원문:https://github.com/airbnb/javascript

Airbnb JavaScript 스타일 가이드() {

JavaScript에 대한 대부분 합리적인 접근 방법

다른 스타일 가이드들

- ES5
- React
- CSS & Sass
- Ruby

목차

형(Types)

- 1.1 **Primitives**: When you access a primitive type you work directly on its value.
- 1.1 **Primitives**: primitive type은 그 값을 직접 조작합니다.
    - `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 **참조형**: 참조형(Complex)은 참조를 통해 값을 조작합니다.
    - `object`
    - `array`
    - `function`

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

bar[0] = 9;

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

↑ back to top

참조(References)

- 2.1 Use `const` for all of your references; avoid using `var`.
- 2.1 모든 참조는 `const` 를 사용하고, `var` 를 사용하지 마십시오.

> Why? This ensures that you can't reassign your references, which can lead to bugs and difficult to comprehend code.

> 왜? 참조를 재할당 할 수 없으므로, 버그로 이어지고 이해하기 어려운 코드가 되는것을 방지합니다.

```
// 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 참조를 재할당 해야한다면 `var` 대신 `let` 을 사용하십시오.

> Why? `let` is block-scoped rather than function-scoped like `var`.

> 왜? `var` 같은 함수스코프 보다는 오히려 블록스코프의 `let`

```
// 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` 과 `const` 는 같이 블록스코프라는것을 유의하십시오.

```
// const and let only exist in the blocks they are defined in.
// const 와 let 은 선언된 블록의 안에서만 존재합니다.
{
  let a = 1;
  const b = 1;
}
console.log(a); // ReferenceError
console.log(b); // ReferenceError
```

↑ back to top

오브젝트(Objects)

- 3.1 Use the literal syntax for object creation.
- 3.1 오브젝트를 작성할때는, 리터럴 구문을 사용하십시오.

```
// 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 코드가 브라우저상의 스크립트로 실행될때 예약어를 키로 이용하지 마십시오. IE8에서 작동하지 않습니다. More info 하지만 ES6 모듈안이나 서버사이드에서는 이용가능합니다.

```
// 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 예약어 대신 알기쉬운 동의어를 사용해 주십시오.

```
// 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 동적 프로퍼티명을 갖는 오브젝트를 작성할때, 계산된 프로퍼티명(computed property names)을 이용해 주십시오.

> Why? They allow you to define all the properties of an object in one place.

> 왜? 오브젝트의 모든 프로퍼티를 한 장소에서 정의 할 수 있습니다.

```
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 메소드의 단축구문을 이용해 주십시오.

```
// 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 프로퍼티의 단축구문을 이용해 주십시오.

> Why? It is shorter to write and descriptive.

> 왜? 기술과 설명이 간결해지기 때문입니다.

```
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 프로퍼티의 단축구문은 오브젝트 선언의 시작부분에 그룹화 해주십시오.

> Why? It's easier to tell which properties are using the shorthand.

> 왜? 어떤 프로퍼티가 단축구문을 이용하고 있는지가 알기쉽기 때문입니다.

```
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,
};
```

↑ back to top

배열(Arrays)

- 4.1 Use the literal syntax for array creation.
- 4.1 배열을 작성 할 때는 리터럴 구문을 사용해 주십시오.

```
// 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 직접 배열에 항목을 대입하지 말고, Array#push를 이용해 주십시오.

```
const someStack = [];

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

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

- 4.3 Use array spreads `...` to copy arrays.
- 4.3 배열을 복사할때는 배열의 확장연산자 `...` 를 이용해 주십시오.

```
// 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 오브젝트를 배열로 변환하는 경우는 Array#from을 이용해 주십시오.

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

↑ back to top

구조화대입(Destructuring)

- 5.1 Use object destructuring when accessing and using multiple properties of an object.
- 5.1 하나의 오브젝트에서 복수의 프로퍼티를 억세스 할 때는 오브젝트 구조화대입을 이용해 주십시오.

> Why? Destructuring saves you from creating temporary references for those properties.

> 왜? 구조화대입을 이용하는 것으로 프로퍼티를 위한 임시적인 참조의 작성을 줄일 수 있습니다.

```
// 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 배열의 구조화대입을 이용해 주십시오.

```
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 복수의 값을 반환하는 경우는 배열의 구조화대입이 아닌 오브젝트의 구조화대입을 이용해 주십시오.

> Why? You can add new properties over time or change the order of things without breaking call sites.

> 왜? 이렇게 함으로써 나중에 호출처에 영향을 주지않고 새로운 프로퍼티를 추가하거나 순서를 변경할수 있습니다.

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

// the caller needs to think about the order of return data
// 호출처에서 반환된 데이터의 순서를 고려할 필요가 있습니다.
const [left, __, top] = processInput(input);

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

// the caller selects only the data they need
// 호출처에서는 필요한 데이터만 선택하면 됩니다.
const { left, right } = processInput(input);
```

↑ back to top

문자열(Strings)

- 6.1 Use single quotes `''` for strings.
- 6.1 문자열에는 싱크쿼트 `''` 를 사용해 주십시오.

```
// 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 100문자 이상의 문자열은 문자열연결을 사용해서 복수행에 걸쳐 기술할 필요가 있습니다.
- 6.3 Note: If overused, long strings with concatenation could impact performance. jsPerf & Discussion.
- 6.3 주의: 문자연결을 과용하면 성능에 영향을 미칠 수 있습니다. jsPerf & Discussion.

