してください。

> Why? It's easier to tell which properties are using the shorthand.
>
> なぜ? どの propertyが略文を使っているのか分かりやすくなります。

```
const anakinSkywalker = 'Anakin Skywalker';
const lukeSkywalker = 'Luke Skywalker';

// bad
const obj = {
  episodeOne: 1,
  twoJediWalkIntoACantina: 2,
  lukeSkywalker,
  episodeThree: 3,
  mayTheFourth: 4,
  anakinSkywalker,
};

// good
const obj = {
  lukeSkywalker,
  anakinSkywalker,
  episodeOne: 1,
  twoJediWalkIntoACantina: 2,
  episodeThree: 3,
  mayTheFourth: 4,
};
```

配列(Arrays)

- 4.1 Use the literal syntax for array creation.
- 4.1 配列を作成するときはliteralで作成してください。

```
// bad
const items = new Array();

// good
const items = [];
```

- 4.2 Use Array#push instead of direct assignment to add items to an array.
- 4.2 直接 配列に代入するのではなくArray pushを使ってください。

```
const someStack = [];

// bad
someStack[someStack.length] = 'abracadabra';

// good
someStack.push('abracadabra');
```

- 4.3 Use array spreads `...` to copy arrays.
- 4.3 配列をコピーする時は `...` を使ってください。

```
// bad
const len = items.length;
const itemsCopy = [];
let i;

for (i = 0; i < len; i++) {
  itemsCopy[i] = items[i];
}

// good
const itemsCopy = [...items];
```

- 4.4 To convert an array-like object to an array, use Array#from.
- 4.4 array-like オブジェクトを 配列に変換する時は Array#fromを使ってください。

```
const foo = document.querySelectorAll('.foo');
const nodes = Array.from(foo);
```

構造化代入(Destructuring)

- 5.1 Use object destructuring when accessing and using multiple properties of an object.
- 5.1 一個のオブジェクトから複数のpropertyをアクセスする時はオブジェクト 構造化代入を使ってください。

> Why? Destructuring saves you from creating temporary references for those properties.
>
> なぜ? 構造化代入を利用することで propertyのための臨時的な参照作成が防げます。

```
// bad
function getFullName(user) {
  const firstName = user.firstName;
  const lastName = user.lastName;

  return `${firstName} ${lastName}`;
}

// good
function getFullName(obj) {
  const { firstName, lastName } = obj;
  return `${firstName} ${lastName}`;
}

// best
function getFullName({ firstName, lastName }) {
  return `${firstName} ${lastName}`;
}
```

- 5.2 Use array destructuring.
- 5.2 配列の 構造化代入を使ってください。

```
const arr = [1, 2, 3, 4];

// bad
const first = arr[0];
const second = arr[1];

// good
const [first, second] = arr;
```

- 5.3 Use object destructuring for multiple return values, not array destructuring.
- 5.3 複数の値をreturnする時は 配列の 構造化代入ではなく オブジェクトの 構造化代入を使ってください。

```
// bad
function processInput(input) {
  // then a miracle occurs
  return [left, right, top, bottom];
}

// the caller needs to think about the order of return data
const [left, __, top] = processInput(input);

// good
function processInput(input) {
  // then a miracle occurs
  return { left, right, top, bottom };
}

// the caller selects only the data they need
const { left, right } = processInput(input);
```

文字列(Strings)

- 6.1 Use single quotes `''` for strings.
- 6.1 文字列には `''` を使ってください。

```
// bad
const name = "Capt. Janeway";

// good
const name = 'Capt. Janeway';
```

- 6.2 Strings longer than 100 characters should be written across multiple lines using string concatenation.
- 6.2 100文字以上の 文字列は 文字列連結で複数行で記述してください。

```
// bad
const errorMessage = 'This is a super long error that was thrown because of Batman. When you stop to think about how Batman ha

// bad
const errorMessage = 'This is a super long error that was thrown because \
of Batman. When you stop to think about how Batman had anything to do \
with this, you would get nowhere \
fast.';

// good
const errorMessage = 'This is a super long error that was thrown because ' +
  'of Batman. When you stop to think about how Batman had anything to do ' +
  'with this, you would get nowhere fast.';
```

