

**『SEMA』 차세대 정보시스템 구축**

개발표준정의서
(UI\_Javascript 표준가이드)

[AA\_A\_10]

Ver. 1.01

2020.11.09



개 정 이 력

| **버전** | **작성일** | **변경 내용** | **작성자** | **승인자** |
| --- | --- | --- | --- | --- |
| 1.00 | 2020.11.09 | 최초 작성 | 홍기웅  | 윤화병 |
| 1.01 | 2020.11.26 | 오타 수정 (2칸 띄기 🡪 4칸, 80자 🡪 132자) | 홍기웅  | 윤화병 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

Table of Contents

[Table of Contents 3](#_Toc55305851)

[1. 문서 개요 5](#_Toc55305852)

[1.1. 개요 5](#_Toc55305853)

[1.1.1. 목적 5](#_Toc55305854)

[2. 코딩 가이드 6](#_Toc55305855)

[2.1. 형식 6](#_Toc55305856)

[2.1.1. 중괄호 6](#_Toc55305857)

[2.1.2. 들여쓰기 7](#_Toc55305858)

[2.1.3. 구문 9](#_Toc55305859)

[2.1.4. 공백 9](#_Toc55305860)

[2.2. 변수 13](#_Toc55305861)

[2.3. 배열 15](#_Toc55305862)

[2.4. 객체 17](#_Toc55305863)

[2.5. 클래스 20](#_Toc55305864)

[2.6. 함수 22](#_Toc55305865)

[2.7. 화살표 함수 (Arrow Function) 24](#_Toc55305866)

[2.8. 문자열 26](#_Toc55305867)

[2.9. 숫자 27](#_Toc55305868)

[2.10. 제어문 27](#_Toc55305869)

[2.10.1. 반복문 27](#_Toc55305870)

[2.10.2. 스위치문 28](#_Toc55305871)

[2.10.3. this 28](#_Toc55305872)

[2.10.4. 조건식, 등가식 28](#_Toc55305873)

[3. 네이밍 가이드 30](#_Toc55305874)

[3.1. 네이밍 기본 30](#_Toc55305875)

[3.2. 파일 및 패키지 32](#_Toc55305876)

[3.3. 변수 34](#_Toc55305877)

[3.4. 함수 34](#_Toc55305878)

[3.5. 객체 36](#_Toc55305879)

[3.6. 클래스 36](#_Toc55305880)

# 문서 개요

## 개요

### 목적

본 문서는 프로젝트명(이하 ‘프로젝트’라 한다.)의 성공적인 구축을 위하여 소프트웨어 개발 과정에 참여하는 Nexacro Platform화면 개발자에게 일반적인 Javascirpt 코딩 스타일을 전달하는 것에 목적이 있습니다. 이를 통해 개발자간의 원활한 협업을 가능하게 하고 유지보수를 위한 코드 가독성을 높이고자 합니다.

본 문서에서 설명하는 Javascript 표준과 Nexacro Script 표준이 서로 상이한 경우에는 Nexacro Script 표준을 우선으로 적용하도록 합니다.

본 문서는 크게 코딩가이드와 네이밍가이드로 나누어 설명하고 있습니다.

# 코딩 가이드

## 형식

### 중괄호

* 중괄호는 하나의 구문만을 포함하고 있더라도 모든 제어문*(if, else, for, do, while..)*에 사용되어야 한다.

|  |
| --- |
| // badif (someVeryLongCondition()) doSomething();// Nofor (let i = 0; i < foo.length; i++) bar(foo[i]);// Yesif (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.

|  |
| --- |
| // Yesfunction doNothing() {} // Noif (condition) {// …} else if (otherCondition) {} else { // …}// Notry {  // …} catch (e) {} |