```
// 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.';

// 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 프로그램에서 문자열을 생성하는 경우는 문자열 연결이 아닌 template strings를 이용해 주십시오.

> Why? Template strings give you a readable, concise syntax with proper newlines and string interpolation features.

> 왜? Template strings 는 문자열 보간기능과 적절한 줄바꿈 기능을 갖는 간결한 구문으로 가독성이 좋기 때문입니다.

```
// 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 절대로 `eval()` 을 이용하지 마십시오. 이것은 많은 취약점을 만들기 때문입니다.

↑ back to top

함수(Functions)

- 7.1 Use function declarations instead of function expressions.
- 7.1 함수식 보다 함수선언을 이용해 주십시오.

> Why? Function declarations are named, so they're easier to identify in call stacks. Also, the whole body of a function declaration is hoisted, whereas only the reference of a function expression is hoisted. This rule makes it possible to always use Arrow Functions in place of function expressions.

> 왜? 이름이 부여된 함수선언은 콜스택에서 간단하게 확인하는 것이 가능합니다. 또한 함수선언은 함수의 본체가 hoist 되어집니다. 그에 반해 함수식은 참조만이 hoist 되어집니다. 이 룰에 의해 함수식의 부분을 항상 Arrow함수에서 이용하는것이 가능합니다.

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

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

- 7.2 Function expressions:
- 7.2 함수식

```
// immediately-invoked function expression (IIFE)
// 즉시 실행 함수식(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 함수이외의 블록 (if나 while같은) 안에서 함수를 선언하지 마십시오. 변수에 함수를 대입하는 대신 브라우저들은 그것을 허용하지만 모두가 다르게 해석합니다.
- 7.4 **Note:** ECMA-262 defines a `block` as a list of statements. A function declaration is not a statement. Read ECMA-262's note on this issue.

- 7.4 **주의:** ECMA-262 사양에서는 `block` 은 statements의 일람으로 정의되어 있지만 함수선언은 statements 가 아닙니다. Read ECMA-262's note on this issue.

```
// 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 절대 파라메터에 `arguments` 를 지정하지 마십시오. 이것은 함수스코프에 전해지는 `arguments` 오브젝트의 참조를 덮어써 버립니다.

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

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

- 7.6 Never use `arguments`, opt to use rest syntax `...` instead.

- 7.6 절대 `arguments` 를 이용하지 마십시오. 대신에 rest syntax `...` 를 이용해 주십시오.

> Why? `...` is explicit about which arguments you want pulled. Plus rest arguments are a real Array and not Array-like like `arguments`.

> 왜? `...` 을 이용하는것으로 몇개의 파라메터를 이용하고 싶은가를 확실하게 할 수 있습니다. 더해서 rest 파라메터는 `arguments` 와 같은 Array-like 오브젝트가 아닌 진짜 Array 입니다.

```
// bad
function concatenateAll() {
  const args = Array.prototype.slice.call(arguments);
  return args.join('');
```

```
}

// good
function concatenateAll(...args) {
  return args.join('');
}
```

- 7.7 Use default parameter syntax rather than mutating function arguments.
- 7.7 함수의 파라메터를 변이시키는 것보다 default 파라메터를 이용해 주십시오.

```
// really bad
function handleThings(opts) {
  // No! We shouldn't mutate function arguments.
  // Double bad: if opts is falsy it'll be set to an object which may
  // be what you want but it can introduce subtle bugs.

  // 안돼 ! 함수의 파라메터를 변이시키면 안됩니다.
  // 만약 opts가 falsy 하다면 바라는데로 오브젝트가 설정됩니다.
  // 하지만 미묘한 버그를 일으킬지도 모릅니다.
  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 effect가 있을 default 파라메터의 이용은 피해 주십시오.

> Why? They are confusing to reason about.