- 6.4 When programmatically building up strings, use template strings instead of concatenation.
- 6.4 programで 文字列を作る時は文字列連結ではなくtemplate stringsを使ってください。

```
// bad
function sayHi(name) {
  return 'How are you, ' + name + '?';
}

// bad
function sayHi(name) {
  return ['How are you, ', name, '?'].join();
}

// good
function sayHi(name) {
  return `How are you, ${name}?`;
}
```

- 6.5 Never use `eval()` on a string, it opens too many vulnerabilities.
- 6.5 絶対に `eval()` を使わないでください。

関数(Functions)

- 7.1 Use function declarations instead of function expressions.
- 7.1 関数式より関数宣言を使ってください。

```
// bad
const foo = function () {
};

// good
function foo() {
}
```

- 7.2 Function expressions:
- 7.2 関数式

```
// immediately-invoked function expression (IIFE)
(() => {
  console.log('Welcome to the Internet. Please follow me.');
})();
```

- 7.3 Never declare a function in a non-function block (if, while, etc). Assign the function to a variable instead. Browsers will allow you to do it, but they all interpret it differently, which is bad news bears.
- 7.3 関数以外のブロックで (if、 whileなど)関数を宣言しないでください。 browserごとにそれの解釈がことなります。

```
// bad
if (currentUser) {
  function test() {
    console.log('Nope.');
  }
}

// good
let test;
if (currentUser) {
  test = () => {
    console.log('Yup.');
  };
}
```

- 7.5 Never name a parameter `arguments`. This will take precedence over the `arguments` object that is given to every function scope.
- 7.5 パラメータに `arguments` を使わないでください。

```
// bad
function nope(name, options, arguments) {
  // ...stuff...
}

// good
function yup(name, options, args) {
  // ...stuff...
}
```

- 7.7 Use default parameter syntax rather than mutating function arguments.
- 7.7 関数のパラメータの変換より default パラメータを使ってください。

```
// really bad
function handleThings(opts) {
  opts = opts || {};
  // ...
}

// still bad
function handleThings(opts) {
  if (opts === void 0) {
    opts = {};
  }
  // ...
}

// good
function handleThings(opts = {}) {
  // ...
}
```

- 7.8 Avoid side effects with default parameters.
- 7.8 side effectのあるdefault パラメータは使わないでください。

```
var b = 1;
// bad
function count(a = b++) {
  console.log(a);
}
count();  // 1
count();  // 2
count(3); // 3
count();  // 3
```

- 7.9 Always put default parameters last.
- 7.9 必ず default パラメータは後ろにおいてください。

```
// bad
function handleThings(opts = {}, name) {
  // ...
}

// good
function handleThings(name, opts = {}) {
  // ...
}
```

  - 7.10 Never use the Function constructor to create a new function.
  - 7.10 new Function constructorを使わないでください。

```
// bad
var add = new Function('a', 'b', 'return a + b');

// still bad
var subtract = Function('a', 'b', 'return a - b');
```

Arrow関数(Arrow Functions)

- 8.1 When you must use function expressions (as when passing an anonymous function), use arrow function notation.
- 8.1 (無名関数のような)関数式の場合はarrow関数 表記を使ってください。

```
// bad
[1, 2, 3].map(function (x) {
  const y = x + 1;
  return x * y;
});

// good
[1, 2, 3].map((x) => {
  const y = x + 1;
  return x * y;
});
```

