

14. 전역 변수의 문제점

TABLE OF CONTENTS

1. 변수의 생명 주기

1.1. 지역 변수의 생명 주기

1.2. 전역 변수의 생명 주기

2. 전역 변수의 문제점

!

3. 전역 변수 사용 억제 방법

3.1. 즉시 실행 함수

3.2. 네임 스페이스 객체

3.3. 모듈 패턴

3.4. ES6 모듈

전역 변수의 무분별한 사용은 위험하다. 전역 변수를 반드시 사용하여야 할 이유를 찾지 못한다면 지역 변수를 사용하여야 한다. 전역 변수의 문제점과 전역 변수의 사용을 억제할 수 있는 방법에 대해 살펴해보도록 하자.

1. 변수의 생명 주기

1.1. 지역 변수의 생명 주기

변수는 선언에 의해 생성되고 할당을 통해 값을 갖는다. 그리고 언젠가 소멸한다. 즉, 변수는 생물과 유사하게 생성되고 소멸되는 생명 주기(Life cycle)가 있다. 변수에 생명 주기가 없다면 한번 선언된 변수는 프로그램을 종료하지 않는 한 영원히 메모리 공간을 점유하게 된다.

변수는 자신이 선언된 위치에서 생성되고 소멸한다. 전역 변수의 생명 주기는 애플리케이션의 생명 주기와 같다. 하지만 함수 내부에서 선언된 지역 변수는 함수가 호출되면 생성되고 함수가 종료하면 소멸한다.

아래 예제를 살펴보자.

```
var
var x = 1; -> window.x      1
                               (let, const)
                               , window
                               가
```

JAVASCRIPT

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

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

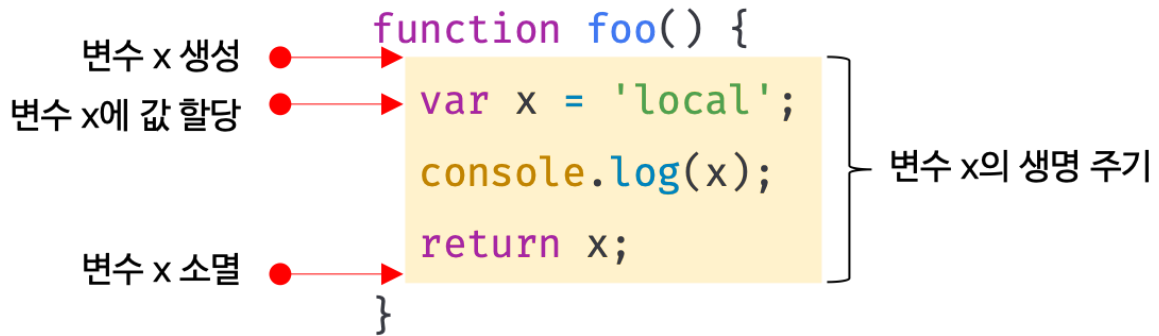
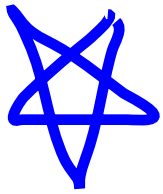
지역 변수 x는 foo 함수가 호출되기 이전까지는 생성되지 않는다. foo 함수를 호출하지 않으면 함수 내부의 변수 선언문이 실행되지 않기 때문이다.

“4.4. 변수 선언의 실행 시점과 변수 호이스팅”에서 살펴보았듯이 변수 선언은 다른 코드가 실행되기 이전에 변수 선언이 어디에 있던지 상관없이 가장 먼저 실행된다. 다시 말해, 변수 선언은 코드가 한 줄씩 순차적으로 실행되는 시점인 런타임(runtime)에 실행되는 것이 아니라 런타임 이전 단계에서 자바스크립트 엔진에 의해 먼저 실행된다.

그런데 엄밀히 말하자면 위 설명은 전역 변수에 한정된 것이다. 함수 내부에서 선언한 변수는 함수가 호출된 직후에 함수 몸체의 다른 코드가 실행되기 이전에 자바스크립트 엔진에 의해 먼저 실행된다.