> 왜? 혼란을 야기하기 때문입니다.

```
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 항상 default 파라메터는 뒤쪽에 두십시오.

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

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

- 7.10 Never use the Function constructor to create a new function.
- 7.10 절대 새 함수를 작성하기 위해 Function constructor를 이용하지 마십시오.

> Why? Creating a function in this way evaluates a string similarly to eval(), which opens vulnerabilities.

> 왜? 이 방법으로 문자열을 평가시켜 새 함수를 작성하는것은 eval() 과 같은 수준의 취약점을 일으킬 수 있습니다.

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

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

↑ back to top

Arrow함수(Arrow Functions)

- 8.1 When you must use function expressions (as when passing an anonymous function), use arrow function notation.
- 8.1 (무명함수를 전달하는 듯한)함수식을 이용하는 경우 arrow함수 표기를 이용해 주십시오.

> Why? It creates a version of the function that executes in the context of `this`, which is usually what you want, and is a more concise syntax.

> 왜? arrow함수는 그 context의 `this` 에서 실행하는 버전의 함수를 작성합니다. 이것은 통상 기대대로의 동작을 하고, 보다 간결한 구문이기 때문입니다.

> Why not? If you have a fairly complicated function, you might move that logic out into its own function declaration.

> 이용해야만 하지 않나 ? 복잡한 함수에서 로직을 정의한 함수의 바깥으로 이동하고 싶을때.

```
// 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 함수의 본체가 하나의 식으로 구성된 경우에는 중괄호({})를 생략하고 암시적 return을 이용하는것이 가능합니다. 그 외에는 `return` 문을 이용해 주십시오.

> Why? Syntactic sugar. It reads well when multiple functions are chained together.

> 왜? 문법 설탕이니까요. 복수의 함수가 연결된 경우에 읽기 쉬워집니다.

> Why not? If you plan on returning an object.

> 사용해야만 하지 않아? 오브젝트를 반환할때.

```
// 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 식이 복수행에 걸쳐있을 경우는 가독성을 더욱 좋게하기 위해 소괄호()로 감싸 주십시오.

> Why? It shows clearly where the function starts and ends.

> 왜? 함수의 개시와 종료부분이 알기쉽게 보이기 때문입니다.

```
// 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 함수의 인수가 하나인 경우 소괄호()를 생략하는게 가능합니다.

> Why? Less visual clutter.

> 왜? 별로 보기 어렵지 않기 때문입니다.

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

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

↑ back to top

Classes & Constructors

- 9.1 Always use `class`. Avoid manipulating `prototype` directly.
- 9.1 `prototype` 을 직접 조작하는것을 피하고 항상 `class` 를 이용해 주십시오.

> Why? `class` syntax is more concise and easier to reason about.

> 왜? `class` 구문은 간결하고 의미를 알기 쉽기 때문입니다.

```
// 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 상속은 `extends` 를 이용해 주십시오.

> Why? It is a built-in way to inherit prototype functionality without breaking `instanceof`.

> 왜? `instanceof` 를 파괴하지 않고 프로토타입 상속을 하기 위해 빌트인 된 방법이기 때문입니다.

```
// 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 메소드의 반환값으로 `this` 를 반환하는 것으로 메소드채이닝을 할 수 있습니다.

```
// 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);
```

- 9.4 It's okay to write a custom toString() method, just make sure it works successfully and causes no side effects.
- 9.4 독자의 toString()을 작성하는것을 허용하지만 올바르게 동작하는지와 side effect 가 없는지만 확인해 주십시오.

```
class Jedi {
  constructor(options = {}) {
    this.name = options.name || 'no name';
  }

  getName() {
    return this.name;
  }

  toString() {
    return `Jedi - ${this.getName()}`;
  }
}
```

↑ back to top

모듈(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 비표준 모듈시스템이 아닌 항상 (`import`/`export`) 를 이용해 주십시오. 이렇게 함으로써 선호하는 모듈시스템에 언제라도 옮겨가는게 가능해 집니다.

  > Why? Modules are the future, let's start using the future now.

  > 왜? 모듈은 미래가 있습니다. 지금 그 미래를 사용하여 시작합시다.

  ```
  // 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 는 이용하지 마십시오.

  > Why? This makes sure you have a single default export.

  > 왜? single default export 임을 주의할 필요가 있습니다.

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

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

- 10.3 And do not export directly from an import.
- 10.3 import 문으로부터 직접 export 하는것은 하지말아 주십시오.

  > Why? Although the one-liner is concise, having one clear way to import and one clear way to export makes things consistent.

  > 왜? 한줄짜리는 간결하지만 import 와 export 방법을 명확히 한가지로 해서 일관성을 갖는 것이 가능합니다.

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

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

↑ back to top

이터레이터와 제너레이터(Iterators and Generators)

- 11.1 Don't use iterators. Prefer JavaScript's higher-order functions like `map()` and `reduce()` instead of loops like `for-of`.
- 11.1 iterators를 이용하지 마십시오. `for-of` 루프 대신에 `map()` 과 `reduce()` 와 같은 JavaScript 고급함수(higher-order functions)를 이용해 주십시오.

> Why? This enforces our immutable rule. Dealing with pure functions that return values is easier to reason about than side-effects.

> 왜? 고급함수는 immutable(불변)룰을 적용합니다. side effect에 대해 추측하는거보다 값을 반환하는 순수 함수를 다루는게 간단하기 때문입니다.

