JavaScript

var, let, const – 김근형 강사

var

- var 키워드
  - 이전 ES5.1 버전에서는 var 키워드를 통해 변수를 선언해 사용한다.
  - 변수에 var를 붙여 사용할 수도 있지만 모든 변수에는 앞에 var가 붙으므로 생략도 가능하였다.
  - 하지만 Scope의 구분이 되지 않아 전역변수와 지역변수의 혼동이 오는 사례가 상당히 많았다.
  - 일부는 var와 네이밍을 통해 실제 전역변수와 지역변수를 구분하여 쓸 수는 있었지만 역시나 한계가 있었다.

```
var one = 100;
function get(){
  var one = 300;
  console.log("함수:", one);
}
get();
console.log("글로벌:", one);
```

```
함수: 300
글로벌: 100
>
```

```
one = 100;
function get(){
  one = 300;
  console.log("함수:", one);
}
get();
console.log("글로벌:", one);
```

```
함수: 300
글로벌: 300
>
```

```
var sports = "축구";
if (sports){
  var sports = "농구";
  console.log("블록: ", sports);
}
console.log("글로벌: ", sports);
```

```
블록:  농구
글로벌:  농구
```

use strict

- use strict
  - 기존의 자바스크립트 코드에 대해 좀 더 엄격한 검사를 실행시키고 싶을 때 쓴다.
  - use strict는 상단에 선언이 가능하며 흔히 발생하는 코딩 실수를 잡아내서 예외를 발생시킨다.
  - 상대적으로 안전하지 않은 액션이 발생하는 것을 방지할 때 예외를 발생시킨다.
  - 혼란스럽거나 제대로 고려되지 않은 기능들을 비활성화 시킨다.

use strict

- use strict를 씀으로 해서 일어나는 특징
  - 선언없이 전역 변수 금지
  - get으로 선언된 객체 수정 불가
  - extensible 특성이 false로 설정된 객체에 속성을 확장 불가(확장 불가 객체에 확장 불가능)
  - delete를 호출 불가
  - 리터럴 객체는 동일한 이름의 property를 가질 수 없다.(하지만ES6는 가능함)
  - 함수의 동일한 매개 변수 이름을 선언하는 것이 불가능
  - 8진수 숫자 리터럴 및 이스케이프 문자 사용 불가
  - primitive values의 속성 설정 불가능

use strict

- use strict를 씀으로 해서 일어나는 특징
  - with 사용 불가
  - eval 함수는 주변 스코프에 새로운 변수 추가 불가능.
  - eval과 arguments는 변수 또는 함수, 매개 변수의 이름으로 사용할 수 없다
  - 인자값을 수정함으로 arguments의 값이 수정되지 않는다.
  - callee를 지원하지 않는다.
  - this의 값이 null 또는 undefined인 경우 전역 객체로 변환하지 않는다.
  - callee, caller를 통해 stack 검색이 불가능
  - arguments의 caller를 지원하지 않는다.
  - 예약된 키워드의 이름으로 변수 또한 함수를 생성 불가
  - 함수 선언은 스크립트나 함수의 최상위에서 해야 한다.

use strict

- use strict
  - 실제 자바스크립트의 성능을 떨어뜨리고 에러를 발생시키는 여러가지 기능들을 제한하기 위해 이러한 모드가 도입되었다.
  - 하지만 근본적인 문제는 해결되지 않음

let

- let 키워드
  - let 키워드는 var처럼 사용이 가능한 변수 선언 키워드

```
let var_name1 [= value1] {[, var_name2 [= value2]] [, var_name3 [= value3]] ... }
```

  - let 키워드는 아래와 같은 특징을 가진다.
    - 함수 안에 작성한 let 변수는 함수 안에서만 쓰인다.
    - 블록 {} 밖에 같은 이름의 변수가 있어도 스코프가 다르므로 변수 각각에 값을 설정할 수 있으며 변수 값이 유지된다.
    - 블록 안에 블록을 계층적으로 작성하면 각각의 블록이 스코프가 된다.
    - 같은 스코프에서는 같은 이름의 let 변수를 선언할 수 없다.
    - let 변수는 호이스팅(hoisting)은 되지만 선언 부분 전까지 접근이 되지 않는다.