위 예제의 foo 함수를 호출하면 함수 몸체의 다른 문들이 순차적으로 실행되기 이전에 변수 x의 선언문이 자바스크립트 엔진에 의해 가장 먼저 실행되어 변수 x가 선언되고 undefined로 초기화된다. 그 후, 함수 몸체의 문들이 순차적으로 실행되기 시작하고 변수 할당문이 실행되면 변수 x에 값이 할당된다. 그리고 함수가 종료하면 변수 x도 소멸되어 생명 주기가 종료된다. 따라서 함수 내부에서 선언된 지역 변수 x는 foo 함수가 호출되어 실행되는 동안에만 유효하다. 즉, 지역 변수의 생명 주기는 함수의 생명 주기와 일치한다.

```
? :
-> 가
```



```
foo();
console.log(x);
```

지역 변수의 생명 주기

함수 몸체 내부에서 선언된 지역 변수의 생명 주기는 대부분 함수의 생명 주기와 일치하지만 지역 변수가 함수보다 오래 생존하는 경우도 있다.

변수는 하나의 값을 저장하기 위해 확보한 메모리 공간 자체 또는 그 메모리 공간을 식별하기 위해 붙인 이름이다. 따라서 **변수의 생명 주기는 메모리 공간이 확보(allocate)된 시점부터 메모리 공간이 해제(release)되어 가용 메모리 풀(memory pool)에 반환되는 시점까지이다.**

함수 내부에서 선언된 지역 변수는 함수가 생성한 스코프에 등록된다. 함수가 생성한 스코프는 렉시컬 환경이라 부르는 물리적인 실체가 있다고 했다. (“13.3. 스코프 체인” 참고) 따라서 변수는 자신이 등록된 스코프가 소멸(스코프가 메모리에서 해제)될 때까지 유효하다. 할당(allocate)된 메모리 공간은 더 이상 그 누구도 참조하지 않을 때 가비지 컬렉터에 의해 해제(release)되어 가용 메모리 풀에 반환된다. 즉, 누군가가 메모리 공간을 참조하고 있으면 해제되지 않고 확보된 상태로 남아 있게 된다. 이는 스코프도 마찬가지다. 누군가 스코프를 참조하고 있으면 스코프는 해제되지 않고 생존하게 된다.

일반적으로 함수가 종료하면 함수가 생성한 스코프도 소멸한다. 하지만 누군가가 스코프를 참조하고 있다면 스코프는 해제되지 않고 생존하게 된다. 이에 대해서는 “24. 클로저”에서 자세히 살펴보도록 하자.

위 예제를 조금 변형한 퀴즈를 풀어보자. 아래 예제의 ①에서 출력되는 값은 무엇인가?

JAVASCRIPT

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

```
function foo() {
  console.log(x); // ①
  var x = 'local';
  return x;
}
```

.

x global .

```
foo();
console.log(x); // global
```

함수 foo 내부에서 선언된 지역 변수 x는 ①의 시점에 이미 선언되고 undefined로 초기화 되었다. 따라서 전역 변수 x를 참조하는 것이 아니라 지역 변수 x를 참조하여 값을 출력한다. 즉, 지역 변수는 함수 전체에서 유효하다. 단, 변수 할당문이 실행되기 이전까지는 undefined 값을 갖는다.

이처럼 **호이스팅은 스코프를 단위로 동작한다.** 전역 변수의 호이스팅은 전역 변수의 선언이 전역 스코프의 선두로 끌어 올려진 것처럼 동작한다. 따라서 전역 변수는 전역 전체에서 유효하다. 지역 변수의 호이스팅은 지역 변수의 선언이 지역 스코프의 선두로 끌어 올려진 것처럼 동작한다. 따라서 지역 변수는 함수 전체에서 유효하다. 즉, 호이스팅은 변수 선언이 스코프의 선두로 끌어 올려진 것처럼 동작하는 자바스크립트 고유의 특징을 말한다.

1.2. 전역 변수의 생명 주기

함수와는 달리 전역 코드는 명시적인 호출없이 실행된다. 다시 말해 전역 코드는 함수 호출과 같이 전역 코드를 실행하는 특별한 진입점(entry point)이 없고 코드가 로드되자마자 곧바로 해석되고 실행된다.

진입점(entry point)

C나 Java으로 작성된 코드를 실행하면 가장 먼저 main 함수가 호출된다. 이 main 함수는 프로그램이 시작되는 지점이므로 이를 진입점 또는 시작점이라고 한다.

