let, const와 블록 레벨 스코프

ES5까지 변수를 선언할 수 있는 유일한 방법은 var 키워드를 사용하는 것이었다. var 키워드로 선언된 변수는 아래와 같은 특징이 있다. 이는 다른 언어와는 다른 특징으로 주의를 기울이지 않으면 심각한 문제를 일으킨다.

1. 함수 레벨 스코프(Function-level scope)

- 함수의 코드 블록만을 스코프로 인정한다. 따라서 전역 함수 외부에서 생성한 변수는 모두 전역 변수이다. 이는 전역 변수를 남발할 가능성을 높인다.

- for 문의 변수 선언문에서 선언한 변수를 for 문의 코드 블록 외부에서 참조할 수 있다.

2. var 키워드 생략 허용

- 암묵적 전역 변수를 양산할 가능성이 크다.

3. 변수 중복 선언 허용

- 의도하지 않은 변수값의 변경이 일어날 가능성이 크다.

4. 변수 호이스팅

- 변수를 선언하기 이전에 참조할 수 있다.

대부분의 문제는 전역 변수로 인해 발생한다. 전역 변수는 간단한 애플리케이션의 경우, 사용이 편리하다는 장점이 있지만 불가피한 상황을 제외하고 사용을 억제해야 한다. 전역 변수는 유효 범위(scope)가 넓어서 어디에서 어떻게 사용될 것인지 파악하기 힘들며, 비순수 함수(Impure function)에 의해 의도하지 않게 변경될 수도 있어서 복잡성을 증가시키는 원인이 된다. 따라서 변수의 스코프는 좁을수록 좋다.

ES6는 이러한 var 키워드의 단점을 보완하기 위해 let과 const 키워드를 도입하였다.

let

블록 레벨 스코프

대부분의 프로그래밍 언어는 블록 레벨 스코프(Block-level scope)를 따르지만 자바스크립트는 함수 레벨 스코프(Function-level scope)를 따른다.

함수 레벨 스코프(Function-level scope)

함수 내에서 선언된 변수는 함수 내에서만 유효하며 함수 외부에서는 참조할 수 없다. 즉, 함수 내부에서 선언한 변수는 지역 변수이며 함수 외부에서 선언한 변수는 모두 전역 변수이다.

블록 레벨 스코프(Block-level scope)

모든 코드 블록(함수, if 문, for 문, while 문, try/catch 문 등) 내에서 선언된 변수는 코드 블록 내에서만 유효하며 코드 블록 외부에서는 참조할 수 없다. 즉, 코드 블록 내부에서 선언한 변수는 지역 변수이다.

아래 예제를 살펴보자.

```
var foo = 123; // 전역 변수

console.log(foo); // 123

{
  var foo = 456; // 전역 변수
}

console.log(foo); // 456
```

블록 레벨 스코프를 따르지 않는 var 키워드의 특성 상, 코드 블록 내의 변수 foo는 전역 변수이다. 그런데 이미 전역 변수 foo가 선언되어 있다. var 키워드를 사용하여 선언한 변수는 중복 선언이 허용되므로 위의 코드는 문법적으로 아무런 문제가 없다. 단, 코드 블록 내의 변수 foo는 전역 변수이기 때문에 전역에서 선언된 전역 변수 foo의 값 123을 새로운 값 456으로 재할당하여 덮어쓴다.

ES6는 **블록 레벨 스코프**를 따르는 변수를 선언하기 위해 `let` 키워드를 제공한다.

```
let foo = 123; // 전역 변수

{
  let foo = 456; // 지역 변수
  let bar = 456; // 지역 변수
}

console.log(foo); // 123
console.log(bar); // ReferenceError: bar is not defined
```

let 키워드로 선언된 변수는 블록 레벨 스코프를 따른다. 위 예제에서 코드 블록 내에 선언된 변수 foo는 블록 레벨 스코프를 갖는 지역 변수이다. 전역에서 선언된 변수 foo와는 다른 별개의 변수이다. 또한 변수 bar도 블록 레벨 스코프를 갖는 지역 변수이다. 따라서 전역에서는 변수 bar를 참조할 수 없다.

