**목차**

* 소스파일 기본
* 형식
* 변수
* 배열
* 객체
* 클래스
* 함수
* 문자열
* 숫자
* 제어문

**소스파일 기본 📰**

* 소스파일의 이름은 **알파벳 소문자, 하이픈( - ), 밑줄( \_ )** 으로만 작성한다.
* 소스파일의 확장자명은 **.js** 으로 작성한다.
* 소스파일의 인코딩은 **UTF-8** 으로 작성한다.
* 탭을 이용한 들여쓰기는 하지 않는다.

**형식 ✔️**

* **중괄호**
  + 중괄호는 하나의 구문만을 포함하고 있더라도 모든 제어문*(i.e. if, else, for, do, while..)*에 사용되어야 한다.
  + if문의 경우, 하나의 구문만 포함할 경우, 같은 라인에 작성하고 중괄호를 생략할 수 있다.
* // bad
* if (someVeryLongCondition())
* doSomething();
* // No
* for (let i = 0; i < foo.length; i++) bar(foo[i]);
* // Yes

if (shortCondition()) foo();

* + 중괄호의 사용은 **Kernighan and Ritchie Style**에 따른다.
    - 여는 중괄호 전에는 줄을 바꾸지 않는다.
    - 여는 중괄호 이후에 줄을 바꾼다.
    - 닫는 중괄호 전에 줄을 바꾼다.
    - 닫는 중괄호 이후에 줄을 바꾼다. 단, 닫는 중괄호 다음에 else, catch, while, 점, 세미콜론, 닫는 소괄호가 따라올 경우에는 줄을 바꾸지 않는다.
  + class InnerClass {
  + constructor() {}
  + /\*\* @param {number} foo \*/
  + method(foo) {
  + if (condition(foo)) {
  + try {
  + something();
  + } catch (err) {
  + recover();
  + }
  + }
  + }

}

* + 빈 블록의 경우, **여는 중괄호( { )** 바로 다음에 닫는 중괄호를 사용한다.  
    단, **if-else 구조** 혹은 **try-catch-finally 구조**에서의 빈 블록에서는 줄을 바꾼다.
  + // Yes
  + function doNothing() {}
  + // No
  + if (condition) {
  + // …
  + } else if (otherCondition) {} else {
  + // …
  + }
  + // No
  + try {
  + // …

} catch (e) {}

* **들여쓰기**
  + 새로운 블록이 나타날 때 마다 **두 칸의 공백**(스페이스바)를 통해 들여쓴다.
  + 배열, 객체는 블록과 동일하게 들여쓴다.
  + 클래스
    - 클래스 리터럴은 블록과 동일하게 들여쓴다.
    - 클래스의 각 메소드와 닫는 중괄호 다음에는 세미콜론을 사용하지 않는다.
    - 클래스의 상속 키워드로 @extends가 아닌 extends를 사용한다.
  + 함수 표현식을 작성할 대에는 함수를 이전 레벨보다 2칸 들여써 작성한다.
  + prefix.something.LongFunctionName('whatever', (a1, a2) => {
  + // Indent the function body +2 relative to indentation depth
  + // of the 'prefix' statement one line above.
  + if (a1.equals(a2)) {
  + someOtherLongFunctionName(a1);
  + } else {
  + andNowForSomethingCompletelyDifferent(a2.parrot);
  + }

});

* + **스위치문**은 블록과 동일하게 들여쓰며, break와 다음 case 사이의 공백은 선택
  + switch (animal) {
  + case Animal.BANDERSNATCH:
  + handleBandersnatch();
  + break;
  + case Animal.JABBERWOCK:
  + handleJabberwock();
  + break;
  + default:
  + throw new Error('Unknown animal');

}

* + **점( . )**을 이용하여 긴 메소드 체인을 작성할 때는 각 메소드마다 들여쓴다.
  + // bad
  + $('#items').find('.selected').highlight().end().find('.open').updateCount();
  + // bad
  + $('#items').
  + find('.selected').
  + highlight().
  + end().
  + find('.open').
  + updateCount();
  + // good
  + $('#items')
  + .find('.selected')
  + .highlight()
  + .end()
  + .find('.open')
  + .updateCount();
  + // bad
  + const leds = stage.selectAll('.led').data(data).enter().append('svg:svg').classed('led', true)
  + .attr('width', (radius + margin) \* 2).append('svg:g')
  + .attr('transform', `translate(${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);
  + // good

const leds = stage.selectAll('.led').data(data);