함수는 함수 몸체의 마지막 문 또는 반환문이 실행되면 종료한다. 하지만 전역 코드에는 반환문을 사용할 수 없으므로 마지막 문이 실행되어 더 이상 실행할 문이 없을 때 종료한다. (“12.5.4. 반환문” 참고)

var 키워드로 선언한 전역 변수는 전역 객체의 프로퍼티가 된다. 이는 전역 변수의 생명 주기가 전역 객체의 생명 주기와 일치한다는 것을 말한다.

전역 객체

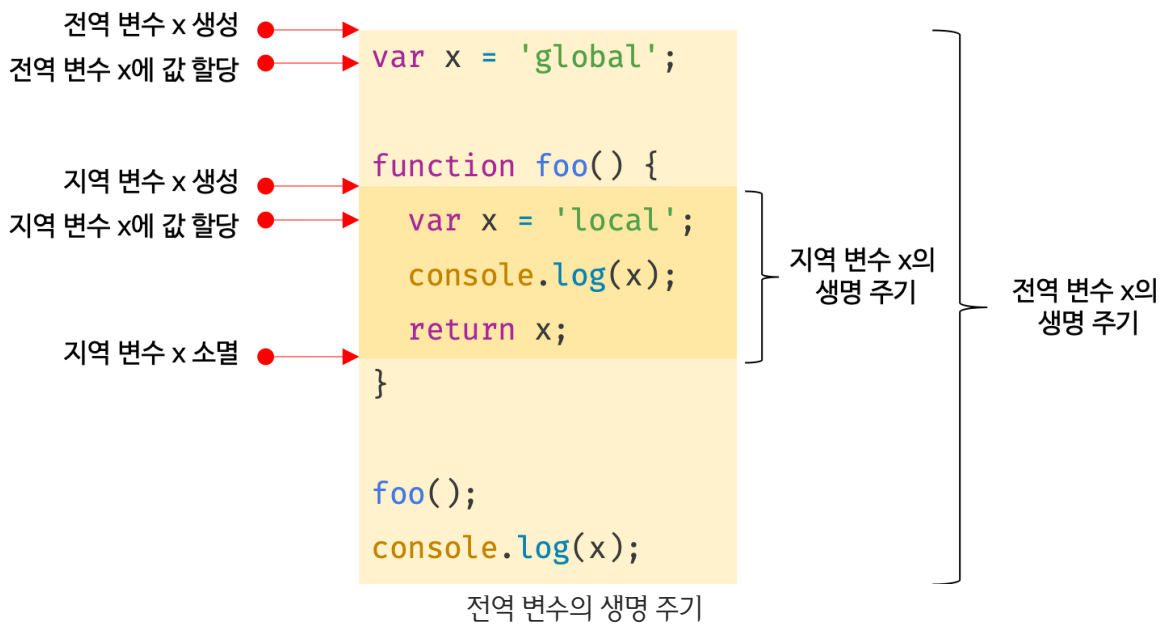
전역 객체(Global Object)는 코드가 실행되기 이전 단계에 자바스크립트 엔진에 의해 어떤 객체보다도 먼저 생성되는 특수한 객체이다. 전역 객체는 클라이언트 사이드 환경(브라우저)에서는 window, 서버 사이드 환경(Node.js)에서는 global 객체를 의미한다.

전역 객체에는 표준 빌트인 객체(Object, String, Number, Function, Array...)들과 환경에 따른 호스트 객체(클라이언트 web API 또는 Node.js의 호스트 API), 그리고 **var** 키워드로 선언한 전역 변수와 전

역 함수를 프로퍼티로 갖는다.

전역 객체와 표준 빌트인 객체에 대해서는 “21. 빌트인 객체”에서 자세히 살펴볼 것이다.

브라우저 환경에서 전역 객체는 window이므로 브라우저 환경에서 var 키워드로 선언한 전역 변수는 전역 객체 window의 프로퍼티이다. 전역 객체 window는 웹페이지를 종료하기 전까지 유효하다. 따라서 브라우저 환경에서 var 키워드로 선언한 전역 변수는 웹페이지를 종료할 때까지 유효하다. 즉, var 키워드로 선언한 전역 변수의 생명 주기는 전역 객체의 생명 주기와 일치한다.



