스코프

배정	
상태	8월
속성	@2021년 8월 15일 오후 5:46
언어	

스코프(유효범위)

자바스크립트를 포함한 모든 프로그래밍 언어의 기본적이며 중요한 개념이다. 스코프의 이해가 부족하면 다른개념을 이해하기 어려울 수도 있으며, 더욱이 자바스크립트의 스코프는 다른 언어의 스코프와 구별되는 특징이 있으므로 주의가 필요하다. 또, 스코프는 변수 함수와 깊은 관련이 있다.

▼ 스코프의 개념과 정리

1. 함수의 매개변수 유효범위 (내부와 외부)

```
function add(x, y) {
  // 매개변수는 함수 몸체 내부에서만 참조할 수 있다.
  // 즉, 매개변수의 스코프(유효범위)는 함수 몸체 내부다.
  console.log(x, y); // 2 5
  return x + y;
}

add(2, 5);

// 매개변수는 함수 몸체 내부에서만 참조할 수 있다.
console.log(x, y); // ReferenceError: x is not defined
```

함수의 매개변수는 함수 몸체 내부에서만 참조할수있고 함수 몸체 외부에서는 참조할 수 없다. 매개변수를 참조할 수있는 유효범위, 즉 매개변수의 스코프가 함수 몸체로 한정되기 때문이다.

2. 변수의 경우

```
// 코드의 바깥쪽 혹은 코드블록이나 함수 몸체 내에서도 선언가능. 이럴때 중첩될수 있음.

var var1 = 1; // 코드의 가장 바깥 영역에서 선언한 변수
```

```
if (true) {
  var var2 = 2; // 코드 블록 내에서 선언한 변수
  if (true) {
    var var3 = 3; // 중첩된 코드 블록 내에서 선언한 변수
  }
}

function foo() {
  var var4 = 4; // 함수 내에서 선언한 변수

  function bar() {
    var var5 = 5; // 중첩된 함수 내에서 선언한 변수
  }
}

console.log(var1); // 1
console.log(var2); // 2
console.log(var3); // 3
console.log(var4); // ReferenceError: var4 is not defined
console.log(var5); // ReferenceError: var5 is not defined
```

변수는 자신이 선언된 위치에 의해 자신이 유효한 범위, 다른코드가 변수 자신을 참조 할 수 있는 범위가 결정됨. 모든 식별자(변수이름, 함수이름, 클래스 이름 ..)는 자신이 선언된 위치에 의해 다른 코드가 식별자 자신을 참조 할 수 있는 유효 범위가 결정됨.

3. 식별자 결정

```
// 같은 이름을 갖는 x변수를 선언함
var x = 'global';

// 이름이 같은 두개 변수중에 어떤 변수를 참조해야할지 결정함
function foo() {
  var x = 'local';
  console.log(x); // 1번 'local'
}

foo();

console.log(x); // 2번 'global'
```

자바스크립트 엔진은 스코프를 통해 어떤 변수를 참조해야할것인지 결정함. 따라서 스코프란 자바스크립트 엔진이 식별자를 검색할때 사용하는 규칙이라고 할 수 있음.

자바스크립트엔진은 코드를 실행할때 코드의 문맥을 고려함. 코드가 어디서 실행되며 주변에 어떤 코드가 있는지에 따라 1과 2처럼 동일한 코드도 다른 결과를 만들어낸다.

만약 스코프라는 개념이 없다면 ?

변수는 충돌을 일으키므로 프로그램 전체에서 하나밖에 사용 할 수 없다. 변수와 함수의 이름과 같은 식별자는 어떤 값을 구별하여 식별해 낼 수 있는 고유한 이름을 말한다. 사람을 고유한 이름으로 구별하듯이 값도 사람이 이해할 수 있는 언어로 지정한 고유한 식별자인 변수이름에 의해 구별하여 참조 할 수 있다.

프로그래밍 언어에는 스코프를 통해 식별자인 변수 이름의 충돌을 방지하여 같은 이름의 변수를 사용 할 수 있게한다. 스코프 내에서 식별자는 유일해야 하지만 다른 스코프에는 같은 이름의 식별자를 사용 할 수 있다. 즉 스코프는 '이름공간'이다.