변수 중복 선언 금지

var 키워드로는 동일한 이름을 갖는 변수를 중복해서 선언할 수 있었다. 하지만, let 키워드로는 동일한 이름을 갖는 변수를 중복해서 선언할 수 없다. 변수를 중복 선언하면 문법 에러(SyntaxError)가 발생한다.

```
var foo = 123;
var foo = 456;  // 중복 선언 허용

let bar = 123;
let bar = 456;  // Uncaught SyntaxError: Identifier 'bar' h
as already been declared
```

호이스팅

자바스크립트는 ES6에서 도입된 let, const를 포함하여 모든 선언(var, let, const, function, function*, class)을 호이스팅한다. 호이스팅(Hoisting)이란, var 선언문이나 function 선언문 등을 해당 스코프의 선두로 옮긴 것처럼 동작하는 특성을 말한다.

하지만 var 키워드로 선언된 변수와는 달리 let 키워드로 선언된 변수를 선언문 이전에 참조하면 참조 에러(ReferenceError)가 발생한다. 이는 let 키워드로 선언된 변수는 스코프의 시작에서 변수의 선언까지 **일시적 사각지대(Temporal Dead Zone; TDZ)**에 빠지기 때문이다.

```
console.log(foo); // undefined
var foo;

console.log(bar); // Error: Uncaught ReferenceError: bar is
not defined
let bar;
```

변수가 어떻게 생성되며 호이스팅은 어떻게 이루어지는지 좀 더 자세히 살펴보자. 변수는 3단계에 걸쳐 생성된다. 자세한 내용은 Execution Context을 참조하기 바란다.

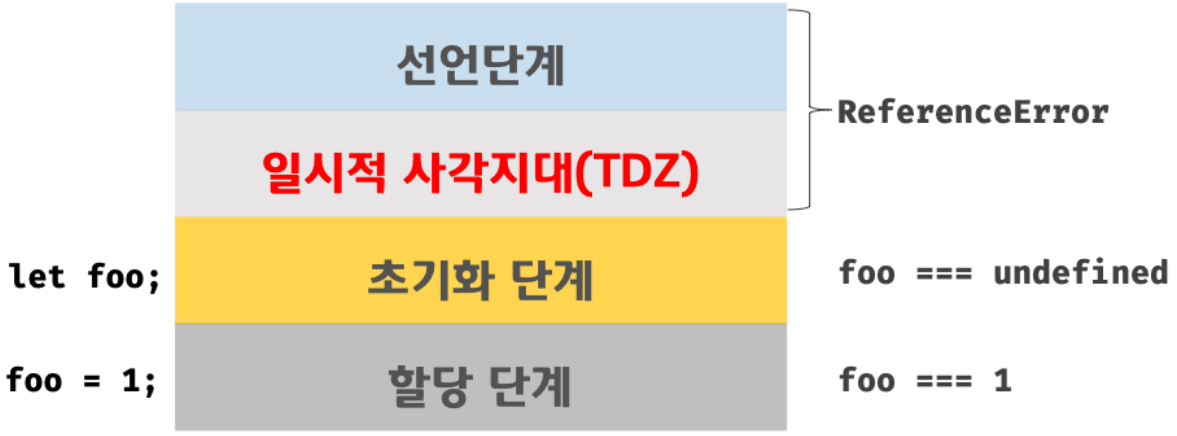
선언 단계(Declaration phase)변수를 실행 컨텍스트의 변수 객체(Variable Object)에 등록한다. 이 변수 객체는 스코프가 참조하는 대상이 된다.**초기화 단계(Initialization phase)** 변수 객체(Variable Object)에 등록된 변수를 위한 공간을 메모리에 확보한다. 이 단계에서 변수는 undefined로 초기화된다.**할당 단계(Assignment phase)**undefined로 초기화된 변수에 실제 값을 할당한다.

