함수 심화학습

재귀와 스택

- 함수 내부에서 자기 자신(함수)을 호출하는 경우 이를 **재귀(recursion)**라고 함

반복문으로 작성한 거듭 제곱 함수(pow) 코드

```
function pow(x, n) {
    let result = 1;

    // 반복문을 돌면서 x를 n번 곱함
    for (let i = 0; i < n; i++) {
        result *= x;
    }

    return result;
}

alert( pow(2, 3) ); // 8
```

재귀적으로 작성한 거듭 제곱 함수(pow) 코드

```
function pow(x, n) {
    if (n == 1) {
        return x;
    } else {
        // 함수 내부에서 자기 자신(pow)을 호출
        return x * pow(x, n - 1);
    }
}

alert( pow(2, 3) ); // 8
```

- pow는 n == 1이 될 때까지 재귀적으로 자신을 호출

pow(2, 4) 호출 시 함수 호출 진행 과정

```
pow(2, 4) = 2 * pow(2, 3)
pow(2, 3) = 2 * pow(2, 2)
pow(2, 2) = 2 * pow(2, 1)
pow(2, 1) = 2
```

pow(2, 4) 호출 시 함수 호출 진행 과정 (다르게 표현)

```
pow(2, 4) = 2 * (2 * (2 * 2))
```

- 가장 처음 하는 호출을 포함한 **중첩 호출의 최대 개수는 재귀 깊이(recursion depth)**라고 함
  - pow(x, n)의 재귀 깊이는 n
    - pow(2, 4)의 경우 함수 호출 스택을 4 단계까지 쌓음

```
pow(2, 4) => 2 * pow(2, 3) // pow(2, 3) 호출 (1스택)
                => 2 * pow(2, 2) // pow(2, 2) 호출 (2스택)
                       => 2 * pow(2, 1) // pow(2, 1) 호출 (3스택)
                              => 2 // (4스택)
```

- 자바스크립트 엔진은 **최대 재귀 깊이를 제한**하므로 무한히 재귀 호출을 진행할 수는 없음

실행 컨텍스트와 스택

- 실행 중인 함수의 실행 절차에 대한 정보는 해당 **함수의 실행 컨텍스트(execution context)**에 저장됨
  - 실행 컨텍스트는 **함수 실행에 대한 세부 정보를 담고 있는 내부 데이터 구조**
  - 제어 흐름의 현재 위치, 변수의 현재 값, this의 값 등 상세 내부 정보가 실행 컨텍스트에 저장
  - 함수 호출 일회당 정확히 하나의 실행 컨텍스트가 생성됨
- 함수 내부에서 **함수의 중첩 호출**이 있을 경우 아래와 같은 절차가 수행됨

```
1. 현재 함수의 실행이 일시 중지됨
2. 중지된 함수와 연관된 실행 컨텍스트는 실행 컨텍스트 스택(execution context stack) 이라는 특별한 자료 구조에 저장됨
3. 중첩된 함수의 호출이 실행됨
4. 중첩 호출 실행이 끝난 이후 실행 컨텍스트 스택에서 일시 중단한 함수의 실행 컨텍스트를 꺼내오고, 중단한 함수의 실행을 다시 진행함
```

실행 컨텍스트 (보충 설명) (@) (***)

- 오래된 var 내용 먼저 살펴봐도 무방
- **전역 실행 컨텍스트 (Global Execution Context)**
  - 스크립트가 시작되면 곧바로 **전역 실행 컨텍스트가 생성되고 실행**됨
  - 특정 "함수" 안에서 실행되는 코드가 아니라면 모두 전역 컨텍스트에서 실행
  - 전역 컨텍스트를 생성하는 시점에 **세 가지 작업이 수행**됨
    1. 전역 객체를 생성 (브라우저에서는 window, 노드인 경우 global)
    2. this를 전역 객체로 할당
    3. 사용할 값들(ex: 변수, 함수 등)을 위한 메모리 할당 및 초기화

```
<script>
</script>
```

- 위의 script 태그 내부의 코드 내용은 비어있지만, 어쨌든 전역 실행 컨텍스트가 생성되고 실행 단계로 진입함

- 내용은 없지만 1, 2의 작업이 수행되었으므로, 다음과 같이 this 값 및 전역 객체는 확인이 가능 (사용할 값들이 아무것도 없으므로 3의 작업 결과는 없음)

```
console.log(window);
console.log(this); // window 전역 객체 출력
```