- 8.2 If the function body consists of a single expression, feel free to omit the braces and use the implicit return. Otherwise use a `return` statement.
- 8.2 関数の本体が一つの式である場合は{}を略し暗示的な returnを使ってください。

```
// good
[1, 2, 3].map(number => `A string containing the ${number}.`);

// bad
[1, 2, 3].map(number => {
  const nextNumber = number + 1;
  `A string containing the ${nextNumber}.`;
});

// good
[1, 2, 3].map(number => {
  const nextNumber = number + 1;
  return `A string containing the ${nextNumber}.`;
});
```

- 8.3 In case the expression spans over multiple lines, wrap it in parentheses for better readability.
- 8.3 複数行の式は()で囲んでください。

```
// bad
[1, 2, 3].map(number => 'As time went by, the string containing the ' +
  `${number} became much longer. So we needed to break it over multiple ` +
  'lines.'
);

// good
[1, 2, 3].map(number => (
  `As time went by, the string containing the ${number} became much ` +
  'longer. So we needed to break it over multiple lines.'
));
```

- 8.4 If your function only takes a single argument, feel free to omit the parentheses.
- 8.4 関数の引数が一つの場合は()を略してください。

```
// good
[1, 2, 3].map(x => x * x);

// good
[1, 2, 3].reduce((y, x) => x + y);
```

Classes & Constructors

- 9.1 Always use `class`. Avoid manipulating `prototype` directly.
- 9.1 `prototype` を使わないでください。 `class` を使ってください。

```
 // bad
function Queue(contents = []) {
  this._queue = [...contents];
}
Queue.prototype.pop = function() {
  const value = this._queue[0];
  this._queue.splice(0, 1);
  return value;
}

// good
class Queue {
  constructor(contents = []) {
    this._queue = [...contents];
  }
  pop() {
    const value = this._queue[0];
    this._queue.splice(0, 1);
    return value;
  }
}
```

- 9.2 Use `extends` for inheritance.
- 9.2 継承は `extends` を使ってください。

```
 // bad
const inherits = require('inherits');
function PeekableQueue(contents) {
  Queue.apply(this, contents);
}
inherits(PeekableQueue, Queue);
PeekableQueue.prototype.peek = function() {
  return this._queue[0];
}

// good
class PeekableQueue extends Queue {
  peek() {
    return this._queue[0];
  }
}
```

- 9.3 Methods can return `this` to help with method chaining.
- 9.3 `this` をリターンすることでmethod chainingが可能となります。

```
// bad
Jedi.prototype.jump = function() {
  this.jumping = true;
  return true;
};

Jedi.prototype.setHeight = function(height) {
  this.height = height;
};

const luke = new Jedi();
luke.jump(); // => true
luke.setHeight(20); // => undefined

// good
class Jedi {
  jump() {
    this.jumping = true;
    return this;
  }

  setHeight(height) {
    this.height = height;
    return this;
  }
}

const luke = new Jedi();

luke.jump()
  .setHeight(20);
```

モジュール(Modules)

- 10.1 Always use modules (`import` / `export`) over a non-standard module system. You can always transpile to your preferred module system.
- 10.1 `import` / `export` を使ってください。 MODULEには未来があります。

```
// bad
const AirbnbStyleGuide = require('./AirbnbStyleGuide');
module.exports = AirbnbStyleGuide.es6;

// ok
import AirbnbStyleGuide from './AirbnbStyleGuide';
export default AirbnbStyleGuide.es6;

// best
import { es6 } from './AirbnbStyleGuide';
export default es6;
```

- 10.2 Do not use wildcard imports.
- 10.2 wildcard import は使わないでください。

```
// bad
import * as AirbnbStyleGuide from './AirbnbStyleGuide';

// good
import AirbnbStyleGuide from './AirbnbStyleGuide';
```

- 10.3 And do not export directly from an import.
- 10.3 import から直接 export しないでください。

```
// bad
// filename es6.js
export { es6 as default } from './airbnbStyleGuide';

// good
// filename es6.js
import { es6 } from './AirbnbStyleGuide';
export default es6;
```

property(Properties)