```
const numbers = [1, 2, 3, 4, 5];

// bad
let sum = 0;
for (let num of numbers) {
  sum += num;
}

sum === 15;

// good
let sum = 0;
numbers.forEach((num) => sum += num);
sum === 15;

// best (use the functional force)
const sum = numbers.reduce((total, num) => total + num, 0);
sum === 15;
```

- 11.2 Don't use generators for now.
- 11.2 현시점에서는 generators는 이용하지 마십시오.

> Why? They don't transpile well to ES5.

> 왜? ES5로 잘 transpile 하지 않기 때문입니다.

↑ back to top

프로퍼티(Properties)

- 12.1 Use dot notation when accessing properties.
- 12.1 프로퍼티에 억세스하는 경우는 점 `.` 을 사용해 주십시오.

```
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 변수를 사용해 프로퍼티에 억세스하는 경우는 대괄호 `[]` 를 사용해 주십시오.

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

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

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

↑ back to top

변수(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 변수를 선언 할 때는 항상 `const` 를 사용해 주십시오. 그렇게 하지 않으면 글로벌 변수로 선언됩니다. 글로벌 namespace 를 오염시키지 않도록 캡틴플래닛도 경고하고 있습니다.

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

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

- 13.2 Use one `const` declaration per variable.
- 13.2 하나의 변수선언에 대해 하나의 `const` 를 이용해 주십시오.

> Why? It's easier to add new variable declarations this way, and you never have to worry about swapping out a `;` for a `,` or introducing punctuation-only diffs.

> 왜? 이 방법의 경우, 간단히 새 변수를 추가하는게 가능합니다. 또한 `,` 를 `;` 로 바꿔버리는 것에 대해 걱정할 필요가 없습니다.

```
// 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 우선 `const` 를 그룹화하고 다음에 `let` 을 그룹화 해주십시오.

> Why? This is helpful when later on you might need to assign a variable depending on one of the previous assigned variables.

> 왜? 이전에 할당한 변수에 대해 나중에 새 변수를 추가하는 경우에 유용하기 때문입니다.

```
// 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;
```

- 13.4 Assign variables where you need them, but place them in a reasonable place.
- 13.4 변수를 할당할때는 필요하고 합리적인 장소에 두시기 바랍니다.

> Why? `let` and `const` are block scoped and not function scoped.

> 왜? `let` 과 `const` 는 블록스코프이기 때문입니다. 함수스코프가 아닙니다.

```
// good
function() {
  test();
  console.log('doing stuff..');

  //..other stuff..

  const name = getName();

  if (name === 'test') {
    return false;
  }

  return name;
}

// bad - unnecessary function call
// 필요없는 함수 호출
function(hasName) {
  const name = getName();

  if (!hasName) {
    return false;
  }

  this.setFirstName(name);

  return true;
}

// good
function(hasName) {
  if (!hasName) {
    return false;
  }
```

```
  const name = getName();
  this.setFirstName(name);

  return true;
}
```

↑ back to top

Hoisting

- 14.1 `var` declarations get hoisted to the top of their scope, their assignment does not. `const` and `let` declarations are blessed with a new concept called Temporal Dead Zones (TDZ). It's important to know why typeof is no longer safe.
- 14.1 `var` 선언은 할당이 없이 스코프의 선두에 hoist 됩니다. `const` 와 `let` 선언은Temporal Dead Zones (TDZ) 라고 불리는 새로운 컨셉의 혜택을 받고 있습니다. 이것은 왜 typeof 는 더이상 안전하지 않은가를 알고있는 것이 중요합니다.

```
// we know this wouldn't work (assuming there
// is no notDefined global variable)
// (notDefined 가 글로벌변수에 존재하지 않는다고 판정한 경우.)
// 잘 동작하지 않습니다.
function example() {
  console.log(notDefined); // => throws a ReferenceError
}

// creating a variable declaration after you
// reference the variable will work due to
// variable hoisting. Note: the assignment
// value of `true` is not hoisted.
// 그 변수를 참조하는 코드의 뒤에서 그 변수를 선언한 경우
// 변수가 hoist 된 상태에서 동작합니다..
// 주의 : `true` 라는 값 자체는 hoist 되지 않습니다.
function example() {
  console.log(declaredButNotAssigned); // => undefined
  var declaredButNotAssigned = true;
}

// The interpreter is hoisting the variable
// declaration to the top of the scope,
// which means our example could be rewritten as:
// 인터프리터는 변수선언을 스코프의 선두에 hoist 합니다.
// 위의 예는 다음과 같이 다시 쓸수 있습니다.
function example() {
  let declaredButNotAssigned;
  console.log(declaredButNotAssigned); // => undefined
  declaredButNotAssigned = true;
}

// using const and let
// const 와 let 을 이용한 경우
```

```
function example() {
  console.log(declaredButNotAssigned); // => throws a ReferenceError
  console.log(typeof declaredButNotAssigned); // => throws a ReferenceError
  const declaredButNotAssigned = true;
}
```

- 14.2 Anonymous function expressions hoist their variable name, but not the function assignment.
- 14.2 무명함수의 경우 함수가 할당되기 전의 변수가 hoist 됩니다.