- 실행 컨텍스트의 경우 항상 **두 가지의 단계를 거쳐 실행**됨
  - **생성 단계 (creation phase)**
    - 코드를 본격적으로 실행하기 전 실행할 코드를 분석하여 초기 컨텍스트를 만드는 단계
  - **실행 단계 (execution phase)**
    - 코드를 한 줄씩 실행하며 컨텍스트의 내용을 수정해 나가는 단계

```
<script>
var a = 100;
let b = 200;
function f() { console.log('hello'); }
a = 300;
b = 400;
</script>
```

- 위의 script 코드 내용은 생성 단계에서 코드를 분석 후 다음과 같은 전역 실행 컨텍스트를 만듦

```
global => window
this => window
a => undefined
b => <uninitialized>
f => f() { console.log('hello'); }
```

- 이후 실행 단계에 진입하며 코드를 한 줄 씩 실행
- 마지막 코드까지 실행 한 후의 실행 컨텍스트의 모습은 다음과 같음

```
global => window
this => window
a => 300
b => 400
f => f() { console.log('hello'); }
```

- **함수 실행 컨텍스트 (Function Execution Context)**
  - 새로운 함수가 실행될 때마다 **매번 새로운 함수 실행 컨텍스트가 생성 단계로 진입하여 생성되고 이후 실행 단계로 진입**함

```
<script>
var a = 100;
let b = 200;
function add(x, y) {
    console.log(arguments);
    return x + y;
}
let result = add(a, b);
</script>
```

- 위의 script 코드 내용은 생성 단계에서 코드를 분석 후 다음과 같은 전역 실행 컨텍스트를 만듦

```
a => undefined
b => <uninitialized>
result => <uninitialized>
add => add(x, y) {
    console.log(arguments);
    return x + y;
}
```

- 마지막 라인에서 add 함수를 실행하므로 함수 실행 컨텍스트가 "생성 단계"에 진입하여 생성됨
    - 전역 컨텍스트 컨텍스트 생성과는 다르게 함수 실행 컨텍스트에서는 ...
        - 전역 객체 대신 접근할 수 있는 **arguments 유사 배열이 생성**됨
            - arguments에는 전달된 인자값들이 모두 저장됨
        - (전역 컨텍스트는 this 값이 전역 객체로 설정되는 것과 달리) this 값이 **함수 호출 맥락에 의해서 정해짐**

생성된 add 함수 실행 컨텍스트의 내용

```
arguments => [ 100, 200 ]
this => undefined ((*) => 엄격 모드라고 가정)
x => 100
y => 200
```

- 함수 실행 컨텍스트는 **함수 실행이 종료되는 시점**에 반환값을 반환하며 사라지게 됨

함수 내부에서 또 함수를 호출하는 스크립트 예시

- 함수 호출시마다 함수 호출 컨텍스트가 생성되므로 함수 내부에서 함수를 생성할 때에도 호출 스택에 새로운 함수 호출 컨텍스트가 생성됨

```
<script>
var a = 100;
let b = 200;
function addAndDouble(x, y) {
    return double(x + y);
}
function double(x) {
    return 2 * x;
}
let result = addAndDouble(a, b);
</script>
```

addAndDouble 함수 내부에서 double 함수를 실행하는 시점의 실행 컨텍스트 상황

(GEC => Global Execution Context, FEC => Function Execution Context)

```
[GEC]
global => window
this => window
a => 100
b => 200
addAndDouble => f
double => f
result => <uninitialized>

    [addAndDouble FEC]
```

```
    arguments => [ 100, 200 ]
    this => undefined
    x => 100
    y => 200

        [double FEC]
        arguments => [ 300 ]
        this => undefined
        x => 300
```

- 단, 여기서 addAndDouble이 호출되는 시점에 함수 호출 컨텍스트에서는 double 함수를 찾을 수 없으므로, double을 호출하기 위해서 전역 실행 컨텍스트를 참조함 (즉, **호출된 함수의 상위에 있는 실행 컨텍스트를 참조하며 검색을 진행**함)

```
<script>
var a = 100;
let b = 200;
function addAndDouble(x, y) {
    function double(x) {
        return 2 * x;
    }
    return double(x + y);
}
let result = addAndDouble(a, b);
</script>
```