* **구문**
  + 한 줄에는 하나의 구문만을 작성한다.
  + 모든 구문의 끝에는 **세미콜론**을 붙인다.
  + // bad
  + let name = 'Cada'
  + doSomething() saySomething()
  + // good
  + let name = 'Cada';
  + doSomething();

saySomething();

* **공백**
  + 한 줄이 **80자**를 넘기지 않도록 한다.
  + 줄 공백 (Vertical Whitespace)
    - 줄 공백은 아래의 사항에서 나타날 수 있다.
      * 각 구문 사이
      * 클래스나 객체 리터럴 내의 연속적으로 작성된 메소드 사이
      * 오브젝트 리터럴 내의 연속적으로 작성된 속성 사이에서는 선택적으로 빈 줄을 사용할 수 있다. 이는, 논리적 그룹을 만드는데 사용된다.
      * 클래스나 객체 리터럴 내의 첫 메소드 전이나 마지막 메소드 다음에 빈 줄을 선택적으로 사용할 수 있다.
    - 줄 공백으로 2줄 이상 연속으로 사용하지 않는다.
  + // 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;

* + 칸 공백 (Horizontal Whitespace)  
    칸 공백은 **리딩**(줄의 시작), **트레일링**(줄의 끝), **인터널**로 나눌 수 있다.
    - 리딩 공백은 들여쓰기 규칙에 따라 언제든지 사용될 수 있다.
    - 트레일링 공백은 사용하지 않는다.
    - 칸 공백은 아래의 사항에서 나타날 수 있다.
      * **if**, **for**, **catch**와 같은 키워드와 소괄호 사이  
        (단, function과 super는 예외)
      * // bad
      * if(myCondition) {
      * // do something
      * }
      * // good
      * if (myCondition) {
      * // do something

}

* + - * **else**, **catch**와 같은 키워드와 닫는 중괄호 사이
      * // bad
      * if(myCondition) {
      * // do something
      * }else { }
      * // good
      * if (myCondition) {
      * // do something

} else { }

* + - * **여는 중괄호 ( { )** 전  
        (단, 오브젝트 리터럴의 첫 인자나 배열의 첫 인자, 템플릿 리터럴은 예외)
      * **이항 연산자**와 **삼항 연산자**의 양 쪽
      * // bad
      * const name = someCondotion?20:30;
      * // good

const name = someCondotion ? 20 : 30;

* + - * **반점( , )**, **세미콜론( ; )** 다음 (단, 반점과 세미콜론 전에는 사용하지 않음)
      * // bad
      * const arr = [1,2,3,4];
      * // good

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

* + - * 오브젝트 리터럴 내에서 **콜론( : )** 다음
      * // bad
      * const myobject = {
      * "name":"Chris",
      * "age":34,
      * };
      * // good
      * const myobject = {
      * "name": "Chris",
      * "age": 34,

};

* + - * **//** 양 쪽, **/\*** 다음, **\*/** 전
    - 소괄호 사이, 대괄호 사이에는 칸 공백을 사용하지 않는다.
    - // bad
    - function bar( foo ) {
    - return foo;
    - }
    - // good
    - function bar(foo) {
    - return foo;
    - }
    - // bad
    - if ( foo ) {
    - console.log(foo);
    - }
    - // good
    - if (foo) {
    - console.log(foo);
    - }
    - // bad
    - const foo = [ 1, 2, 3 ];
    - console.log(foo[ 0 ]);
    - // good
    - const foo = [1, 2, 3];

console.log(foo[0]);

* + - 중괄호 사이에는 칸 공백을 사용한다.
    - // bad
    - const foo = {clark: 'kent'};
    - // good

const foo = { clark: 'kent' };

**변수 💾**

* 한 줄에 하나의 변수를 선언한다.
* // bad
* let a = 1, b = 3;
* // good
* let a = 1;

let b = 2;

* 지역 변수는 그 변수를 포함하는 블록 시작에서 선언하지 않고, 사용 범위를 최소화하기 위해 사용되는 지점과 가장 가까운 곳에서 선언한다.
* // good
* function() {
* test();
* console.log('doing stuff..');
* const name = getName();
* if (name === 'test') {
* return false;
* }
* return name;
* }
* // bad - 함수 호출의 부적절한 위치
* 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;