```
function example() {
  console.log(anonymous); // => undefined

  anonymous(); // => TypeError anonymous is not a function

  var anonymous = function() {
    console.log('anonymous function expression');
  };
}
```

- 14.3 Named function expressions hoist the variable name, not the function name or the function body.
- 14.3 명명함수의 경우도 똑같이 변수가 hoist 됩니다. 함수명이나 함수본체는 hoist 되지 않습니다.

```
function example() {
  console.log(named); // => undefined

  named(); // => TypeError named is not a function

  superPower(); // => ReferenceError superPower is not defined

  var named = function superPower() {
    console.log('Flying');
  };
}

// the same is true when the function name
// is the same as the variable name.
// 함수명과 변수명이 같은 경우도 같은 현상이 발생합니다.
function example() {
  console.log(named); // => undefined

  named(); // => TypeError named is not a function

  var named = function named() {
    console.log('named');
  }
}
```

- 14.4 Function declarations hoist their name and the function body.
- 14.4 함수선언은 함수명과 함수본체가 hoist 됩니다.

```
function example() {
  superPower(); // => Flying

  function superPower() {
    console.log('Flying');
  }
}
```

- For more information refer to JavaScript Scoping & Hoisting by Ben Cherry.
- 더 자세한건 이쪽을 참고해 주십시오. JavaScript Scoping & Hoisting by Ben Cherry.

↑ back to top

조건식과 등가식(Comparison Operators & Equality)

- 15.1 Use `===` and `!==` over `==` and `!=`.
- 15.1 `==` 이나 `!=` 보다 `===` 와 `!==` 을 사용해 주십시오.
- 15.2 Conditional statements such as the `if` statement evaluate their expression using coercion with the `ToBoolean` abstract method and always follow these simple rules:
  - **Objects** evaluate to **true**
  - **Undefined** evaluates to **false**
  - **Null** evaluates to **false**
  - **Booleans** evaluate to **the value of the boolean**
  - **Numbers** evaluate to **false** if **+0, -0, or NaN**, otherwise **true**
  - **Strings** evaluate to **false** if an empty string `''`, otherwise **true**
- 15.2 `if` 문과 같은 조건식은 `ToBoolean` 메소드에 의한 강제형변환으로 평가되어 항상 다음과 같은 심플한 룰을 따릅니다.
  - **오브젝트** 는 **true** 로 평가됩니다.
  - **undefined** 는 **false** 로 평가됩니다.
  - **null** 은 **false** 로 평가됩니다.
  - **부울값** 은 **boolean형의 값** 으로 평가됩니다.
  - **수치** 는 **true** 로 평가됩니다. 하지만 **+0, -0, or NaN** 의 경우는 **false** 입니다.
  - **문자열** 은 **true** 로 평가됩니다. 하지만 빈문자 `''` 의 경우는 **false** 입니다.

```
if ([0]) {
  // true
  // An array is an object, objects evaluate to true
  // 배열은 오브젝트이므로 true 로 평가됩니다.
}
```

- 15.3 Use shortcuts.
- 15.3 단축형을 사용해 주십시오.

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

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

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

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

- 15.4 For more information see Truth Equality and JavaScript by Angus Croll.
- 15.4 더 자세한건 이쪽을 참고해 주십시오. Truth Equality and JavaScript by Angus Croll.

↑ back to top

블록(Blocks)

- 16.1 Use braces with all multi-line blocks.
- 16.1 복수행의 블록에는 중괄호 ({}) 를 사용해 주십시오.

```
// bad
if (test)
  return false;

// good
if (test) return false;

// good
if (test) {
  return false;
}

// bad
```

```
function() { return false; }

// good
function() {
  return false;
}
```

- 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 복수행 블록의 `if` 와 `else` 를 이용하는 경우 `else` 는 `if` 블록 끝의 중괄호(})와 같은 행에 위치시켜 주십시오.

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

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

↑ back to top

코멘트(Comments)

- 17.1 Use `/** ... */` for multi-line comments. Include a description, specify types and values for all parameters and return values.
- 17.1 복수행의 코멘트는 `/** ... */` 을 사용해 주십시오. 그 안에는 설명과 모든 파라메터, 반환값에 대해 형이나 값을 기술해 주십시오.

