지역변수와 전역변수

지역 변수의 생명 주기

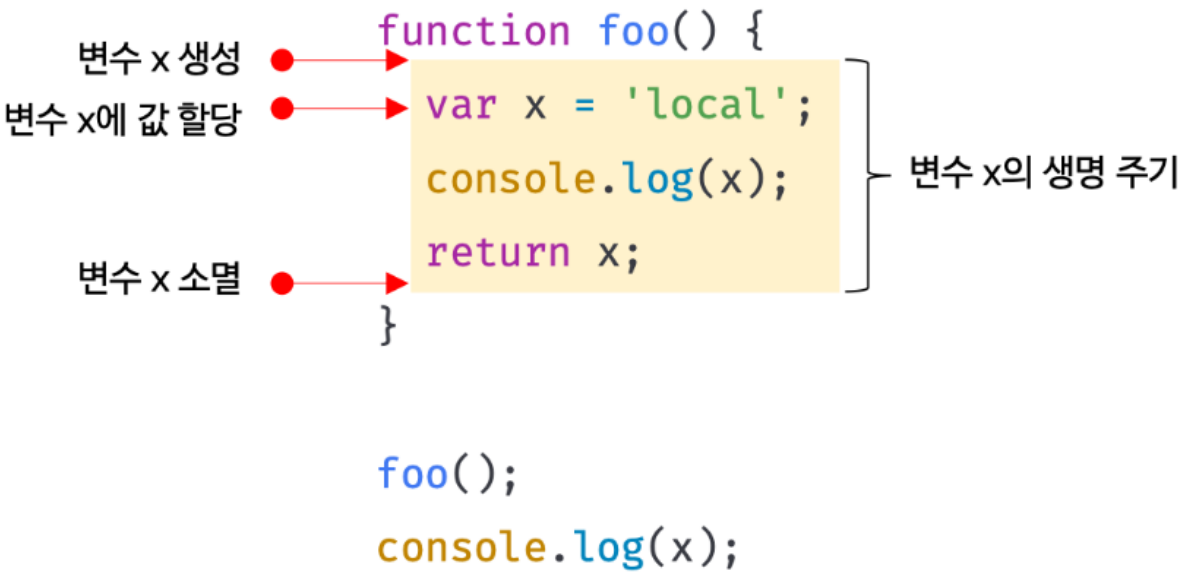
- 변수는 선언에 의해 생성되고 할당을 통해 값을 갖는다. 그리고 언젠가 소멸한다.
- 변수는 생성되고 소멸되는 생명 주기(life cycle)를 갖는다.
- 생명 주기가 없다면 한번 선언된 변수는 프로그램을 종료하지 않는 한 영원히 메모리 공간을 점유하게 된다.
- 변수는 자신이 선언된 위치에서 생성되고 소멸한다.
- 하지만 함수 내부에서 선언된 지역 변수는 함수가 호출되면 생성되고 함수가 종료하면 소멸한다.

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

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

- 변수 x는 foo함수가 호출되기 이전까지는 생성되지 않는다.
- foo함수를 호출하지 않으면 함수 내부의 변수 선언문이 실행되지 않기 때문이다.

💡 **지역변수의 생명 주기는 함수의 생명 주기와 일치한다**



▼ 연습 문제(지역변수와 전역변수)

다음 ①에서 출력되는 값은 무엇인가?

```
var x = 'global';

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

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

▼ 답

- foo함수 내부에서 선언된 변수 x는 ①의 시점에 이미 선언되었고 undefined로 초기화 되어 있다.
- 따라서 전역 변수 x를 참조하는 것이 아니라 지역변수 x를 참조해 값을 출력한다.
- 지역변수는 함수 전체에서 유효하다

> 💡 **호이스팅 = 변수 선언이 스코프의 선두로 끌어 올려진 것처럼 동작하는 자바스크립트 고유의 특징이다.**

전역 변수의 생명 주기

- 함수와 달리 전역 코드는 명시적인 호출 없이 실행된다.
- 전역 변수는 마지막 문이 실행되어 더 이상 실행할 문이 없을 때 종료된다.
- 전역 변수의 생명 주기 = 전역 객체의 생명 주기

▼ 전역 변수의 문제점

암묵적 결합

- 전역변수는 코드 어디서든 참조하고 할당할 수 있는 변수를 사용하기 위함이다.
- 변수의 유효 범위가 크면 클수록 코드의 가독성은 나빠지고 의도치 않게 상태가 변경될 수 있는 위험도 높아진다.

긴 생명 주기

- var 키워드는 변수의 중복 선언을 허용하므로 생명 주기가 긴 전역 변수는 의도치 않은 재할당이 이뤄질수 있다.

스코프 체인 상에서 종점에 존재

- 전역변수는 스코프 체인 상에서 종점에 존재한다.(가장 마지막에 검색된다.)
- 검색 속도가 가장 느리다. 검색 속도의 차이는 그다지 크지 않지만 속도의 차이는 분명히 있다.

네임스페이스 오염

- 다른 파일 내에서 동일한 이름으로 명명된 전역 변수나 전역 함수가 같은 스코프 내에 존재할 경우 예상치 못한 결과를 가져올 수 있다.

연습 문제

> starX 변수는 전역 변수와 printStarX()의 지역 변수로 선언되어 있습니다.
>
> printStar() 호출 시 파라미터로 3을 전달하면, '*'을 몇 개 출력하는지 실행 결과를 예측해 보세요