}

* JSDoc을 위한 주석은 변수 선언 이전 혹은 변수 이름 이전에 작성한다.  
  (단, 두 가지 위치에 동시에 모두 작성하지 않는다)
* // bad
* /\*\* Some description. \*/
* const /\*\* !Array<number> \*/ data = [];
* const /\*\* !Array<number> \*/ data = [];
* // good
* /\*\*
* \* Some description.
* \* @type {!Array<number>}
* \*/

const data = [];

* 변수를 선언할 때는 **const**를 사용한다.  
  단, 변수의 값이 바뀌는 경우 **let**을 사용한다.
* const 선언문을 먼저 그룹화한 다음에 let 선언문을 그룹화한다.
* // 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;

**배열 🏢**

* 배열을 선언할 때는 Array 생성자가 아닌 **리터럴 구문**을 사용한다.
* // bad
* const items = new Array();
* // good

const items = [];

* 배열에 값을 넣을 때는 **Array.push**를 사용한다.
* const someStack = [];
* // bad
* someStack[someStack.length] = 'abracadabra';
* // good

someStack.push('abracadabra');

* 배열을 복사할 때는 배열의 **확장연산자 ( ... )**를 사용한다.
* // bad
* const len = items.length;
* const itemsCopy = [];
* let i;
* for (i = 0; i < len; i++) {
* itemsCopy[i] = items[i];
* }
* // good

const itemsCopy = [...items];

* 배열과 유사한 오브젝트를 배열로 변환할 때는 **Array.from**을 사용한다.
* const foo = document.querySelectorAll('.foo');

const nodes = Array.from(foo);

* 배열의 선언 시, 마지막 원소를 포함하여 각 원소 끝에는 **점( . )**을 포함한다.
* const values = [
* 'first value',
* 'second value',
* 'third value',

];

* 배열에 숫자가 아닌 속성을 사용하지 않는다.  
  사용을 원할경우 **맵** 혹은 **오브젝트**를 사용한다.
* // bad
* const arr = [];
* arr["str"] = 32;
* // good
* const obj = {};
* arr.str = 32;
* const map = new Map();

map.set("str", 32);

* 한 배열로부터 복수개의 값을 할당받을 때는 **구조 분해 할당을 사용한다.**  
  이는 함수의 파라미터에서도 사용할 수 있다.
* const [a, b, c, ...rest] = generateResults();
* let [, b,, d] = someArray;

function optionalDestructuring([a = 4, b = 2] = []) { … };

**객체 🎍**

* 오브젝트를 선언할 때는 Object 생성자가 아닌 **리터럴 구문**을 사용한다.
* // bad
* const item = new Object();
* // good

const item = {};

* **예약어**를 키로 사용하지 않는다. 대신, 동의어를 사용한다.
* // bad
* const superman = {
* default: { clark: 'kent' },
* private: true,
* };
* // good
* const superman = {
* defaults: { clark: 'kent' },
* hidden: true,

};

* 키는 **큰 따옴표( " )**를 씌운다.  
  단, 큰 따옴표를 씌운 키와 씌우지 않은 키를 한 오브젝트에 동시에 사용하지 않는다.
* // bad
* var my\_object = {
* key: "value",
* };
* var my\_object = {
* key: "value",
* "key2": "value2",
* };
* // good
* var my\_object = {
* "key": "value",
* "key2": "value2",

};

* 동적 속성명을 사용할 때는, **계산된 속성명(Computed Porperty Name)**을 사용한다.
* 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

};

* 메소드 단축 구문을 사용한다.
* // bad
* const atom = {
* value: 1,
* addValue: function (value) {
* return atom.value + value;
* },
* };
* // good
* const atom = {
* value: 1,
* addValue(value) {
* return atom.value + value;
* },

};

* 속성 단축 구문을 사용한다.  
  단, 선언의 시작 부분에 그룹화하여 작성한다.
* const lukeSkywalker = 'Luke Skywalker';
* // bad
* const obj = {
* lukeSkywalker: lukeSkywalker,
* };
* // good
* const obj = {
* lukeSkywalker,
* };
* // 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,

};

**클래스 📇**

* 생성자는 선택적으로 작성한다. 하지만 하위 클래스는 필드를 설정하기 전에 반드시 **super()**를 호출해야한다. 그렇지 않으면 **this**에 접근할 것이다.  
  인터페이스의 경우 메소드가 아닌 속성을 생성자에서 반드시 선언해야 한다.
* 정적 메소드의 사용 보다는 모듈 함수를 더 지향한다.
  + 정적 메소드는 클래스 내에서만 호출되어야 gksek.
  + 동적 인스턴스를 포함하는 변수나 하위 클래스의 생성자 내부에서 호출되어서는 안된다.
  + 하위 클래스 내에 선언되지 않은 정적 메소드는 호출되어서는 안된다.
* class Base { /\*\* @nocollapse \*/ static foo() {} }
* class Sub extends Base {}
* function callFoo(cls) { cls.foo(); } // Bad: 동적 인스턴스를 통해 호출되어서는 안된다.