```
// 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 단일행 코멘트에는 `//` 을 사용해 주십시오. 코멘트를 추가하고 싶은 코드의 상부에 배치해 주십시오. 또한, 코멘트의 앞에 빈행을 넣어 주십시오.

```
// 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;
}
```

- 17.3 Prefixing your comments with `FIXME` or `TODO` helps other developers quickly understand if you're pointing out a problem that needs to be revisited, or if you're suggesting a solution to the problem that needs to be implemented. These are different than regular comments because they are actionable. The actions are `FIXME -- need to figure this out` or `TODO -- need to implement`.
- 17.3 문제를 지적하고 재고를 촉구하는 경우나 문제의 해결책을 제안하는 경우 등, 코멘트의 앞에 `FIXME` 나 `TODO` 를 붙이는 것으로 다른 개발자의 빠른 이해를 도울수 있습니다. 이런것들은 어떤 액션을 따른다는 의미로 통상의 코멘트와는 다릅니다. 액션이라는 것은 `FIXME -- 해결이 필요` 또는 `TODO -- 구현이 필요` 를 뜻합니다.
- 17.4 Use `// FIXME:` to annotate problems.
- 17.4 문제에 대한 주석으로써 `// FIXME:` 를 사용해 주십시오.

```
class Calculator extends Abacus {
  constructor() {
    super();

    // FIXME: shouldn't use a global here
    // FIXME: 글로벌변수를 사용해서는 안됨.
    total = 0;
  }
}
```

- 17.5 Use `// TODO:` to annotate solutions to problems.
- 17.5 문제의 해결책에 대한 주석으로 `// TODO:` 를 사용해 주십시오.

```
class Calculator extends Abacus {
  constructor() {
    super();

    // TODO: total should be configurable by an options param
    // TODO: total 은 옵션 파라메터로 설정해야함.
    this.total = 0;
  }
}
```

↑ back to top

공백(Whitespace)

- 18.1 Use soft tabs set to 2 spaces.
- 18.1 탭에는 스페이스 2개를 설정해 주십시오.

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

// bad
function() {
·const name;
}

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

- 18.2 Place 1 space before the leading brace.
- 18.2 주요 중괄호 ({}) 앞에는 스페이스를 1개 넣어 주십시오.

```
// 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',
});
```

- 18.3 Place 1 space before the opening parenthesis in control statements (`if`, `while` etc.). Place no space before the argument list in function calls and declarations.

- 18.3 제어구문 (`if` 문이나 `while` 문 등) 의 소괄호 (()) 앞에는 스페이스를 1개 넣어 주십시오. 함수선언이나 함수호출시 인수리스트의 앞에는 스페이스를 넣지 마십시오.

```
// 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 연산자 사이에는 스페이스를 넣어 주십시오.

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

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

- 18.5 End files with a single newline character.
- 18.5 파일 끝에는 개행문자를 1개 넣어 주십시오.

```
// 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 길게 메소드를 채이닝하는 경우는 인덴트를 이용해 주십시오. 행이 새로운 문이 아닌 메소드 호출인 것을 강조하기 위해서 선두에 점 (.) 을 배치해 주십시오.

```
// 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 문의 앞과 블록의 뒤에는 빈행을 남겨 주십시오.

```
// 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 블록에 빈행을 끼워 넣지 마십시오.

```
// 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 소괄호()의 안쪽에 스페이스를 추가하지 마십시오.

```
// 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 대괄호([])의 안쪽에 스페이스를 추가하지 마십시오.

```
// 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 중괄호({})의 안쪽에 스페이스를 추가해 주십시오.

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

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

↑ back to top

콤마(Commas)

- 19.1 Leading commas: **Nope.**
- 19.1 선두의 콤마 **하지마요**

```
// 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 끝의 콤마 **좋아요**

> Why? This leads to cleaner git diffs. Also, transpilers like Babel will remove the additional trailing comma in the transpiled code which means you don't have to worry about the trailing comma problem in legacy browsers.

> 왜? 이것은 깨끗한 git의 diffs 로 이어집니다. 또한 Babel과 같은 트랜스파일러는 transpile 하는 사이에 쓸데없는 끝의 콤마를 제거합니다. 이것은 레거시브라우저에서의 불필요한 콤마 문제를 고민할 필요가 없다는것을 의미합니다.

```
// 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',
];
```

↑ back to top

세미콜론(Semicolons)

- 20.1 **Yup.**
- 20.1 **쓉시다**

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

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

// good (guards against the function becoming an argument when two files
with IIFEs are concatenated)
// good (즉시함수가 연결된 2개의 파일일때 인수가 되는 부분을 보호합니다.
;(() => {
  const name = 'Skywalker';
```

```
  return name;
})();
```

Read more.

↑ back to top

형변환과 강제(Type Casting & Coercion)

- 21.1 Perform type coercion at the beginning of the statement.
- 21.1 문의 선두에서 형의 강제를 행합니다.
- 21.2 Strings:
- 21.2 문자열의 경우:

```
//  => 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 수치의 경우: `Number` 로 형변환하는 경우는 `parseInt` 를 이용하고, 항상 형변환을 위한 기수를 인수로 넘겨 주십시오.

```
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);
```