let

- Let 키워드

```
let book;
let sports = "축구";

sports = "농구";

// 같은 스코프 내에 같은 이름의 let 변수는 사용 불가
// let sports = "배구";

let one = 1, two = 2, three;

// 변수를 나열하여 선언할 경우 뒤의 변수 에는 let을 앞에 붙일 필요가 없음
// let four = 4, let five = 5;
// let과 var는 혼용 사용이 안됨
// let six = 6, var seven = 7;
```

let

- Let 블록 스코프
  - 블록 스코프는 {}를 통해서 영역을 구분하는 것.
  - 블록 {} 안과 밖의 변수 이름이 같더라도 스코프가 다르므로 변수가 선언되고 각 변수에 할당된 값이 대체되지 않고 유지된다.

```
let sports = "축구";
if (sports){
  let sports = "농구";
  console.log("블록: ", sports);
}
console.log("글로벌: ", sports);
```

```
블록:  농구
글로벌:  축구
```

let

- this
  - Var를 통해 전역에서 변수를 선언하게 되면 this를 통해 접근이 가능했었다.
  - 하지만 let을 통해 접근하게 되면 this를 통한 전역변수의 접근이 불가능하다.
  - Let을 사용한 변수는 전역으로 사용될 수 없다.

```
var music = "음악";
console.log(this.music);

let sports = "축구";
console.log(this.sports);
```

```
음악
undefined
```

let

- function
  - 함수도 스코프를 가지므로 하나의 블록 스코프가 된다.
  - {}안에 {}를 두는 형태의 구조를 계층적 스코프 구조라고 한다.

```
let sports = "축구", music = "재즈";

function get(){
  let music = "클래식";
  console.log(music);
  console.log(sports);
}
get();
```

```
클래식
축구
```

let

- function

```
var sports = "축구";
let music = "재즈";

function get(){
  var sports = "농구";
  let music = "클래식";

  console.log("1:", sports);
  console.log("2:", this.sports);
  console.log("3:", this.music);
};
window.get();
get();
```

```
1: 농구
2: 축구
3: undefined
1: 농구
Uncaught TypeError: Cannot read property 'sports' of
undefined
    at get (let-function-this.js:12)
    at let-function-this.js:16
```

"use strict" 적용 안할 시

```
1: 농구
2: 축구
3: undefined
1: 농구
2: 축구
3: undefined
```

let

- try~catch
  - Try-catch 문도 실제 if문과 동일한 스코프 특성을 가진다.

```
let sports = "축구";
try {
  let sports = "농구";
  console.log(sports);
} catch (e) {};

console.log(sports);
```

```
농구
축구
```

let

- switch~case
  - switch-case 문도 위의 if문과 동일한 스코프 특성을 가진다.

```
var count = 1;
let sports = "축구";
switch (count) {
  case 1:
    let sports = "농구";
    console.log(sports);
};
console.log(sports);
```

```
농구
축구
```

let

- 호이스팅
  - 자바스크립트 및 액션스크립트 코드를 인터프리터가 로드할 때, 변수의 정의가 그 범위에 따라 선언과 할당으로 분리되어 변수의 선언을 항상 컨텍스트 내의 최상위로 끌어올리는 것을 의미한다.
  - 이는 오로지 변수에만 해당되는 것은 아니고 함수도 가능하며, 자바스크립트에서 함수의 호출을 첫 줄에서 하고 마지막 줄에 함수를 정의해도 문제없이 작동되도록 하는 유용한 특성이다.