Sub.foo(); // Bad: 하위 클래스 내에 선언되지 않은 정적 메소드는 호출되어서는 안된다.

* **prototype**을 직접 조작하지 않고 class를 사용한다.
* // 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;
* }

}

* 상속은**extends**를 사용한다.
* // 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];
* }

}

* 메소드의 반환값으로 **this**를 사용하는 것으로 **메소드 채이닝**을 할 수 있다.
* // 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);

* toString() 메소드를 오버라이딩할 수 있지만 **사이드 이펙트**가 나타나지 않도록 해야한다.
* class Jedi {
* constructor(options = {}) {
* this.name = options.name || 'no name'; // null이 반환되는 것을 방지
* }
* getName() {
* return this.name;
* }
* toString() {
* return `Jedi - ${this.getName()}`;
* }

}

**함수 📤**

* 함수 내에 또다른 함수를 선언할 수 있다. 함수에 이름이 필요한 경우 **const**를 사용한다.
* 함수식 보다는 함수 선언을 사용한다.
* // bad
* const foo = function () { // foo만 hoist됨
* };
* // good
* function foo() { // 함수 전체가 hoist됨

}

* 함수 이외의 블록(i.e. if, while) 안에서 함수를 선언하지 않는다.
* 함수의 마라미터에 **arguments**를 사용하지 않는다. 이는 오브젝트의 참조를 덮어쓰는 것을 야기한다.
* **arguments**를 사용하는 것 대신에 **Rest 신택스( ... )**를 사용한다.
* // bad
* function concatenateAll() {
* const args = Array.prototype.slice.call(arguments);
* return args.join('');
* }
* // good
* function concatenateAll(...args) {
* return args.join('');

}

* 함수의 파라미터를 조작하는 것보다는 **파라미터 기본값**을 사용한다.
* // really bad
* function handleThings(opts) {
* // 이는 좋지 않습니다. 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 = {}) {
* // ...

}

* 사이드 이펙트가 발생할 파라미터 기본값의 사용을 지양한다.
* var b = 1;
* // bad
* function count(a = b++) {
* console.log(a);
* }
* count(); // 1
* count(); // 2
* count(3); // 3

count(); // 3

* 파라미터 기본값은 가장 뒤쪽에 둔다.
* // bad
* function handleThings(opts = {}, name) {
* // ...
* }
* // good
* function handleThings(name, opts = {}) {
* // ...

}

* 함수식을 사용해야만한다면, 화살표 함수(Arrow Function)을 사용하라.
* // 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;

});

**화살표 함수 (Arrow Function) ➡️**

* 함수 바디가 하나의 식으로 구성된 경우, 중괄호와 return문을 생략할 수 있다.  
  중괄호를 생략하지 않을 경우, **return**문을 포함시켜야 한다.
* // 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}.`;

});

* 식이 여러 줄에 걸쳐있을 경우에 가독성을 위해 **소괄호**로 감싸 사용한다.
* // 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.'

));

* 함수의 인수가 하나인 경우 **소괄호**를 생략할 수 있다.
* // good
* [1, 2, 3].map(x => x \* x);
* // good

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

**문자열 👅**

* 문자열을 선언할 때는 **작은 따옴표( ' )**를 사용한다.  
  문자열 내에 작은 따옴표가 포함될 경우 **템플릿 리터럴( ` ` )**을 사용한다.
* // bad
* const name = "Capt. Janeway";
* // good
* const name = 'Capt. Janeway';
* // good

const name = `Mark J' Maclachlan`;

* 80글자 이상의 긴 문자열을 여러 줄에 걸쳐 쓰기 위해서는 **템플릿 리터럴** 혹은 **문자열 연결( + )**을 사용한다.
* // 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.';
* 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.`;

* 동적으로 문자열을 생성할 경우에는 문자열 연결이 아닌 **템플릿 리터럴**을 사용한다.
* // 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}?`;

}

* **eval()** 함수를 사용하지 않는다. 취약점이 많은 함수이다.