2. 전역 변수의 문제점

암묵적 결합

전역 변수를 선언한 의도는 전역, 즉 코드 어디에서든지 전역 변수를 사용하겠다는 것이다. 이는 모든 코드가 전역 변수를 참조하고 변경할 수 있는 **암묵적 결합(implicit coupling)**을 허용하는 것이다. 변수의 유효 범위가 크면 클수록 코드의 가독성은 나빠지고 의도치 않게 상태가 변경될 수 있는 위험성도 높아진다.

긴 생명 주기

전역 변수는 생명 주기가 길다. 따라서 메모리 리소스도 오랜 기간 소비한다. 또한 전역 변수의 상태를 변경할 수 있는 시간도 길고, 모든 함수가 참조할 수 있기 때문에 상태를 변경할 기회도 많다.

더욱이 `var` 키워드는 변수의 중복 선언을 허용하므로 생명 주기가 긴 전역 변수는 변수 이름이 중복될 가능성이 있다. 변수 이름이 중복되면 의도치 않은 재할당이 이루어진다.

JAVASCRIPT

```
var x = 1;

// ...

// 변수의 중복 선언. 기존 변수에 값을 재할당한다.
var x = 100;

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

지역 변수는 전역 변수보다 생명 주기가 훨씬 짧다. 크지 않은 함수의 지역 변수는 생존 시간이 극히 짧다. 따라서 지역 변수의 상태를 변경할 수 있는 시간도 짧고 기회도 적다. 이는 전역 변수보다 상태 변경에 의한 오류가 발생할 확률이 작다는 것을 의미한다. 또한 메모리 리소스도 짧은 기간만 소비한다.

스코프 체인 상에서 종점에 존재

전역 변수는 또 하나의 문제는 스코프 체인 상에서 종점에 존재한다는 것이다. 이는 변수를 검색할 때 전역 변수가 가장 마지막에 검색된다는 것을 말한다. 즉, 전역 변수의 검색 속도가 가장 느리다. 검색 속도의 차이는 그다지 크지 않지만 속도의 차이는 분명히 있다.

네임 스페이스 오염

자바스크립트의 가장 큰 문제점 중 하나는 파일이 분리되어 있다하여도 하나의 전역 스코프를 공유한다는 것이다. 따라서 다른 파일 내에서 동일한 이름으로 명명된 변수나 함수가 같은 스코프 내에 존재할 경우 예상치 못한 결과를 가져올 수 있다.

3. 전역 변수 사용 억제 방법

전역 변수의 무분별한 사용은 위험하다. 전역 변수를 반드시 사용하여야 할 이유를 찾지 못한다면 지역 변수를 사용하여야 한다. 변수의 스코프는 좁을수록 좋다. 전역 변수를 절대 사용하지 말라는 의미는 아니

다. 무분별한 전역 변수의 남발은 억제해야 한다는 것이다. 전역 변수의 사용을 억제할 수 있는 몇가지 방법에 대해 살펴보자.

3.1. 즉시 실행 함수

함수 정의와 동시에 호출되는 즉시 실행 함수는 단 한번만 호출된다. 모든 코드를 즉시 실행 함수로 감싸면 모든 변수는 즉시 실행 함수의 지역 변수가 된다. 이러한 특성을 이용해 전역 변수의 사용을 제한하는 방법이다.

JAVASCRIPT

```
(function () {  
    var foo = 10; // 즉시 실행 함수의 지역 변수  
    // ...  
})();  
  
console.log(foo); // ReferenceError: foo is not defined
```

이 방법을 사용하면 전역 변수를 생성하지 않으므로 라이브러리 등에 자주 사용된다.

3.2. 네임 스페이스 객체

전역에 네임 스페이스(Namespace) 역할을 담당할 객체를 생성하고 전역 변수처럼 사용하고 싶은 변수를 프로퍼티로 추가하는 방법이다.

가.

JAVASCRIPT es3

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

네임 스페이스 객체에 또 다른 네임 스페이스 객체를 프로퍼티로 추가하여 네임 스페이스를 계층적으로 구성할 수도 있다.

JAVASCRIPT

