
모던 자바스크립트

Deep Dive

14장_전역 변수의 문제점

전역 변수의 문제점과 전역 변수의 사용을 억제할 수 있는 방법에 대해서 알아보자

지역 변수의 생명 주기

변수는 선언에 의해 생성되고(메모리 할당)
할당을 통해 값을 갖고,
언젠가 소멸함(메모리 해제, 스코프가 소멸, 렉시컬 환경이 소멸)
=== **변수의 생명 주기(Life Cycle)**

지역 변수의 생명 주기는 함수의 생명 주기(호출 ~ 반환문 혹은 함수의 마지막 문)와 일치

```
function foo() {  
    var x = 'local';  
    console.log(x);  
    return x;  
}
```

변수 x 생성
변수 x 값 할당

변수 x 소멸

변수 x 생명 주기

지역 변수의 생명 주기

변수는 선언에 의해 생성되고(메모리 할당)
할당을 통해 값을 갖고,
언젠가 소멸함(메모리 해제, 스코프가 소멸, 렉시컬 환경이 소멸)
=== **변수의 생명 주기(Life Cycle)**

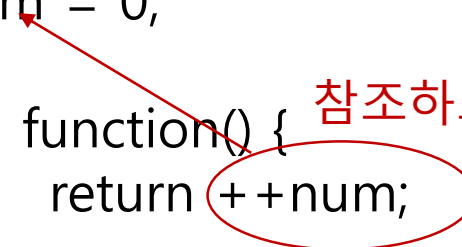
지역 변수의 생명 주기는 함수의 생명 주기와 **일치하지 않는 경우가 존재 '클로저'**

할당된 메모리 공간은 더 이상 그 누구도 참조하지 않을 때,
가비지 콜렉터에 의해 해제

num은 계속해서 참조되고 있음.

```
const increase = (function() {  
    let num = 0;  
  
    return function() {  
        return ++num;  
    };  
})();
```

참조하고 있음



```
console.log(increase()); // 1  
console.log(increase()); // 2  
console.log(increase()); // 3
```

지역 변수의 호이스팅

호이스팅은 스코프를 단위로 동작

전역 변수의 호이스팅은 전역 변수의 선언이 전역 스코프의 선두로 끌어 올려진 것처럼 동작

지역 변수의 호이스팅은 지역 변수의 선언이 지역 스코프의 선두로 끌어 올려진 것처럼 동작

즉, 호이스팅은 변수 선언이 **스코프의 선두**로 끌어 올려진 것처럼 동작하는 자바스크립트 고유의 특징

```
var x = 'global'
```

```
function foo() {  
    console.log(x); // undefined  
    var x = 'local'  
}  
foo();  
console.log(x); // global
```

전역 변수의 생명 주기

- 명시적인 호출 없이 실행
- 더 이상 실행할 문이 없을 때 종료
- **var 키워드로 선언한 전역 변수 및 함수 선언문은 전역 객체의 프로퍼티에 등록**
즉, 전역 변수의 생명 주기는 전역 객체의 생명 주기와 동일

※ 전역 객체

- 런타임 이전에, 자바스크립트 엔진에 의해서 어떤 객체보다도 먼저 생성되는 특수한 객체
- 환경에 따라 지칭하는 이름이 제각각
브라우저 환경에서는 window(또는 self, this, frames), Node.js 환경에서는 global
ES11에 도입된 globalThis는 환경에 관계없이 사용 가능
- **표준 빌트인 객체**(Object, String, Number...), 환경에 따른 **호스트 객체**(Web API 또는 Node.js API),
var 키워드로 선언한 전역 변수와 전역 함수, 암묵적 전역 변수(선언하지 않은 변수에 값을 할당)
를 프로퍼티로 가짐

전역 변수의 문제점

1. 암묵적 결합

- 모든 코드가 전역 변수를 참조하고 변경할 수 있는 암묵적 결합 허용
- 이는 코드의 가독성은 나빠지고, 의도치 않게 상태가 변경될 수 있는 위험도 증가

2. 긴 생명 주기

- 메모리 리소스 오랜 기간 소비
- 상태를 변경할 수 있는 시간도 길고, 기회도 많음
- var 키워드는 변수의 **중복 선언이 가능**하므로, 중복될 가능성 존재
당연하게도 let과 const는 발생하지 않음

3. 스코프 체인 상에서 종점에 존재

- 전역 변수의 검색 속도가 가장 느림

4. 네임스페이스 오염

- 자바스크립트의 가장 큰 문제점 중 하나는 파일이 분리되어 있다 해도 하나의 전역 스코프를 공유하는 것
- 변수 명이 겹칠 수 있음

전역 변수 억제 방법

1. 지역 변수 사용

- 전역 변수를 반드시 사용해야 할 이유를 찾지 못한다면 지역 변수를 사용해야 함
- 변수의 스코프 범위는 좁을수록 좋음

2. 즉시 실행 함수

- 모든 코드를 즉시 실행 함수로 감싸면, 모든 변수는 즉시 실행 함수의 지역 변수가 됨

```
(function () {  
    var foo = 10;  
})();
```

3. 네임스페이스 객체

- 전역에 네임스페이스 역할을 담당하는 객체를 생성하고 전역 변수를 프로퍼티에 추가

```
var MYAPP = {}; // 전역 네임스페이스 객체  
MYAPP.name = 'Lee';  
console.log(MYAPP.name); // Lee
```

전역 변수 억제 방법

3. 네임스페이스 객체

- 계층적으로 구성 가능

```
var MYAPP = {}; // 전역 네임스페이스 객체
MYAPP.person = {
  name: 'Lee',
  address: 'Seoul'
}
console.log(MYAPP.person.name); // Lee
```