### 들여쓰기

* 새로운 블록이 나타날 때 마다 **네 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 때에는 함수를 이전 레벨보다 4칸 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$('#items') .find('.selected') .highlight() .end() .find('.open') .updateCount();// badconst leds = stage.selectAll('.led').data(data).enter().append('svg:svg').classed('led', true) .attr('width', (radius + margin) \* 2).append('svg:g') .attr('transform', `translate(${radius + margin},${radius + margin})`) .call(tron.led);// goodconst leds = stage.selectAll('.led') .data(data) .enter().append('svg:svg') .classed('led', true) .attr('width', (radius + margin) \* 2) .append('svg:g') .attr('transform', `translate(${radius + margin},${radius + margin})`) .call(tron.led);// goodconst leds = stage.selectAll('.led').data(data); |

### 구문

* 한 줄에는 하나의 구문만을 작성한다.
* 모든 구문의 끝에는 **세미콜론**을 붙인다.

|  |
| --- |
| // badlet name = 'Cada'doSomething() saySomething()// goodlet name = 'Cada';doSomething();saySomething(); |

### 공백

* 한 줄이 **132자**를 넘기지 않도록 권고한다.
* 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.

|  |
| --- |
| // badif (foo) { return bar;}return baz;// goodif (foo) { return bar;}return baz;// badconst obj = { foo() { }, bar() { },};return obj;// goodconst obj = { foo() { }, bar() { },};return obj;// badconst arr = [ function foo() { }, function bar() { },];return arr;// goodconst 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)

|  |
| --- |
| // badif(myCondition) { // do something}// goodif (myCondition) { // do something} |

* + - **else**, **catch**와 같은 키워드와 닫는 중괄호 사이

|  |
| --- |
| // badif(myCondition) { // do something}else { }// goodif (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

|  |
| --- |
| // badconst name = someCondotion?20:30;// goodconst name = someCondotion ? 20 : 30; |

* + - **반점( , )**, **세미콜론( ; )** 다음 (단, 반점과 세미콜론 전에는 사용하지 않음)

|  |
| --- |
| // badconst arr = [1,2,3,4];// goodconst arr = [1, 2, 3, 4]; |

* + - 오브젝트 리터럴 내에서 **콜론( : )** 다음

|  |
| --- |
| // badconst myobject = { "name":"Chris", "age":34};// goodconst myobject = { "name": "Chris", "age": 34}; |

* + 소괄호 사이, 대괄호 사이에는 칸 공백을 사용하지 않는다.

|  |
| --- |
| // badfunction bar( foo ) { return foo;}// goodfunction bar(foo) { return foo;}// badif ( foo ) { console.log(foo);}// goodif (foo) { console.log(foo);}// badconst foo = [ 1, 2, 3 ];console.log(foo[ 0 ]);// goodconst foo = [1, 2, 3];console.log(foo[0]); |

* + 중괄호 사이에는 칸 공백을 사용한다.

|  |
| --- |
| // badconst foo = {clark: 'kent'};// goodconst foo = { clark: 'kent' }; |

## 변수

* 한 줄에 하나의 변수를 선언한다.

|  |
| --- |
| // badlet a = 1, b = 3;// goodlet a = 1;let b = 2; |

* 지역 변수는 그 변수를 포함하는 블록 시작에서 선언하지 않고, 사용 범위를 최소화하기 위해 사용되는 지점과 가장 가까운 곳에서 선언한다.

|  |
| --- |
| // goodfunction() { test(); console.log('doing stuff..');  const name = getName(); if (name === 'test') { return false; } return name;}// bad - 함수 호출의 부적절한 위치function(hasName) { const name = getName(); if (!hasName) { return false; } this.setFirstName(name); return true;}// goodfunction(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.

|  |
| --- |
| // badlet i, len, dragonball, items = getItems(), goSportsTeam = true;// badlet i;const items = getItems();let dragonball;const goSportsTeam = true;let len;// goodconst goSportsTeam = true;const items = getItems();let dragonball;let i;let length; |

## 배열

* 배열을 선언할 때는 Array 생성자가 아닌 **리터럴 구문**을 사용한다.

|  |
| --- |
| // badconst items = new Array();// goodconst items = []; |

* 배열에 값을 넣을 때는 **Array.push**를 사용한다.