```
var starX = 1;
function printStar(Num){
    for(starX = 1; starX <= Num; starX++){
        printStarX(starX);
    }
}

function printStarX(Num){
    var starX;
    for(starX = 1; starX <= Num; starX++){
        console.log('*');
    }
}

printStar(3);
```

전역 변수의 사용을 억제하는 방법

- 전역 변수를 반드시 사용해야 할 이유를 찾지 못한다면 지역 변수를 사용해야 한다.

(변수의 스코프는 좁을수록 좋다)

즉시 실행 함수

모든 코드를 즉시 실행 함수로 감싸면 모든 변수는 즉시 실행 함수의 지역 변수가 된다.

```
(function () {
  var foo = 10; // 즉시 실행 함수의 지역 변수
  // ...
}());

console.log(foo); // ReferenceError: foo is not defined
※이 방법은 전역 변수를 생성하지 않으므로 라이브러리 등에 자주 사용된다.
```

네임 스페이스 객체

- 전역에 네임스페이스 역할을 담당할 객체를 생성하고 전역 변수 처럼 사용하고 싶은 변수를 프로퍼티로 추가하는 방법

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

//네임스페이스 객체에 또 다른 네임스페이스 객체를 프로퍼티로 추가해서 네임스페이스를 계층적으로 구성할 수 있다
var MYAPP = {}; // 전역 네임스페이스 객체
MYAPP.person = {
  name: 'Lee',
  address: 'Seoul'
};
console.log(MYAPP.person.name); // Lee
```

모듈 패턴

- 모듈 패턴은 클래스를 모방해서 관련이 있는 변수와 함수를 모아 즉시 실행 함수로 감싸 하나의 모듈을 만든다.

- 모듈 패턴의 특징은 **전역 변수의 억제**와 **캡슐화**를 구현할 수 있다.

※캡슐화(encapsulation) = 객체의 상태(state)를 나타내는 프로퍼티와 프로퍼티를 참조하고 조작할 수 있는 동작(behavior)인 메서드를 하나로 묶은 것 (객체의 특정 프로퍼티나 메서드를 감출 목적으로 사용= 정보 은닉)

- 대부분 객체지향 프로그래밍 언어는 클래스를 구성하는 멤버에 대해 public, private, protected등의 접근 제한자를 사용해 공개 범위를 한정할 수 있다.
- public으로 선언된 데이터 또는 메서드는 외부에서 접근이 가능하지만 private으로 선언된 경우는 외부에서 접근할 수 없다(외부의 접근으로 부터 보호)
- 자바스크립트는 접근 제한자를 제공하지 않는다. 모듈 패턴은 전역 네임스페이스의 오염을 막는 기능은 물론 한정적이지만 정보 은닉을 구현하기 위해 사용한다.

```
var Counter = (function () {
  // private 변수
  var num = 0; // 프라이빗 멤버 (외부에 노출되는 프로퍼티를 '퍼블릭 멤버'라고 한다.)

  // 외부로 공개할 데이터나 메서드를 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
```

▼ ES6 모듈

- ES6 모듈은 파일 자체의 독자적인 모듈 스코프를 제공하기에 전역변수를 사용할 수 없다.
- 모듈 내에서 var 키워드로 선언한 변수는 전역변수가 아니며 window 객체의 프로퍼티도 아니다.

script태그에 **type = "module"** 어트리뷰트를 추가하면 로드된 자바스크립트 파일은 모듈로 동작한다.

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

※IE를 포함한 구형 브라우저에서는 동작하지 않는다. 브라우저의 ES6모듈 기능을 사용하더라도 트랜스파일링이나 번들링이 필요하기 때문에 아직까지는 브라우저가 지원하는 ES6모듈 기능보다는 Webpack등의 모듈 번들러를 사용하는 것이 일반적이다.