- 12.1 Use dot notation when accessing properties.
- 12.1 propertyにアクセスする時は `.` を使ってください。

```
const luke = {
  jedi: true,
  age: 28,
};

// bad
const isJedi = luke['jedi'];

// good
const isJedi = luke.jedi;
```

- 12.2 Use subscript notation `[]` when accessing properties with a variable.
- 12.2 変数でpropertyにアクセスする時は `[]` を使ってください。

```
const luke = {
  jedi: true,
  age: 28,
};

function getProp(prop) {
  return luke[prop];
}

const isJedi = getProp('jedi');
```

変数(Variables)

- 13.1 Always use `const` to declare variables. Not doing so will result in global variables. We want to avoid polluting the global namespace. Captain Planet warned us of that.
- 13.1 変数宣言 `const` を使ってください。

```
// bad
superPower = new SuperPower();

// good
const superPower = new SuperPower();
```

- 13.2 Use one `const` declaration per variable.
- 13.2 各 `const` を使ってください。

```
// bad
const items = getItems(),
    goSportsTeam = true,
    dragonball = 'z';

// bad
// (compare to above, and try to spot the mistake)
const items = getItems(),
    goSportsTeam = true;
    dragonball = 'z';

// good
const items = getItems();
const goSportsTeam = true;
const dragonball = 'z';
```

- 13.3 Group all your `const` s and then group all your `let` s.
- 13.3 まず `const` をグループして `let` をグループしてください。

```
// bad
let i, len, dragonball,
    items = getItems(),
    goSportsTeam = true;

// bad
let i;
const items = getItems();
let dragonball;
const goSportsTeam = true;
let len;

// good
const goSportsTeam = true;
const items = getItems();
let dragonball;
let i;
let length;
```

条件(Comparison Operators & Equality)

- 15.1 Use `===` and `!==` over `==` and `!=`.
- 15.1 `==` 、 `!=` より `===` と `!==` を使ってください。
- 15.3 Use shortcuts.
- 15.3 短縮した形で表記してください。

```
// bad
if (name !== '') {
  // ...stuff...
}

// good
if (name) {
  // ...stuff...
}

// bad
if (collection.length > 0) {
  // ...stuff...
}

// good
if (collection.length) {
  // ...stuff...
}
```

ブロック(Blocks)

- 16.2 If you're using multi-line blocks with `if` and `else`, put `else` on the same line as your `if` block's closing brace.
- 16.2 複数行 ブロックの `if` と `else` は下記の通りと書きます

  ```
  // bad
  if (test) {
    thing1();
    thing2();
  }
  else {
    thing3();
  }

  // good
  if (test) {
    thing1();
    thing2();
  } else {
    thing3();
  }
  ```

コメント(Comments)

- 17.1 Use `/** ... */` for multi-line comments. Include a description, specify types and values for all parameters and return values.
- 17.1 複数行のコメントは `/** ... */` を使ってください。

  ```
  // bad
  // make() returns a new element
  // based on the passed in tag name
  //
  // @param {String} tag
  // @return {Element} element
  function make(tag) {

    // ...stuff...

    return element;
  }

  // good
  /**
   * make() returns a new element
   * based on the passed in tag name
   *
   * @param {String} tag
   * @return {Element} element
   */
  function make(tag) {

    // ...stuff...

    return element;
  }
  ```

- 17.2 Use `//` for single line comments. Place single line comments on a newline above the subject of the comment. Put an empty line before the comment unless it's on the first line of a block.
- 17.2 一行の コメントには `//` を使ってください。そしてコメントの前は一行あけます。

```
// bad
const active = true;  // is current tab

// good
// is current tab
const active = true;

// bad
function getType() {
  console.log('fetching type...');
  // set the default type to 'no type'
  const type = this._type || 'no type';

  return type;
}

// good
function getType() {
  console.log('fetching type...');

  // set the default type to 'no type'
  const type = this._type || 'no type';

  return type;
}

// also good
function getType() {
  // set the default type to 'no type'
  const type = this._type || 'no type';

  return type;
}
```