- 21.4 If for whatever reason you are doing something wild and `parseInt` is your bottleneck and need to use Bitshift for performance reasons, leave a comment explaining why and what you're doing.
- 21.4 무언가의 이유로 인해 `parseInt` 가 bottleneck 이 되어, 성능적인 이유로 Bitshift를 사용할 필요가 있는 경우 하려고 했던 것에 대해, 왜(why) 와 무엇(what)의 설명을 코멘트로 해서 남겨 주십시오.

```
// good
/**
 * parseInt was the reason my code was slow.
 * Bitshifting the String to coerce it to a
 * Number made it a lot faster.
 * parseInt 가 원인으로 느렸음.
 * Bitshift를 통한 수치로의 문자열 강제 형변환으로
 * 성능을 개선시킴.
 */
const val = inputValue >> 0;
```

- 21.5 **Note:** Be careful when using bitshift operations. Numbers are represented as 64-bit values, but Bitshift operations always return a 32-bit integer (source). Bitshift can lead to unexpected behavior for integer values larger than 32 bits. Discussion. Largest signed 32-bit Int is 2,147,483,647:
- 21.5 **주의:** bitshift를 사용하는 경우의 주의사항. 수치는 64비트 값으로 표현되어 있으나 bitshift 연산한 경우는 항상 32비트 integer 로 넘겨집니다.(소스). 32비트 이상의 int 를 bitshift 하는 경우 예상치 못한 현상을 야기할 수 있습니다.(토론) 부호가 포함된 32비트 정수의 최대치는 2,147,483,647 입니다.

```
2147483647 >> 0 //=> 2147483647
2147483648 >> 0 //=> -2147483648
2147483649 >> 0 //=> -2147483647
```

- 21.6 Booleans:
- 21.6 부울값의 경우:

```
const age = 0;

// bad
const hasAge = new Boolean(age);

// good
const hasAge = Boolean(age);

// good
const hasAge = !!age;
```

↑ back to top

명명규칙(Naming Conventions)

- 22.1 Avoid single letter names. Be descriptive with your naming.
- 22.1 1문자의 이름은 피해 주십시오. 이름으로부터 의도가 읽혀질수 있게 해주십시오.

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

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

- 22.2 Use camelCase when naming objects, functions, and instances.
- 22.2 오브젝트, 함수 그리고 인스턴스에는 camelCase를 사용해 주십시오.

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

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

- 22.3 Use PascalCase when naming constructors or classes.
- 22.3 클래스나 constructor에는 PascalCase 를 사용해 주십시오.

```
// 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 프로퍼티명은 선두에 언더스코어 _ 를사용해 주십시오.

```
// 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` 의 참조를 보존하는것은 피해주십시오. arrow함수나 Function#bind 를 이용해 주십시오.

```
// 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 파일을 1개의 클래스로 export 하는 경우, 파일명은 클래스명과 완전히 일치시키지 않으면 안됩니다.

```
// 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 export가 함수일 경우, camelCase를 이용해 주십시오. 파일명은 함수명과 동일해야 합니다.

```
function makeStyleGuide() {
}

export default makeStyleGuide;
```

- 22.8 Use PascalCase when you export a singleton / function library / bare object.
- 22.8 singleton / function library / 빈오브젝트를 export 하는 경우, PascalCase를 이용해 주십시오.

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

export default AirbnbStyleGuide;
```

↑ back to top

억세서(Accessors)

- 23.1 Accessor functions for properties are not required.
- 23.1 프로퍼티를 위한 억세서 (Accessor) 함수는 필수는 아닙니다.
- 23.2 If you do make accessor functions use getVal() and setVal('hello').
- 23.2 억세서 함수가 필요한 경우, `getVal()` 이나 `setVal('hello')` 로 해주십시오.

```
// bad
dragon.age();

// good
dragon.getAge();

// bad
dragon.age(25);

// good
dragon.setAge(25);
```

- 23.3 If the property is a `boolean`, use `isVal()` or `hasVal()`.
- 23.3 프로퍼티가 `boolean` 인 경우, `isVal()` 이나 `hasVal()` 로 해주십시오.

```
// bad
if (!dragon.age()) {
  return false;
}

// good
if (!dragon.hasAge()) {
  return false;
}
```

- 23.4 It's okay to create get() and set() functions, but be consistent.
- 23.4 일관된 경우, `get()` 과 `set()` 으로 함수를 작성해도 좋습니다.