|  |
| --- |
| const someStack = [];// badsomeStack[someStack.length] = 'abracadabra';// goodsomeStack.push('abracadabra'); |

* 배열을 복사할 때는 배열의 **확장연산자 ( ... )**를 사용한다.

|  |
| --- |
| // badconst len = items.length;const itemsCopy = [];let i;for (i = 0; i < len; i++) { itemsCopy[i] = items[i];}// goodconst 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.

|  |
| --- |
| // badconst arr = [];arr["str"] = 32;// goodconst 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.

|  |
| --- |
| // badconst item = new Object();// goodconst item = {}; |

* **예약어**를 키로 사용하지 않는다. 대신, 동의어를 사용한다.

|  |
| --- |
| // badconst superman = { default: { clark: 'kent' }, private: true,};// goodconst superman = { defaults: { clark: 'kent' }, hidden: true,}; |

* 키는 **큰 따옴표( " )**를 씌운다.
단, 큰 따옴표를 씌운 키와 씌우지 않은 키를 한 오브젝트에 동시에 사용하지 않는다.

|  |
| --- |
| // badvar my\_object = { key: "value",};var my\_object = { key: "value", "key2": "value2",};// goodvar 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}`;}// badconst obj = { id: 5, name: 'San Francisco',};obj[getKey('enabled')] = true;// goodconst obj = { id: 5, name: 'San Francisco', ［getKey('enabled')]: true}; |

* 메소드 단축 구문을 사용한다.

|  |
| --- |
| // badconst atom = { value: 1, addValue: function (value) { return atom.value + value; },};// goodconst atom = { value: 1, addValue(value) { return atom.value + value; },}; |

* 속성 단축 구문을 사용한다.
단, 선언의 시작 부분에 그룹화하여 작성한다.

|  |
| --- |
| const lukeSkywalker = 'Luke Skywalker';// badconst obj = { lukeSkywalker: lukeSkywalker,};// goodconst obj = { lukeSkywalker,};// badconst obj = { episodeOne: 1, twoJediWalkIntoACantina: 2, lukeSkywalker, episodeThree: 3, mayTheFourth: 4, anakinSkywalker,};// goodconst 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 한다.
	+ 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.

|  |
| --- |
| // badfunction Queue(contents = []) { this.\_queue = [...contents];}Queue.prototype.pop = function() { const value = this.\_queue[0]; this.\_queue.splice(0, 1); return value;}// goodclass Queue { constructor(contents = []) { this.\_queue = [...contents]; } pop() { const value = this.\_queue[0]; this.\_queue.splice(0, 1); return value; }} |

* 상속은**extends**를 사용한다.

|  |
| --- |
| // badconst inherits = require('inherits');function PeekableQueue(contents) { Queue.apply(this, contents);}inherits(PeekableQueue, Queue);PeekableQueue.prototype.peek = function() { return this.\_queue[0];}// goodclass PeekableQueue extends Queue { peek() { return this.\_queue[0]; }} |

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

|  |
| --- |
| // badJedi.prototype.jump = function() { this.jumping = true; return true;};Jedi.prototype.setHeight = function(height) { this.height = height;};const luke = new Jedi();luke.jump(); // => trueluke.setHeight(20); // => undefined// goodclass 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.

|  |
| --- |
| // badconst foo = function () { // foo만 hoist됨};// goodfunction 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.

|  |
| --- |
| // badfunction concatenateAll() { const args = Array.prototype.slice.call(arguments); return args.join('');}// goodfunction concatenateAll(...args) { return args.join('');} |

* 함수의 파라미터를 조작하는 것보다는 **파라미터 기본값**을 사용한다.

|  |
| --- |
| // really badfunction handleThings(opts) { // 이는 좋지 않습니다. 함수의 파라메터를 변이시키면 안됩니다. // 만약 opts가 falsy 하다면 바라는데로 오브젝트가 설정됩니다. // 하지만 미묘한 버그를 일으킬지도 모릅니다. opts = opts || {}; // ...}// still badfunction handleThings(opts) { if (opts === void 0) { opts = {}; } // ...}// goodfunction handleThings(opts = {}) { // ...} |