var 키워드로 선언된 변수는 선언 단계와 초기화 단계가 한번에 이루어진다. 즉, 스코프에 변수를 등록(선언 단계)하고 메모리에 변수를 위한 공간을 확보한 후, undefined로 초기화(초기화 단계)한다. 따라서 변수 선언문 이전에 변수에 접근하여도 스코프에 변수가 존재하기 때문에 에러가 발생하지 않는다. 다만 undefined를 반환한다. 이후 변수 할당문에 도달하면 비로소 값이 할당된다. 이러한 현상을 변수 호이스팅(Variable Hoisting)이라 한다.

```
// 스코프의 선두에서 선언 단계와 초기화 단계가 실행된다.
// 따라서 변수 선언문 이전에 변수를 참조할 수 있다.
console.log(foo); // undefined

var foo;
console.log(foo); // undefined

foo = 1; // 할당문에서 할당 단계가 실행된다.
console.log(foo); // 1
```

let 키워드로 선언된 변수는 선언 단계와 초기화 단계가 분리되어 진행된다. 즉, 스코프에 변수를 등록(선언단계)하지만 초기화 단계는 변수 선언문에 도달했을 때 이루어진다. 초기화 이전에 변수에 접근하려고 하면 참조 에러(ReferenceError)가 발생한다. 이는 변수가 아직 초기화되지 않았기 때문이다. 다시 말하면 변수를 위한 메모리 공간이 아직 확보되지 않았기 때문이다. 따라서 스코프의 시작 지점부터 초기화 시작 지점까지는 변수를 참조할 수 없다. 스코프의 시작 지점부터 초기화 시작 지점까지의 구간을 '일시적 사각지대(Temporal Dead Zone; TDZ)'라고 부른다.

```
// 스코프의 선두에서 선언 단계가 실행된다.
// 아직 변수가 초기화(메모리 공간 확보와 undefined로 초기화)되지 않았다.
// 따라서 변수 선언문 이전에 변수를 참조할 수 없다.
console.log(foo); // ReferenceError: foo is not defined

let foo; // 변수 선언문에서 초기화 단계가 실행된다.
console.log(foo); // undefined

foo = 1; // 할당문에서 할당 단계가 실행된다.
console.log(foo); // 1
```



결국 ES6에서는 호이스팅이 발생하지 않는 것과 차이가 없어 보인다. 하지만 그렇지 않다. 아래 예제를 살펴보자.

```
let foo = 1; // 전역 변수

{
  console.log(foo); // ReferenceError: foo is not defined
  let foo = 2; // 지역 변수
}
```

위 예제의 경우, 전역 변수 foo의 값이 출력될 것처럼 보인다. 하지만 ES6의 선언문도 여전히 호이스팅이 발생하기 때문에 참조 에러(ReferenceError)가 발생한다.

ES6의 let으로 선언된 변수는 블록 레벨 스코프를 가지므로 코드 블록 내에서 선언된 변수 foo는 지역 변수이다. 따라서 지역 변수 foo도 해당 스코프에서 호이스팅되고 코드 블록의 선두부터 초기화가 이루어지는 지점까지 일시적 사각지대(TDZ)에 빠진다. 따라서 전역 변수 foo의 값이 출력되지 않고 참조 에러(ReferenceError)가 발생한다.