**숫자 🏦**

* 숫자는 10진수, 16진수(0x), 8진수(0o), 2진수(0b)로 표현될 수 있다.  
  x, o, b 바로 다음에는 불필요한 0을 포함시키지 않는다.

**제어문 🚀**

* **반복문**
  + 일반적인 for문 보다는 **for-of**를 사용한다.  
    하지만, 가능하다면 **map()**, **reduce()**와 같은 고급 함수를 사용하라.
  + 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;

* **스위치문**
  + 다음 case 구문이 실행되어야 한다면 주석으로 이를 남긴다.
  + default문은 항상 마지막에 위치한다.
  + switch (input) {
  + case 1:
  + case 2:
  + prepareOneOrTwo();
  + // fall through
  + case 3:
  + handleOneTwoOrThree();
  + break;
  + default:
  + handleLargeNumber(input);

}

* **this**
  + **this**키워드는 클래스 생성자와 메소드, 혹은 그 안에서 선언된 화살표 함수 내에서만 사용한다.
* **조건식, 등가식**
  + **==**, **!=** 보다는 **===**, **!==**를 사용하라.
  + 조건식은 **ToBoolean 메소드**에 의한 강제 형변환으로 평가되어 아래의 규칙을 따른다.
    - 오브젝트는 true로 평가
    - undefined는 false로 평가
    - null 은 false 로 평가
    - 값은 true로 평가. 단, +0, -0, or NaN 의 경우는 false로 평가
    - 문자열은 true로 평가. 단, 빈문자( '' )의 경우는 false로 평가
  + if ([0]) {
  + // true
  + // An array is an object, objects evaluate to true
  + // 배열은 오브젝트이므로 true 로 평가됩니다.

}

**목차**

* 네이밍 기본
* 파일 및 패키지
* 변수
* 함수
* 객체
* 클래스
* 기타

**네이밍 기본 🏠**

* **단일 글자**로 이름을 짓지 않고 이름을 통해 쓰임새를 알 수 있도록 한다.
* // bad
* function q() {
* // ...
* }
* // good
* function query() {
* // ...

}

* 이름의 맨 앞이나 맨 뒤쪽에 **밑줄( \_ )**을 사용하지 않는다.
* // bad
* this.\_\_firstName\_\_ = 'Panda';
* this.firstName\_ = 'Panda';
* this.\_firstName = 'Panda';
* // good

this.firstName = 'Panda';

* **this**를 변수의 값으로 사용하지 않는다. 필요하다면 **화살표 함수(Arrow Function)**이나 **바인딩**을 사용하라.
* // bad
* function foo() {
* const self = this;
* return function () {
* console.log(self);
* };
* }
* // good
* function bar() {
* return () => {
* console.log(this);
* };

}

* 가독성을 위해 약어는 **모두 대문자** 혹은 **모두 소문자**로 표기한다.
* // bad
* import SmsContainer from './containers/SmsContainer';
* // bad
* const HttpRequests = [
* // ...
* ];
* // good
* import SMSContainer from './containers/SMSContainer';
* // good
* const HTTPRequests = [
* // ...
* ];
* // also good
* const httpRequests = [
* // ...
* ];
* // best
* import TextMessageContainer from './containers/TextMessageContainer';
* // best
* const requests = [
* // ...

];

* export되는 파일 내의 모든 상수는 **모두 대문자**로 표기한다.
* // bad
* const PRIVATE\_VARIABLE = 'should not be unnecessarily uppercased within a file';
* // bad
* export const THING\_TO\_BE\_CHANGED = 'should obviously not be uppercased';
* // bad
* export let REASSIGNABLE\_VARIABLE = 'do not use let with uppercase variables';
* // ---
* // allowed but does not supply semantic value
* export const apiKey = 'SOMEKEY';
* // better in most cases
* export const API\_KEY = 'SOMEKEY';
* // ---
* // bad - unnecessarily uppercases key while adding no semantic value
* export const MAPPING = {
* KEY: 'value'
* };
* // good
* export const MAPPING = {
* key: 'value'

};

* 이름에 복수형을 표기하지 않는다.
* //bad
* let delivery\_notes = ["one", "two"];
* // good

let delivery\_note\_list = ["one", "two"];