* 사이드 이펙트가 발생할 파라미터 기본값의 사용을 지양한다.

|  |
| --- |
| var b = 1;// badfunction count(a = b++) { console.log(a);}count(); // 1count(); // 2count(3); // 3count(); // 3 |

* 파라미터 기본값은 가장 뒤쪽에 둔다.

|  |
| --- |
| // badfunction handleThings(opts = {}, name) { // ...}// goodfunction 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.

|  |
| --- |
| // badconst name = "Capt. Janeway";// goodconst name = 'Capt. Janeway';// goodconst name = `Mark J' Maclachlan`; |

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

|  |
| --- |
| // badconst 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.';// badconst 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.';// goodconst 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 becauseof Batman. When you stop to think about how Batman had anything to dowith this, you would get nowhere fast.`; |

* 동적으로 문자열을 생성할 경우에는 문자열 연결이 아닌 **템플릿 리터럴**을 사용한다.

|  |
| --- |
| // badfunction sayHi(name) { return 'How are you, ' + name + '?';}// badfunction sayHi(name) { return ['How are you, ', name, '?'].join();}// goodfunction 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];// badlet sum = 0;for (let num of numbers) { sum += num;}sum === 15;// goodlet 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.} |

# 네이밍 가이드

## 표기법

명명 규칙에서 사용되는 표기법의 종류는 다음과 같다.

|  |  |  |  |
| --- | --- | --- | --- |
| 명칭 | 설명 | 사용예 | 적용 대상 |
| 파스칼 표기법 (Pascal Case) | 첫 문자를 대문자로 시작하며, 의미 구분을 위해 대문자를 사용 | **U**ser**M**anual | Java,  |
| 카멜표기법 (Camel Case) | 첫 문자를 소문자로 시작하며, 의미 구분을 위해 대분자를 사용 | **u**ser**M**anual | 변수, 함수명 |
| 헝가리안 표기법 (Hungarian Case) | 사전에 정의된 접두어를 사용 | **n**UserManual | 변수 |
| 소문자 표기법 (Lower Case) | 전체를 소문자로 사용 | usermanual | Java 패키지명 |
| 대문자 표기법 (Upper Case) | 전체를 대문자로 사용 | USERMANUAL | 상수명 |
| 언더스코어 표기법 (Underscored Case) | 의미 구분을 위해 언더스코어를 사용 | user**\_**manual |  |
| 하이픈 표기법 (Hyphen Case) | 의미 구분을 위해 하이픈을 사용. | user**-**manual | url |

## 네이밍 기본

* **단일 글자**로 이름을 짓지 않고 이름을 통해 쓰임새를 알 수 있도록 한다.

|  |
| --- |
| // badfunction q() { // ...}// goodfunction query() { // ...} |

* 이름의 맨 앞이나 맨 뒤쪽에 **밑줄( \_ )**을 사용하지 않는다.

|  |
| --- |
| // badthis.\_\_firstName\_\_ = 'Panda';this.firstName\_ = 'Panda';this.\_firstName = 'Panda';// goodthis.firstName = 'Panda'; |

* **this**를 변수의 값으로 사용하지 않는다. 필요하다면 **화살표 함수(Arrow Function)**이나 **바인딩**을 사용하라.

|  |
| --- |
| // badfunction foo() { const self = this; return function () { console.log(self); };}// goodfunction bar() { return () => { console.log(this); };} |

* 가독성을 위해 약어는 **모두 대문자** 혹은 **모두 소문자**로 표기한다.

|  |
| --- |
| // badimport SmsContainer from './containers/SmsContainer';// badconst HttpRequests = [ // ...];// goodimport SMSContainer from './containers/SMSContainer';// goodconst HTTPRequests = [  // ...];// also goodconst httpRequests = [ // ...];// bestimport TextMessageContainer from './containers/TextMessageContainer';// bestconst requests = [ // ...]; |

* export되는 파일 내의 모든 상수는 **모두 대문자**로 표기한다.