空白(Whitespace)

- 18.1 Use soft tabs set to 2 spaces.
- 18.1 TABはスペース 2 個にしてください。

```
// bad
function() {
∙∙∙∙const name;
}

// bad
function() {
∙const name;
}

// good
function() {
∙∙const name;
}
```

- 18.2 Place 1 space before the leading brace.
- 18.2 {}前にはスペース一個を入れてください。

```
// bad
function test(){
  console.log('test');
}

// good
function test() {
  console.log('test');
}

// bad
dog.set('attr',{
  age: '1 year',
  breed: 'Bernese Mountain Dog',
});

// good
dog.set('attr', {
  age: '1 year',
  breed: 'Bernese Mountain Dog',
});
```

```
// bad
if(isJedi) {
  fight ();
}

// good
if (isJedi) {
  fight();
}

// bad
function fight () {
  console.log ('Swooosh!');
}

// good
function fight() {
  console.log('Swooosh!');
}
```

- 18.4 Set off operators with spaces.
- 18.4 演算子の間はスペースを使ってください。

```
// bad
const x=y+5;

// good
const x = y + 5;
```

- 18.5 End files with a single newline character.
- 18.5 EOLには一行あけます。

```
// bad
(function(global) {
  // ...stuff...
})(this);
```

```
// bad
(function(global) {
  // ...stuff...
})(this);↵
↵
```

```
// good
(function(global) {
  // ...stuff...
})(this);↵
```

- 18.6 Use indentation when making long method chains. Use a leading dot, which emphasizes that the line is a method call, not a new statement.
- 18.6 長いMethod chainingの場合INDENTを使ってください。

```
// bad
$('#items').find('.selected').highlight().end().find('.open').updateCount();

// bad
$('#items').
  find('.selected').
    highlight().
    end().
  find('.open').
    updateCount();

// good
$('#items')
  .find('.selected')
    .highlight()
    .end()
  .find('.open')
    .updateCount();

// bad
const leds = stage.selectAll('.led').data(data).enter().append('svg:svg').class('led', true)
    .attr('width', (radius + margin) * 2).append('svg:g')
    .attr('transform', 'translate(' + (radius + margin) + ',' + (radius + margin) + ')')
    .call(tron.led);

// good
const leds = stage.selectAll('.led')
    .data(data)
  .enter().append('svg:svg')
    .classed('led', true)
    .attr('width', (radius + margin) * 2)
  .append('svg:g')
    .attr('transform', 'translate(' + (radius + margin) + ',' + (radius + margin) + ')')
    .call(tron.led);
```

- 18.7 Leave a blank line after blocks and before the next statement.
- 18.7 文のブロックの後に一行あけます。

```
// bad
if (foo) {
  return bar;
}
return baz;

// good
if (foo) {
  return bar;
}

return baz;

// bad
const obj = {
  foo() {
  },
  bar() {
  },
};
return obj;

// good
const obj = {
  foo() {
  },

  bar() {
  },
};

return obj;

// bad
const arr = [
  function foo() {
  },
  function bar() {
  },
];
return arr;

// good
const arr = [
  function foo() {
  },

  function bar() {
  },
];

return arr;
```

- 18.8 Do not pad your blocks with blank lines.
- 18.8 ブロックに空の行をあけないでください。

```
// bad
function bar() {

  console.log(foo);

}

// also bad
if (baz) {

  console.log(qux);
} else {
  console.log(foo);

}

// good
function bar() {
  console.log(foo);
}

// good
if (baz) {
  console.log(qux);
} else {
  console.log(foo);
}
```