- 위의 script 코드 실행 결과는 같지만, 실행 컨텍스트의 모습이 다름 (addAndDouble 내부에서 double을 호출할 때, 상위 실행 컨텍스트에서 찾지 않고 해당 함수의 실행 컨텍스트 내부에서 찾음)

addAndDouble 함수 내부에서 double 함수를 실행하는 시점(2 * x 코드)의 실행 컨텍스트 상황

```
[GEC]
global => window
this => window
a => 100
b => 200
result => <uninitialized>
addAndDouble => f

    [addAndDouble FEC]
    arguments => [ 100, 200 ]
    this => undefined
    x => 100
    y => 200
    double => f ((*) addAndDouble 함수 실행 컨텍스트의 생성 시점에 double을 위한 저장 공간 확보 및 함수 저장)

        [double FEC]
        arguments => [ 300 ]
        this => undefined
        x => 300
```

- **클로저의 실행 컨텍스트**
    - 함수 내부에서 함수를 반환할 경우 **함수가 만들어진 상황을 기억하는 클로저 환경이 함수와 같이 반환**되므로, 이후 반환 받은 함수를 호출할 때 해당 클로저 환경을 조사하여 함수가 생성

되는 시점에 참조한 외부 변수 및 환경에 접근 가능

```
<script>
let sum = 0;
let y = 100;
function makeAdder(init = 0) {
    let sum = init;
    return function(x) {
        sum += x + y;
        return sum;
    }
}
let adder = makeAdder(sum);
let result = adder(1); // result1 == 101
</script>
```

adder 함수가 호출되는 시점의 실행 컨텍스트 상황

```
[GEC]
global => window
this => window
sum => 0
y => 100
makeAdder => f
adder => f
result => <uninitialized>

    [adder의 클로저 컨텍스트]
    arguments => [ 0 ]
    this => undefined
    sum => 0
    init => 0

        [익명(anonymous) FEC]
        arguments => [ 1 ]
        this => undefined
        x => 1
```

- makeAdder에서 반환한 함수의 이름이 없으므로, adder를 호출하면 **익명 함수 컨텍스트에서 실행**됨
- adder 함수가 호출되는 시점에 **클로저 컨텍스트과 함께 전달**되며, adder 함수에서 찾을 수 없는 값(sum, y)에 접근할 때 클로저 컨텍스트를 참조함
    - 클로저 컨텍스트가 **외부 실행 컨텍스트(여기서는 전역 실행 컨텍스트)를 참조**하기 때문에 클로저 컨텍스트에 존재하지 않는 y 값에도 접근이 가능함

Q1) johnSayHelloToSally가 호출되는 시점의 실행 환경 컨텍스트 분석해보기

```
<script>
let person = { name: "John" };
person.makeSayHelloTo = function(to) {
    let name = this.name;
    return function() {
        console.log(name, 'say hello to', to);
    }
}
let johnSayHelloToSally = person.makeSayHelloTo('Sally');
johnSayHelloToSally();
</script>
```

```
[GEC]
global => window
this => window
person => { name: "John", makeSayHelloTo: f }
johnSayHelloToSally => f

    [johnSayHelloToSally의 클로저 컨텍스트]
    arguments => [ 'Sally' ]
    this => person
    name => "John"
    to => "Sally"

        [익명(anonymous) FEC]
        arguments => []
        this => undefined
```

Q2) 다음과 같이 코드 변경 시 문제점은?

```
person.makeSayHelloTo = function(to) {
    return function() {
        console.log(this.name, 'say hello to', to);
    }
}
```

- 이후 "변수의 유효범위와 클로저 파트"에서 컨텍스트에 접근하는 과정에서 필요한 정보를 제공하는 **환경 레코드(Environment Record, 접근할 수 있는 값에 대한 정보 제공)와 외부 렉시컬 환경(Outer Lexical Environment, 바깥 컨텍스트 환경에 대한 정보 제공)**에 대해서 학습함

나머지 매개변수와 전개 문법 (@)

- 자바스크립트 언어에서 **임의의 개수의 인자값을 전달**받는 방법 배우기
- 함수를 호출할 때 **배열의 내용을 가변 인자의 형태로 전달**하는 방법 배우기

나머지 매개변수 ...