클로저

블록 레벨 스코프를 지원하는 let은 var보다 직관적이다. 다음 코드를 살펴보자.

```
var funcs = [];

// 함수의 배열을 생성하는 for 루프의 i는 전역 변수다.
for (var i = 0; i < 3; i++) {
```

```
  funcs.push(function () { console.log(i); });
}

// 배열에서 함수를 꺼내어 호출한다.
for (var j = 0; j < 3; j++) {
  funcs[j]();
}
```

위 코드의 실행 결과로 0, 1, 2를 기대할 수도 있지만 결과는 3이 세 번 출력된다. 그 이유는 for 루프의 var i가 전역 변수이기 때문이다. 0, 1, 2를 출력하려면 아래와 같은 코드가 필요하다.

```
var funcs = [];

// 함수의 배열을 생성하는 for 루프의 i는 전역 변수다.
for (var i = 0; i < 3; i++) {
  (function (index) { // index는 자유변수다.
    funcs.push(function () { console.log(index); });
  }(i));
}

// 배열에서 함수를 꺼내어 호출한다
for (var j = 0; j < 3; j++) {
  funcs[j]();
}
```

자바스크립트의 함수 레벨 스코프로 인하여 for 루프의 초기화 식에 사용된 변수가 전역 스코프를 갖게 되어 발생하는 문제를 회피하기 위해 클로저를 활용한 방법이다.

ES6의 let 키워드를 for 루프의 초기화 식에 사용하면 클로저를 사용하지 않아도 위 코드와 동일한 동작을 한다.

```
var funcs = [];

// 함수의 배열을 생성하는 for 루프의 i는 for 루프의 코드 블록에서만 유효한 지역 변수이면서 자유 변수이다.
for (let i = 0; i < 3; i++) {
  funcs.push(function () { console.log(i); });
```

```
}

// 배열에서 함수를 꺼내어 호출한다
for (var j = 0; j < 3; j++) {
  console.dir(funcs[j]);
  funcs[j]();
}
```

for 루프의 let i는 for loop에서만 유효한 지역 변수이다. 또한, i는 자유 변수로서 for 루프의 생명주기가 종료되어도 변수 i를 참조하는 함수가 존재하는 한 계속 유지된다.

```
> var funcs = [];

  // 함수의 배열을 생성하는 for 루프의 i는 for 루프의 코드 블록에서만 유효한 지역 변수이면서 자유 변수이다.
  for (let i = 0; i < 3; i++) {
    funcs.push(function () { console.log(i); });
  }

  // 배열에서 함수를 꺼내어 호출한다
  for (var j = 0; j < 3; j++) {
    console.dir(funcs[j])
    funcs[j]();
  }
  ▼ ƒ anonymous()
      arguments: null
      caller: null
      length: 0
      name: ""
    ▶ prototype: {constructor: ƒ}
    ▶ __proto__: ƒ ()
      [[FunctionLocation]]: VM56:5
    ▼ [[Scopes]]: Scopes[2]
      ▼ 0: Block
          i: 0
      ▶ 1: Global {type: "global", name: "", object: Window}
  0
```

전역 객체와 let

전역 객체(Global Object)는 모든 객체의 유일한 최상위 객체를 의미하며 일반적으로 Browser-side에서는 window 객체, Server-side(Node.js)에서는 global 객체를 의미한다. var 키워드로 선언된 변수를 전역 변수로 사용하면 전역 객체의 프로퍼티가 된다.

```
var foo = 123; // 전역변수

console.log(window.foo); // 123
```

let 키워드로 선언된 변수를 전역 변수로 사용하는 경우, let 전역 변수는 전역 객체의 프로퍼티가 아니다. 즉, window.foo와 같이 접근할 수 없다. let 전역 변수는 보이지 않는 개념적인 블록 내에 존재하게 된다.

```
let foo = 123; // 전역변수

console.log(window.foo); // undefined
```

const

const는 상수(변하지 않는 값)를 위해 사용한다. 하지만 반드시 상수만을 위해 사용하지는 않는다. 이에 대해서는 후반부에 설명한다. const의 특징은 let과 대부분 동일하므로 let과 다른 점만 살펴보도록 하자.