```
var MYAPP = {}; // 전역 네임 스페이스 객체

MYAPP.person = {
  name: 'Lee',
  address: 'Seoul'
};

console.log(MYAPP.person.name); // Lee
```

네임 스페이스를 분리하여 식별자 충돌을 방지하는 효과는 있으나 네임 스페이스 객체 자체가 전역 변수에 할당되므로 그다지 유용해 보이지는 않는다.

3.3. 모듈 패턴

x

모듈 패턴은 클래스를 모방하여 관련이 있는 변수와 함수를 모아 즉시 실행 함수로 감싸 하나의 모듈을 만든다. 모듈 패턴은 자바스크립트의 강력한 기능인 클로저를 기반으로 동작한다. 모듈 패턴의 특징은 전역 변수의 억제는 물론 캡슐화까지 구현할 수 있다는 것이다.

캡슐화는 외부에 공개될 필요가 없는 정보를 외부에 노출시키지 않고 숨기는 것을 말하며 정보 은닉 (information hiding)이라고도 한다. Java의 경우, 클래스를 구성하는 멤버에 대하여 public, private, protected 등의 접근 제한자(Access modifier)를 사용해 공개 범위를 한정할 수 있다. public으로 선언된 데이터 또는 메소드는 외부에서 접근이 가능하지만 private으로 선언된 경우는 외부에서 접근할 수 없고 내부에서만 사용된다. 이것은 클래스 외부에는 제한된 접근 권한을 제공하며 원하지 않는 외부의 접근에 대해 내부를 보호하는 기능을 한다.

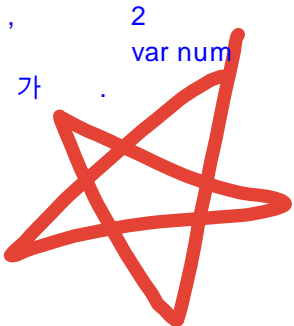
하지만 자바스크립트는 public, private, protected 등의 접근 제한자를 제공하지 않는다. 모듈 패턴은 전역 네임 스페이스의 오염을 막는 기능은 물론 한정적이기는 하지만 캡슐화를 구현하기 위해 사용한다.

JAVASCRIPT

```
var Counter = (function () {
  // private 변수
  var num = 0;
```



```
// 외부로 공개할 데이터나 메소드를 프로퍼티로 추가한 객체를 반환한다.
return {
  increase() {
    return ++num;
  },
  decrease() {
    return --num;
  }
};
}());
```



```
// private 변수는 외부로 노출되지 않는다.
console.log(Counter.num); // undefined

console.log(Counter.increase()); // 1
console.log(Counter.increase()); // 2
console.log(Counter.decrease()); // 1
console.log(Counter.decrease()); // 0
```

위 예제의 즉시 실행 함수는 객체를 반환한다. 이 객체에는 외부에 노출하고 싶은 변수나 함수를 담아 반환한다. 이때 반환되는 객체의 프로퍼티는 외부에 노출되는 퍼블릭 멤버(public member)이다. 외부로 노출하고 싶지 않은 변수나 함수는 반환하는 객체에 추가하지 않으면 외부에서 접근할 수 없는 프라이빗 멤버(private member)가 된다. 이에 대해서는 “24. 클로저”에서 자세히 살펴보도록 하자.

public .

3.4. ES6 모듈

전역 변수의 남발을 억제하기 위해 ES6에서 도입된 모듈을 사용할 수도 있다. 모던 브라우저(Chrome 61, FF 60, SF 10.1, Edge 16 이상)에서 ES6 모듈을 사용할 수 있다.

script 태그에 type="module" 어트리뷰트를 추가하면 로드된 자바스크립트 파일은 모듈로서 동작한다. 모듈의 파일 확장자는 mjs를 권장한다.

HTML

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

하지만 ES6 모듈은 IE를 포함한 구형 브라우저는 동작하지 않으며, 브라우저의 ES6 모듈 기능을 사용하더라도 트랜스파일링이나 번들링이 필요하기 때문에 아직까지는 브라우저가 지원하는 ES6 모듈 기능보다는 Webpack 등의 모듈 번들러를 사용하는 것이 일반적이다.

ES6 모듈 그리고 Webpack 등의 모듈 번들러를 도입하기 위한 방법에 대해서는 “ES6 모듈”, “Babel과 Webpack을 이용한 ES6 환경 구축”에서 자세히 살펴보기로 하자.