▼ 스코프의 종류

전역 vs 지역

Aa 구분	설명	스코프	변수	특징
전역	코드의 가장 바깥	전역	전역변수	어디서든 참조가능
지역	함수 몸체 내부	지역	지역변수	자신의 지역과 하위지역

▼ 스코프 체인

```
// 스코프체인을 통해 변수를 참조하는 코드의 스코프에서 시작하여
// 상위 스코프방향으로 이동하며 선언된 변수를 검색함.

// 전역 함수
function foo() {
  console.log('global function foo');
}

function bar() {
  // 중첩 함수
  function foo() {
    console.log('local function foo');
  }
  // bar함수 스코프내에서의 foo를 실행함.
  foo(); // 'local function foo'
}

foo(); // 'global function foo'
bar();
```

함수는 중첩될 수있으므로 함수의 지역 스코프도 중첩될 수 있음. 이는 스코프가 함수의 중첩에의해 계층적인 구조를 갖는다는것을 의미함. 모든 지역의 최상의 스코프는 전역스코프가 되며, 스코프가 계층적으로 연결된 것을 '스코프 체인'이라 한다.

상위 스코프에서 유효한 변수는 하위스코프에서 자유롭게 참조할 수 있지만 하위 스코프에서 유효한 변수를 상위 스코프에서 참조할 수 없음.

▼ 함수레벨 스코프

> *"함수내부 몸체(지역) —> 지역스코프".*

이는 코드 블록이 아닌 함수에 의해서만 지역 스코프가 생성된다는 의미이다.

이러한 특성을 '블록 레벨 스코프'라 한다. 하지만 var키워드로 선언된 변수는 오로지 함수의 코드블록(함수 몸체)만을 지역 스코프로 인정한다. 이러한 특성을 '함수 레벨 스코프'라 한다.

```
var x = 1;

if (true) {
  // var 키워드로 선언된 변수는 함수의 코드 블록(함수 몸체)만을 지역 스코프로 인정한다.
  // 함수 밖에서 var 키워드로 선언된 변수는 코드 블록 내에서
  // 선언되었다 할지라도 모두 전역 변수다.
  // 따라서 x는 전역 변수다. 이미 선언된 전역 변수 x가 있으므로 x 변수는 중복 선언된다.
  // 이는 의도치 않게 변수 값이 변경되는 부작용을 발생시킨다.
  var x = 10;
}

console.log(x); // 10
-------------------------------------------------------------------------
var i = 10;

// for 문에서 선언한 i는 전역 변수다. 이미 선언된 전역 변수 i가 있으므로 중복 선언된다.
for (var i = 0; i < 5; i++) {
  console.log(i); // 0 1 2 3 4
}

// 의도치 않게 변수의 값이 변경되었다.
console.log(i); //
```

▼ 렉시컬 스코프

```
1. 함수를 어디서 호출했는지에 따라 상위스코프를 결정한다.

2. 함수를 어디서 정의했는지에 따라 상위스코프를 결정한다.
```

프로그래밍 언어는 일반적으로 위의 두가지 방식중 한가지 방식으로 함수의 상위스코프를 결정한다.

첫번째 방식을 '동적 스코프' 라 한다. 함수를 정의하는 시점에선 어디서 함수가 호출될지 알수가 없기때문이다. 따라서, 함수가 호출되는 시점에 동적으로 상위스코프를 결정한다.

두번째 방식을 '렉시컬 스코프' 혹은 '정적 스코프' 라고 한다. 동적스코프 처럼 상위스코프가 동적으로 변하지않고, 함수의 정의가 평가되는 시점에 상위 스코프가 정적으로 결정되기 때문이다.

자바스크립트를 비롯한 대부분의 프로그래밍 언어는 '렉시컬 스코프'를 따른다.

즉, 어디서 호출했는지가 아니라 함수를 어디서 정의했는지에 따라 상위 스코프를 결정한다. 함수가 호출된 위치는 상위 스코프 결정에 어떠한 영향도 주지않는다. 즉, 함수의 상위 스코프는 언제나 자신이 정의된 스코프이다.

> "정의된 함수 실행될때 상위 스코프가 결정됨 -> 생성된 함수객체는 결정된 상위 스코프를 기억함"

렉시컬 환경의 '외부 렉시컬환경에 대한 참조'에 저장할 참조값,

즉 상위 스코프에 대한 참조는 함수 정의가 평가되는 시점에 함수가 정의된 환경(위치)에 의해 결정된다. 이것이 바로 렉시컬 스코프이다.

예제)

```
var x = 1;

function foo() {
  var x = 10;
  bar();
}

function bar() {
  console.log(x);
}

foo(); // 1
bar(); // 1
```