선언과 초기화

let은 재할당이 자유로우나 const는 재할당이 금지된다.

```
const FOO = 123;
FOO = 456; // TypeError: Assignment to constant variable.
```

주의할 점은 **const는 반드시 선언과 동시에 할당이 이루어져야 한다**는 것이다. 그렇지 않으면 다음처럼 문법 에러(SyntaxError)가 발생한다.

```
const FOO; // SyntaxError: Missing initializer in const dec
laration
```

또한, const는 let과 마찬가지로 블록 레벨 스코프를 갖는다.

```
{
  const FOO = 10;
  console.log(FOO); //10
}
console.log(FOO); // ReferenceError: FOO is not defined
```

상수

상수는 가독성과 유지보수의 편의를 위해 적극적으로 사용해야 한다. 예를 들어 아래 코드를 살펴보자.

```
// 10의 의미를 알기 어렵기 때문에 가독성이 좋지 않다.
if (rows > 10) {
}

// 값의 의미를 명확히 기술하여 가독성이 향상되었다.
const MAXROWS = 10;
if (rows > MAXROWS) {
}
```

조건문 내의 10은 어떤 의미로 사용하였는지 파악하기가 곤란하다. 하지만 네이밍이 적절한 상수로 선언하면 가독성과 유지보수성이 대폭 향상된다.

const는 객체에도 사용할 수 있다. 물론 이때도 재할당은 금지된다.

```
const obj = { foo: 123 };
obj = { bar: 456 }; // TypeError: Assignment to constant va
riable.
```

const와 객체

const는 재할당이 금지된다. 이는 const 변수의 타입이 객체인 경우, 객체에 대한 참조를 변경하지 못한다는 것을 의미한다. 하지만 이때 **객체의 프로퍼티는 보호되지 않는다.** 다시 말하자면 재할당은 불가능하지만 할당된 객체의 내용(프로퍼티의 추가, 삭제, 프로퍼티 값의 변경)은 변경할 수 있다.

```
const user = { name: 'Lee' };

// const 변수는 재할당이 금지된다.
// user = {}; // TypeError: Assignment to constant variabl
e.

// 객체의 내용은 변경할 수 있다.
user.name = 'Kim';

console.log(user); // { name: 'Kim' }
```

객체의 내용이 변경되더라도 객체 타입 변수에 할당된 주소값은 변경되지 않는다. 따라서 **객체 타입 변수 선언에는 const를 사용하는 것이 좋다.** 만약에 명시적으로 객체 타입 변수의 주소값을 변경(재할당)하여야 한다면 let을 사용한다.

var vs. let vs. const

변수 선언에는 기본적으로 const를 사용하고 let은 재할당이 필요한 경우에 한정해 사용하는 것이 좋다. 원시 값의 경우, 가급적 상수를 사용하는 편이 좋다. 그리고 객체를 재할당하는 경우는 생각보다 흔하지 않다. const 키워드를 사용하면 의도치 않은 재할당을 방지해 주기 때문에 보다 안전하다.

var와 let, 그리고 const는 다음처럼 사용하는 것을 추천한다.

- ES6를 사용한다면 var 키워드는 사용하지 않는다.

- 재할당이 필요한 경우에 한정해 let 키워드를 사용한다. 이때 변수의 스코프는 최대한 좁게 만든다.

- 변경이 발생하지 않는(재할당이 필요 없는 상수) 원시 값과 객체에는 const 키워드를 사용한다. const 키워드는 재할당을 금지하므로 var, let 보다 안전하다.

변수를 선언하는 시점에는 재할당이 필요할지 잘 모르는 경우가 많다. 그리고 객체는 의외로 재할당을 하는 경우가 드물다. 따라서 변수를 선언할 때에는 일단 const 키워드를 사용하도록 하자. 반드시 재할당이 필요하다면(반드시 재할당이 필요한지 한번 생각해 볼 일이다.) 그때 const를 let 키워드로 변경해도 결코 늦지 않는다.