- 18.9 Do not add spaces inside parentheses.
- 18.9 ()の中にはスペースはお控えください。

```
// bad
function bar( foo ) {
  return foo;
}

// good
function bar(foo) {
  return foo;
}

// bad
if ( foo ) {
  console.log(foo);
}

// good
if (foo) {
  console.log(foo);
}
```

- 18.10 Do not add spaces inside brackets.
- 18.10 [] の中にスペースを使わないでください。

```
// bad
const foo = [ 1, 2, 3 ];
console.log(foo[ 0 ]);

// good
const foo = [1, 2, 3];
console.log(foo[0]);
```

- 18.11 Add spaces inside curly braces.
- 18.11 {}の中にスペースを入れてください。

```
// bad
const foo = {clark: 'kent'};

// good
const foo = { clark: 'kent' };
```

Commas

- 19.1 Leading commas: **Nope.**
- 19.1 下記参考

```
// bad
const story = [
    once
  , upon
  , aTime
];

// good
const story = [
  once,
  upon,
  aTime,
];

// bad
const hero = {
    firstName: 'Ada'
  , lastName: 'Lovelace'
  , birthYear: 1815
  , superPower: 'computers'
};

// good
const hero = {
  firstName: 'Ada',
  lastName: 'Lovelace',
  birthYear: 1815,
  superPower: 'computers',
};
```

- 19.2 Additional trailing comma: **Yup.**
- 19.2 最後は **使ってください。**

```
// bad - git diff without trailing comma
const hero = {
     firstName: 'Florence',
-    lastName: 'Nightingale'
+    lastName: 'Nightingale',
+    inventorOf: ['coxcomb graph', 'modern nursing']
};

// good - git diff with trailing comma
const hero = {
     firstName: 'Florence',
     lastName: 'Nightingale',
+    inventorOf: ['coxcomb chart', 'modern nursing'],
};

// bad
const hero = {
  firstName: 'Dana',
  lastName: 'Scully'
};

const heroes = [
  'Batman',
  'Superman'
];

// good
const hero = {
  firstName: 'Dana',
  lastName: 'Scully',
};

const heroes = [
  'Batman',
  'Superman',
];
```

セミコロン(Semicolons)

- 20.1 **Yup.**
- 20.1 **使います**

```
// bad
(function() {
  const name = 'Skywalker'
  return name
})()

// good
(() => {
  const name = 'Skywalker';
  return name;
})();

;(() => {
  const name = 'Skywalker';
  return name;
})();
```

形変換と強制(Type Casting & Coercion)

- 21.1 Perform type coercion at the beginning of the statement.
- 21.1 文の頭で形変換を行います。
- 21.2 Strings:

- 21.2 文字列の場合:

```
 //  => this.reviewScore = 9;

// bad
const totalScore = this.reviewScore + '';

// good
const totalScore = String(this.reviewScore);
```

- 21.3 Numbers: Use `Number` for type casting and `parseInt` always with a radix for parsing strings.
- 21.3 数の場合: `Number` に変換するときは `parseInt` を使います

```
 const inputValue = '4';

// bad
const val = new Number(inputValue);

// bad
const val = +inputValue;

// bad
const val = inputValue >> 0;

// bad
const val = parseInt(inputValue);

// good
const val = Number(inputValue);

// good
const val = parseInt(inputValue, 10);
```

命名法(Naming Conventions)

- 22.1 Avoid single letter names. Be descriptive with your naming.
- 22.1 1文字の名前はお控えください。名前から意図が読めるようにしてください。

```
 // bad
function q() {
  // ...stuff...
}

// good
function query() {
  // ..stuff..
}
```

- 22.2 Use camelCase when naming objects, functions, and instances.
- 22.2 オブジェクト, 関数 そしてINSTANCEには camelCaseを使ってください。

```
 // bad
const OBJEcttsssss = {};
const this_is_my_object = {};
function c() {}

// good
const thisIsMyObject = {};
function thisIsMyFunction() {}
```