네임스페이스를 분리해서 식별자 충돌을 방지하는 효과는 있으나, 네임스페이스 객체 자체가 전역 변수에 할당되므로, **그다지 유용해 보이진 않음**

전역 변수 억제 방법

4. 모듈 패턴

- 클래스 모방
- 클로저 기반
- 캡슐화까지 구현

※ 캡슐화

객체의 상태(프로퍼티)와 동작(메서드, 프로퍼티 참조 및 조작)을 하나로 묶는 것
프로퍼티나 메서드를 감추는 **정보 은닉(information hiding)** 용도로도 사용됨

```
const increase = (function() {  
  let num = 0;  
  return function() {  
    return ++num;  
  };  
})();
```

private 멤버

참조하고 있음

public 멤버

```
console.log(increase()); // 1  
console.log(increase()); // 2  
console.log(increase()); // 3
```

전역 변수 억제 방법

5. ES6 모듈

- 파일 자체의 **독자적인 모듈 스코프** 제공
- 전역 변수 사용 불가
 - var 키워드로 선언한 변수는 더는 전역 변수가 아니며, window 객체의 프로퍼티가 아님
- 모던 브라우저(Chrome 61, FF 60, SF 10.1, Edge 16이상)에서 사용 가능
- 모듈 파일 **확장자는 mjs** 권장

```
<script type="module" src="lib.mjs"> </script>
```

```
<script type="module" src="app.mjs"> </script>
```

브라우저의 ES6 모듈 기능을 사용하더라도 트랜스파일링이나 번들링이 필요하기 때문에 아직까지는 브라우저가 지원하는 ES6 모듈 기능보다는 Webpack 등의 모듈 번들러를 사용하는 것이 일반적

15장_let, const 키워드와 블록 레벨 스코프

var 키워드 문제점

1. 중복 선언 허용

```
var x = 1;  
var y = 1;  
var x = 100; // 초기화문이 있는 변수 선언문은 자바스크립트 엔진에 의해 var 키워드는 없는 것 처럼 동작  
var y; // 초기화문이 없는 변수 선언문은 무시됨
```

```
console.log(x); // 100  
console.log(y); // 1
```

var 키워드 문제점

2. 함수 레벨 스코프

- 함수의 코드 블록만을 지역 스코프로 인정

(1) 예시

```
var x = 1;
```

```
if(true) {  
    var x = 10; // x는 중복 선언됨  
}
```

```
console.log(x); // 10
```

(2) 예시

```
var i = 10;
```

```
// i는 중복 선언됨  
for(var i = 0; i < 5; i++) {  
    console.log(i);  
}
```

```
console.log(i); //5
```

ES6 let 키워드 등장

1. 변수 중복 선언 금지

```
var foo = 123;  
var foo = 456;
```

```
let bar = 123;  
let bar = 456; // SyntaxError : Identifier 'bar' has already been declared
```

2. 블록 레벨 스코프

- 모든 코드 블록(함수, if문, for문, while문, try/catch)을 지역 스코프로 인정

```
let i = 10; 전역 스코프
```

```
function foo() {  
    let i = 100; 함수 레벨 스코프  
  
    for(let i = 1; i < 3; i++) {  
        console.log(i); // 1, 2 블록 레벨 스코프  
    }  
    console.log(i); // 100  
}
```

```
foo();  
console.log(i); // 10
```


ES6 let 키워드 등장

3. 전역 객체와 let

- var 키워드로 선언한 전역 변수와 함수 선언문의 전역 함수, 암묵적 전역 변수(선언하지 않은 변수에 값을 할당)
- 전역 객체의 프로퍼티 참조시 window 생략 가능

```
var x = 1;  
y = 2; // 암묵적 전역
```

```
function foo() {}
```

(1) var 키워드 사용

```
console.log(window.x); // 1
```

```
console.log(x); //1
```

(2) 암묵적 전역

```
console.log(window.y); //2
```

```
console.log(y); //2
```

(3) 함수선언문

```
console.log(window.foo); // f foo() {}
```

```
console.log(foo); // f foo() {}
```

ES6 const 키워드 등장

1. 선언과 초기화

- 반드시 선언과 동시에 초기화 필요

```
const foo; // SyntaxError: Missing initializer in const declaration
```

2. 재할당 금지

```
const foo = 1;
```

```
foo = 2; // TypeError : Assignment to constant variable
```

3. 상수

- 원시 값을 할당한 경우 변수 값을 변경할 수 없음
- 이러한 특징을 이용해 **상수를 표현**하는 데 사용. 상수는 상태 유지, 가독성, 유지보수 편의에 좋음
- 상수는 재할당이 금지된 **변수**

ES6 const 키워드 등장

4. const 키워드와 객체

- const 키워드로 선언된 변수에 원시 값을 할당한 경우 값을 변경할 수 없음
- const 키워드로 선언된 변수에 객체를 할당한 경우 값을 변경할 수 있음
객체는 재할당 없이도 프로퍼티 동적 생성, 삭제, 프로퍼티 값 변경이 가능하므로,
단, const 키워드로 선언된 변수에 할당된 참조 값은 변경할 수 없음

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

```
person.name = 'Kim';
```

```
console.log(person.name); // Kim
```

var vs let vs const

- (1) 기본적으로 const 사용
- (2) 재할당이 필요한 경우 let 사용
- (3) let을 사용하는 경우, 변수 스코프를 최대한 좁게 만듦
- (4) ES6를 사용한다면 var 키워드는 사용하지 않는다

변수를 선언하는 시점에는 재할당이 필요할지 모르는 경우가 많으므로 const로 선언. 재할당이 필요할 때 let 키워드로 변경한다.

객체는 재할당되는 경우가 드물기 때문에 const로 선언.