* 줄일말을 사용하지 않는다.
* //bad
* let del\_note = 1;
* // good

let delivery\_note = 1;

**파일 및 패키지 🎁**

* 파일의 이름은 소문자로 표기한다.
* // bad
* LonDon.png
* HELLOWORLD.pdf
* APP.js
* // good
* london.png
* helloworld.pdf

app.js

* 패키지의 이름은 **lowerCamelCase**로 표기한다.
* // bad
* my.examplecode.deepspace
* my.example\_code.deep\_space
* // good

my.exampleCode.deepSpace

* 파일의 이름은 **default export의 이름**과 일치해야한다.
* // file 1 contents
* class CheckBox {
* // ...
* }
* export default CheckBox;
* // file 2 contents
* export default function fortyTwo() { return 42; }
* // file 3 contents
* export default function insideDirectory() {}
* // in some other file
* // bad
* import CheckBox from './checkBox'; // PascalCase import/export, camelCase filename
* import FortyTwo from './FortyTwo'; // PascalCase import/filename, camelCase export
* import InsideDirectory from './InsideDirectory'; // PascalCase import/filename, camelCase export
* // bad
* import CheckBox from './check\_box'; // PascalCase import/export, snake\_case filename
* import forty\_two from './forty\_two'; // snake\_case import/filename, camelCase export
* import inside\_directory from './inside\_directory'; // snake\_case import, camelCase export
* import index from './inside\_directory/index'; // requiring the index file explicitly
* import insideDirectory from './insideDirectory/index'; // requiring the index file explicitly
* // good
* import CheckBox from './CheckBox'; // PascalCase export/import/filename
* import fortyTwo from './fortyTwo'; // camelCase export/import/filename
* import insideDirectory from './insideDirectory'; // camelCase export/import/directory name/implicit "index"

// ^ supports both insideDirectory.js and insideDirectory/index.js

**변수 🌓**

* 변수의 이름은 **lowerCamelCase**로 표기한다.  
  단, export되는 파일 내의 상수는 예외.
* 변수의 이름은 **알파벳**으로 시작해야한다.
* // bad
* let 123Number = 123;
* let HELLO\_WORLD = "Hello World";
* // good
* let number = 369;

let helloString = "Hello World";

**함수 📤**

* 함수는 **lowerCamelCase**로 표기한다.
* // bad
* function MyFunction() {...}
* // good

function myFunction() {...}

* 함수의 이름은 동사 또는 동사구문으로 표기한다.
* // bad
* function whereIsCamera() { ... }
* // good
* function findCamera() { ... }
* function getFoo() { ... } // getter
* function setBar() { ... } // setter

function hasCoo() { ... } // booleans

* 함수를 default export할 때는 **camelCase**로 표기한다.  
  단, 함수의 이름이 파일의 이름과 구분되어야 한다.
* function makeStyleGuide() {
* // ...
* }

export default makeStyleGuide;

* 함수 라이브러리를 export할 때는 **PascalCase**로 표기한다.
* 함수의 파라미터는 **lowerCamelCase**로 표기한다.  
  단, 한글자의 파라미터는 public 메소드에서는 사용하지 않는다.
* // bad
* function someFunction(SOMEVALUE, SOMEARRAY) { ... }
* // good

function someFunction(someValue, someArray) { ... }

* 템플릿 함수의 파라미터는 모두 간결해야하고 **한글자** 또는 **한단어**여야 한다.  
  또, **모두 대문자**로 표기한다.

**객체 🎍**

* 객체의 이름은 **lowerCamelCase**로 표기한다.
* // bad
* const OBJEcttsssss = {};
* const this\_is\_my\_object = {};
* function c() {}
* // good
* const thisIsMyObject = {};

function thisIsMyFunction() {}

* 객체를 export할 때는 **PascalCase**로 표기한다.
* const AirbnbStyleGuide = {
* es6: {
* },
* };

export default AirbnbStyleGuide;

**클래스 📇**

* 클래스나 생성자의 이름은 **PascalCase**로 표기한다.
* // 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',

});

* 클래스의 이름은 **명사** 또는 **명사구문**으로 표기한다.  
  또한, 인터페이스의 경우 명사 대신 **형용사** 또는 **형용사구문**으로 표기할 수 있다.
* 클래스를 export할 때는 **PascalCase**로 표기한다.