- 22.3 Use PascalCase when naming constructors or classes.
- 22.3 クラスや constructorには PascalCase を使ってください。

```
// bad
function user(options) {
  this.name = options.name;
}

const bad = new user({
  name: 'nope',
});

// good
class User {
  constructor(options) {
    this.name = options.name;
  }
}

const good = new User({
  name: 'yup',
});
```

- 22.4 Use a leading underscore `_` when naming private properties.
- 22.4 private propertyは前に `_` を使ってください。

```
// bad
this.__firstName__ = 'Panda';
this.firstName_ = 'Panda';

// good
this._firstName = 'Panda';
```

- 22.5 Don't save references to `this`. Use arrow functions or Function#bind.
- 22.5 `this` の参照を保存するのはお控えください。. arrow関数や Function#bind を使ってください。

```
// bad
function foo() {
  const self = this;
  return function() {
    console.log(self);
  };
}

// bad
function foo() {
  const that = this;
  return function() {
    console.log(that);
  };
}

// good
function foo() {
  return () => {
    console.log(this);
  };
}
```

- 22.6 If your file exports a single class, your filename should be exactly the name of the class.
- 22.6 ファイルを 1このクラスに export する場合ファイル名とクラス名を一致させてください。

```
// file contents
class CheckBox {
  // ...
}
export default CheckBox;

// in some other file
// bad
import CheckBox from './checkBox';

// bad
import CheckBox from './check_box';

// good
import CheckBox from './CheckBox';
```

- 22.7 Use camelCase when you export-default a function. Your filename should be identical to your function's name.
- 22.7 Default exportが 関数の場合, camelCaseを使ってください。 ファイル名と関数名を一致させてください。

```
function makeStyleGuide() {
}

export default makeStyleGuide;
```

- 22.8 Use PascalCase when you export a singleton / function library / bare object.
- 22.8 singleton / function library / 空のオブジェクトを export する場合, PascalCaseを使ってください。

```
const AirbnbStyleGuide = {
  es6: {
  }
};

export default AirbnbStyleGuide;
```

下記からはREFERENCEとなります。

Performance

- On Layout & Web Performance
- String vs Array Concat
- Try/Catch Cost In a Loop
- Bang Function
- jQuery Find vs Context, Selector
- innerHTML vs textContent for script text
- Long String Concatenation
- Loading...

Resources

Learning ES6

- Draft ECMA 2015 (ES6) Spec
- ExploringJS
- ES6 Compatibility Table
- Comprehensive Overview of ES6 Features

Read This

- Standard ECMA-262

Tools

- Code Style Linters
  - ESlint - Airbnb Style .eslintrc
  - JSHint - Airbnb Style .jshintrc
  - JSCS - Airbnb Style Preset

Other Style Guides

- Google JavaScript Style Guide
- jQuery Core Style Guidelines
- Principles of Writing Consistent, Idiomatic JavaScript

Other Styles

- Naming this in nested functions - Christian Johansen
- Conditional Callbacks - Ross Allen
- Popular JavaScript Coding Conventions on Github - JeongHoon Byun
- Multiple var statements in JavaScript, not superfluous - Ben Alman

Further Reading

- Understanding JavaScript Closures - Angus Croll
- Basic JavaScript for the impatient programmer - Dr. Axel Rauschmayer
- You Might Not Need jQuery - Zack Bloom & Adam Schwartz
- ES6 Features - Luke Hoban
- Frontend Guidelines - Benjamin De Cock