let

- 호이스팅
  - 변수의 경우 어디서 어떻게 선언을 하더라도 항상 컨텍스트 내의 최상위로 끌고 올라온 뒤 'undefined'를 할당해 둔다.
  - 이후 함수 선언을 끌고 올라오고 난 뒤 변수의 할당과 함수의 실행문을 순서대로 가져온다.
  - 즉 호이스팅은 자바스크립트 인터프리터가 코드를 읽는 방식이며, 이를 이해해야 원치 않는 'undefined'가 출력되는 것을 막을 수 있다

```
function sum(a, b) {
    var x = add(a, b);
    return x;

    function add(c, d) {
        var result = c + d;
        return result;
    }
}
```



```
function sum(a, b) {
    var x = undefined;
    function add(c, d) {
        var result = c + d;
        return result;
    }

    x = add(a, b);
    return x;
}
```

let

- Let의 호이스팅
  - Let은 Var와는 달리 호이스팅은 되지만 선언된 시점 전에 접근이 불가능하다.
  - 이것을 TDZ(Temporal Dead Zone)이라고 하며 let을 통해 만든 변수의 선언부 전까지 엑세스가 불허된다.
  - 따라서 Let은 호이스팅은 되지만 TDZ의 성질로 인해 변수의 선언부에 도달할 경우 에러가 발생한다.

```
console.log(sports);
var sports = "스포츠";

console.log(music);
let music = "음악";
```

```
undefined
Uncaught ReferenceError: music is not defined
    at let-hoisting.js:7
```

let

- for()
  - For문에서 let 으로 초기값을 선언할 시 반복할 때마다 스코프를 갖는다
  - 하지만 var로 초기값을 선언할 시 반복될 때마다 스코프를 갖지 않는다.

```html
<!DOCTYPE html>
<html lang=ko>
<head>
    <meta charset="utf-8">
    <title>ES6+ES7</title>
    <script src="let-for-var.js" defer></script>
</head>
<body>
<div style="position: absolute; top: 30px; left: 70px;">
선택, 클릭
<ul style="margin-top: 0px; padding-left: 20px;">
    <li>1~10</li>
    <li>11~20</li>
    <li>21~30</li>
</ul>
</div>
</body>
</html>
```

```js
var nodes = document.querySelector("ul");
for (var k = 0; k < nodes.children.length; k++) {
    var el = nodes.children[k];
    el.onclick = function(event) {
        event.target.style.backgroundColor = "yellow";
        console.log(k);
    }
};
```

선택, 클릭

- 1~10
- 11~20
- 21~30

3 3

let

- for()

```
var nodes = document.querySelector("ul");
for (let k = 0; k < nodes.children.length; k++) {
    var el = nodes.children[k];
    el.onclick = function(event) {
        event.target.style.backgroundColor = "yellow";
        console.log(k);
    }
}
```

선택, 클릭

- 1~10
- 11~20
- 21~30

0
1
2

const

- Const
  - Const 키워드는 변수를 상수화 시킬 경우 사용한다.

```
const name1 = value1 [, name2 = value2 [, nameN =valueN]]
```

  - Const 키워드는 다음과 같은 특징을 가진다
    - Let과는 달리 const는 무조건 초기값을 할당해야만 한다.
    - Const가 초기화 되면 외부에서 값을 재할당할 경우 에러를 발생시킨다.
    - 그 이외에 다른 성질은 let과 똑같으며 상수를 선언 시 모든 알파벳은 대문자로 쓴다.
    - Object 형태로 Const를 선언 시 Object 자체를 할당할 수는 없지만 Object의 프로퍼티에는 값을 할당할 수 있다.

const

- const

```
const SPORTS = "축구";
try {
  SPORTS = "농구";
} catch (e) {
  console.log("const 재할당 불가");
}
```

```
const 재할당 불가
```

```
const obj = {language: "한글"};
try {
  obj = {};
} catch (e) {
  console.log("const 재할당 불가");
}
obj.language = "영어";
console.log(obj.language);
```

```
const 재할당 불가
영어
```