|  |
| --- |
| // badconst PRIVATE\_VARIABLE = 'should not be unnecessarily uppercased within a file';// badexport const THING\_TO\_BE\_CHANGED = 'should obviously not be uppercased';// badexport let REASSIGNABLE\_VARIABLE = 'do not use let with uppercase variables';// ---// allowed but does not supply semantic valueexport const apiKey = 'SOMEKEY';// better in most casesexport const API\_KEY = 'SOMEKEY';// ---// bad - unnecessarily uppercases key while adding no semantic valueexport const MAPPING = { KEY: 'value'};// goodexport const MAPPING = { key: 'value'}; |

* 이름에 복수형을 표기하지 않는다.

|  |
| --- |
| //badlet delivery\_notes = ["one", "two"];// goodlet delivery\_note\_list = ["one", "two"]; |

* 줄일말을 사용하지 않는다.

|  |
| --- |
| //badlet del\_note = 1;// goodlet delivery\_note = 1; |

## 파일 및 패키지

* 파일의 이름은 소문자로 표기한다.

|  |
| --- |
| // badLonDon.pngHELLOWORLD.pdfAPP.js// goodlondon.pnghelloworld.pdfapp.js |

* 패키지의 이름은 **lowerCamelCase**로 표기한다.

|  |
| --- |
| // badmy.examplecode.deepspacemy.example\_code.deep\_space// goodmy.exampleCode.deepSpace |

* 파일의 이름은 **default export의 이름**과 일치해야한다.

|  |
| --- |
| // file 1 contentsclass CheckBox { // ...}export default CheckBox;// file 2 contentsexport default function fortyTwo() { return 42; }// file 3 contentsexport default function insideDirectory() {}// in some other file// badimport CheckBox from './checkBox'; // PascalCase import/export, camelCase filenameimport FortyTwo from './FortyTwo'; // PascalCase import/filename, camelCase exportimport InsideDirectory from './InsideDirectory'; // PascalCase import/filename, camelCase export// badimport CheckBox from './check\_box'; // PascalCase import/export, snake\_case filenameimport forty\_two from './forty\_two'; // snake\_case import/filename, camelCase exportimport inside\_directory from './inside\_directory'; // snake\_case import, camelCase exportimport index from './inside\_directory/index'; // requiring the index file explicitlyimport insideDirectory from './insideDirectory/index'; // requiring the index file explicitly// goodimport CheckBox from './CheckBox'; // PascalCase export/import/filenameimport fortyTwo from './fortyTwo'; // camelCase export/import/filenameimport 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.

|  |
| --- |
| // badlet 123Number = 123;let HELLO\_WORLD = "Hello World";// goodlet number = 369;let helloString = "Hello World"; |

## 함수

* 함수는 **lowerCamelCase**로 표기한다.

|  |
| --- |
| // badfunction MyFunction() {...}// goodfunction myFunction() {...} |

* 함수의 이름은 동사 또는 동사구문으로 표기한다.

|  |
| --- |
| // badfunction whereIsCamera() { ... }// goodfunction findCamera() { ... }function getFoo() { ... } // getterfunction setBar() { ... } // setterfunction 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.

|  |
| --- |
| // badfunction someFunction(SOMEVALUE, SOMEARRAY) { ... }// goodfunction someFunction(someValue, someArray) { ... } |

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

## 객체

* 객체의 이름은 **lowerCamelCase**로 표기한다.

|  |
| --- |
| // badconst OBJEcttsssss = {};const this\_is\_my\_object = {};function c() {}// goodconst thisIsMyObject = {};function thisIsMyFunction() {} |

* 객체를 export할 때는 **PascalCase**로 표기한다.

|  |
| --- |
| const AirbnbStyleGuide = { es6: { },};export default AirbnbStyleGuide; |

## 클래스

* 클래스나 생성자의 이름은 **PascalCase**로 표기한다.

|  |
| --- |
| // badfunction user(options) { this.name = options.name;}const bad = new user({ name: 'nope',});// goodclass 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.