Books

- JavaScript: The Good Parts - Douglas Crockford
- JavaScript Patterns - Stoyan Stefanov
- Pro JavaScript Design Patterns - Ross Harmes and Dustin Diaz
- High Performance Web Sites: Essential Knowledge for Front-End Engineers - Steve Souders
- Maintainable JavaScript - Nicholas C. Zakas
- JavaScript Web Applications - Alex MacCaw
- Pro JavaScript Techniques - John Resig
- Smashing Node.js: JavaScript Everywhere - Guillermo Rauch
- Secrets of the JavaScript Ninja - John Resig and Bear Bibeault
- Human JavaScript - Henrik Joreteg
- Superhero.js - Kim Joar Bekkelund, Mads Mobæk, & Olav Bjorkoy
- JSBooks - Julien Bouquillon
- Third Party JavaScript - Ben Vinegar and Anton Kovalyov
- Effective JavaScript: 68 Specific Ways to Harness the Power of JavaScript - David Herman
- Eloquent JavaScript - Marijn Haverbeke
- You Don't Know JS: ES6 & Beyond - Kyle Simpson

Blogs

- DailyJS
- JavaScript Weekly
- JavaScript, JavaScript...
- Bocoup Weblog
- Adequately Good
- NCZOnline
- Perfection Kills
- Ben Alman
- Dmitry Baranovskiy
- Dustin Diaz
- nettuts

Podcasts

- JavaScript Jabber

In the Wild

This is a list of organizations that are using this style guide. Send us a pull request and we'll add you to the list.

- **Aan Zee**: AanZee/javascript
- **Adult Swim**: adult-swim/javascript
- **Airbnb**: airbnb/javascript
- **Apartmint**: apartmint/javascript
- **Avalara**: avalara/javascript
- **Billabong**: billabong/javascript
- **Blendle**: blendle/javascript
- **ComparaOnline**: comparaonline/javascript
- **Compass Learning**: compasslearning/javascript-style-guide
- **DailyMotion**: dailymotion/javascript
- **Digitpaint** digitpaint/javascript
- **Ecosia**: ecosia/javascript
- **Evernote**: evernote/javascript-style-guide
- **ExactTarget**: ExactTarget/javascript
- **Expensify** Expensify/Style-Guide
- **Flexberry**: Flexberry/javascript-style-guide
- **Gawker Media**: gawkermedia/javascript
- **General Electric**: GeneralElectric/javascript
- **GoodData**: gooddata/gdc-js-style
- **Grooveshark**: grooveshark/javascript
- **How About We**: howaboutwe/javascript
- **Huballin**: huballin/javascript
- **HubSpot**: HubSpot/javascript
- **Hyper**: hyperoslo/javascript-playbook
- **InfoJobs**: InfoJobs/JavaScript-Style-Guide
- **Intent Media**: intentmedia/javascript
- **Jam3**: Jam3/JavaScript-Code-Conventions
- **JSSolutions**: JSSolutions/javascript
- **Kinetica Solutions**: kinetica/javascript
- **Mighty Spring**: mightyspring/javascript
- **MinnPost**: MinnPost/javascript
- **MitocGroup**: MitocGroup/javascript
- **ModCloth**: modcloth/javascript

- **Money Advice Service**: moneyadviceservice/javascript
- **Muber**: muber/javascript
- **National Geographic**: natgeo/javascript
- **National Park Service**: nationalparkservice/javascript
- **Nimbl3**: nimbl3/javascript
- **Orion Health**: orionhealth/javascript
- **Peerby**: Peerby/javascript
- **Razorfish**: razorfish/javascript-style-guide
- **reddit**: reddit/styleguide/javascript
- **REI**: reidev/js-style-guide
- **Ripple**: ripple/javascript-style-guide
- **SeekingAlpha**: seekingalpha/javascript-style-guide
- **Shutterfly**: shutterfly/javascript
- **Springload**: springload/javascript
- **StudentSphere**: studentsphere/javascript
- **Target**: target/javascript
- **TheLadders**: TheLadders/javascript
- **T4R Technology**: T4R-Technology/javascript
- **VoxFeed**: VoxFeed/javascript-style-guide
- **Weggo**: Weggo/javascript
- **Zillow**: zillow/javascript
- **ZocDoc**: ZocDoc/javascript

The JavaScript Style Guide Guide

- Reference