```
class Jedi {
  constructor(options = {}) {
    const lightsaber = options.lightsaber || 'blue';
    this.set('lightsaber', lightsaber);
  }

  set(key, val) {
    this[key] = val;
  }

  get(key) {
    return this[key];
  }
}
```

↑ back to top

이벤트(Events)

- 24.1 When attaching data payloads to events (whether DOM events or something more proprietary like Backbone events), pass a hash instead of a raw value. This allows a subsequent contributor to add more data to the event payload without finding and updating every handler for the event. For example, instead of:

- 24.1 (DOM이벤트나 Backbone events 와 같은 독자의) 이벤트로 payload의 값을 넘길 경우는 raw값 보다는 해시값을 넘겨 주십시오. 이렇게 함으로써, 이후 기여자가 이벤트에 관련한 모든 핸들러를 찾아서 갱신하는 대신 이벤트 payload에 값을 추가하는 것이 가능합니다. 예를들면 아래와 같이

```
// bad
$(this).trigger('listingUpdated', listing.id);

...

$(this).on('listingUpdated', function(e, listingId) {
  // do something with listingId
});
```

prefer: 이쪽이 좋습니다:

```
// good
$(this).trigger('listingUpdated', { listingId: listing.id });

...

$(this).on('listingUpdated', function(e, data) {
  // do something with data.listingId
});
```

↑ back to top

jQuery

- 25.1 Prefix jQuery object variables with a `$`.

- 25.1 jQuery오브젝트의 변수는 선두에 `$` 를 부여해 주십시오.

```
// bad
const sidebar = $('.sidebar');

// good
const $sidebar = $('.sidebar');

// good
const $sidebarBtn = $('.sidebar-btn');
```

- 25.2 Cache jQuery lookups.
- 25.2 jQuery의 검색결과를 캐시해 주십시오.

```
// bad
function setSidebar() {
  $('.sidebar').hide();

  // ...stuff...

  $('.sidebar').css({
    'background-color': 'pink'
  });
}

// good
function setSidebar() {
  const $sidebar = $('.sidebar');
  $sidebar.hide();

  // ...stuff...

  $sidebar.css({
    'background-color': 'pink'
  });
}
```

- 25.3 For DOM queries use Cascading `$('.sidebar ul')` or parent > child `$('.sidebar > ul')`. jsPerf
- 25.3 DOM 검색에는 `$('.sidebar ul')` 이나 `$('.sidebar > ul')` 와 같은 Cascading 을 사용해 주십시오. jsPerf
- 25.4 Use `find` with scoped jQuery object queries.
- 25.4 한정된 jQuery 오브젝트 쿼리에는 `find` 를 사용해 주십시오.

```
// bad
$('ul', '.sidebar').hide();

// bad
$('.sidebar').find('ul').hide();

// good
$('.sidebar ul').hide();

// good
$('.sidebar > ul').hide();
```

```
// good
$sidebar.find('ul').hide();
```

↑ back to top

ECMAScript 5 Compatibility

- 26.1 Refer to Kangax's ES5 compatibility table.

↑ back to top

ECMAScript 6 Styles

- 27.1 This is a collection of links to the various es6 features.

1. Arrow Functions
2. Classes
3. Object Shorthand
4. Object Concise
5. Object Computed Properties
6. Template Strings
7. Destructuring
8. Default Parameters
9. Rest
10. Array Spreads
11. Let and Const
12. Iterators and Generators
13. Modules

↑ back to top

Testing

- 28.1 **Yup.**

```
function () {
  return true;
}
```

- 28.2 **No, but seriously**:
- Whichever testing framework you use, you should be writing tests!
- Strive to write many small pure functions, and minimize where mutations occur.
- Be cautious about stubs and mocks - they can make your tests more brittle.
- We primarily use `mocha` at Airbnb. `tape` is also used occasionally for small, separate modules.

- 100% test coverage is a good goal to strive for, even if it's not always practical to reach it.
- Whenever you fix a bug, *write a regression test*. A bug fixed without a regression test is almost certainly going to break again in the future.

↑ back to top

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...

↑ back to top

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

↑ back to top

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

↑ back to top

Translation

This style guide is also available in other languages:

-  **Brazilian Portuguese**: armoucar/javascript-style-guide
-  **Bulgarian**: borislavvv/javascript
-  **Catalan**: fpmweb/javascript-style-guide
-  **Chinese (Simplified)**: sivan/javascript-style-guide
-  **Chinese (Traditional)**: jigsawye/javascript
-  **French**: nmussy/javascript-style-guide
-  **German**: timofurrer/javascript-style-guide
-  **Italian**: sinkswim/javascript-style-guide
-  **Japanese**: mitsuruog/javascript-style-guide
-  **Korean**: tipjs/javascript-style-guide
-  **Polish**: mjurczyk/javascript
-  **Russian**: uprock/javascript
-  **Spanish**: paolocarrasco/javascript-style-guide
-  **Thai**: lvarayut/javascript-style-guide

The JavaScript Style Guide Guide

- Reference

Chat With Us About JavaScript

- Find us on gitter.

Contributors

- View Contributors

License

(The MIT License)

↑ back to top

Amendments

We encourage you to fork this guide and change the rules to fit your team's style guide. Below, you may list some amendments to the style guide. This allows you to periodically update your style guide without having to deal with merge conflicts.

};