- 자바스크립트에서는 **함수에 넘겨주는 인수의 개수에 제약이 없음**

```
// 2개의 인자값을 전달받는 함수 정의
function sum(a, b) {
    return a + b;
}

// 호출할 때 원하는 개수만큼 인자값 전달 가능 (하지만 함수에서는 1, 2만 사용)
alert( sum(1, 2, 3, 4, 5) );
```

- 함수를 정의할 땐 인수를 두 개만 받도록 하고, 실제 **함수를 호출할 때 더 많은 인수를 전달해도 에러가 발생하지 않음**
    - 단, 반환값은 처음 두 개의 인수를 사용해서 계산
- 가변 인자를 전달받기 위해서는 **값들을 담을 배열 이름을 마침표 세 개(...)뒤에 붙여주면 됨**
    - 마침표 세 개(...)의 의미 => 나머지 매개변수들을 한데 모아서 **배열**에 집어넣으라는 의미

```
function sumAll(...args) {
    // args는 배열
    console.log(Array.isArray(args)); // true
    let sum = 0;
    for (let arg of args) sum += arg;
    return sum;
}

// args => [1]
alert( sumAll(1) );
// args => [1, 2]
alert( sumAll(1, 2) );
// args => [1, 2, 3]
alert( sumAll(1, 2, 3) );
```

- 앞부분의 매개변수는 변수로, 그 이외의 매개변수들은 배열로 받을 수도 있음
    - 단, 가변 인자를 받을 배열(...이 붙는 배열)은 반드시 **맨 마지막에 정의**해야 함

```
// titles가 가변 인자값을 전달받기 때문에 맨 마지막에 정의
function showName(firstName, lastName, ...titles) {
    alert( firstName + ' ' + lastName ); // Julius Caesar

    // 나머지 인수들은 배열 titles의 요소로 포함됨
    // titles => ["Consul", "Imperator"]
    alert( titles[0] ); // Consul
    alert( titles[1] ); // Imperator
    alert( titles.length ); // 2
}

// "Julius", "Caesar"
showName("Julius", "Caesar", "Consul", "Imperator");
```

arguments 변수

- arguments라는 이름의 특별한 **유사 배열 객체(array-like object)**를 이용하면 인덱스를 사용해 전달받은 모든 인수에 접근 가능
  - arguments는 함수 호출 시점에 함수 내부에서 접근할 수 있는 특수한 객체 (함수를 호출할 때 함수 실행 컨텍스트의 생성 단계에서 만들어지는 객체)
  - arguments는 **유사 배열 객체이면서 이터러블(반복 가능한) 객체**
    - 즉, 엄격히 말하자면 배열은 아니므로, **배열에서 제공하는 메서드를 사용할 수는 없음**
    - 이터러블 객체이므로 순회는 가능함

```
function sumAll() {
    // arguments는 배열은 아님
    console.log(Array.isArray(arguments)); // false
    let sum = 0;
    // 하지만, 이터러블 객체이므로 배열 처럼 요소 순회는 가능
    for(let arg of arguments) sum += arg;
    return sum;
}

console.log( sumAll(1, 2) ); // 3
console.log( sumAll(1, 2, 3) ); // 6
```

- **나머지 매개변수는 최근에 개정된 문법**에서 제공하는 기능이므로, 과거엔 함수의 인수 전체를 얻어내는 방법이 arguments를 사용하는 것밖에 없어서 자주 사용되었음 (따라서 레거시 코드를 분석할 때 알아야 함)

```
function showName() {
    alert( arguments.length );
    alert( arguments[0] );
    alert( arguments[1] );

    // arguments는 이터러블 객체이기 때문에 다음과 같이 for ... of 구문을 사용해서 인수
나열 가능
    // for(let arg of arguments) alert(arg);
}

// 2, Julius, Caesar가 출력됨
showName("Julius", "Caesar");

// 1, Bora, undefined가 출력됨(두 번째 인수는 없음)
showName("Bora");
```

- 화살표 함수가 this를 가지지 않고 바깥 환경에서 빌려오는 것과 같이, 화살표 함수는 자체 arguments도 가지지 않음 (화살표 함수 다시 살펴보기 항목에서 해당 내용 설명)

```
function f() {
    // 화살표 함수 바깥의 함수(f)의 arguments를 참조 (자체 arguments를 가지지 않음)
    let showArg = () => console.log(arguments); // [1, "Hello"]
    showArg();

    // 인자값을 여럿 전달했지만 바깥의 arguments를 참조
    let another = () => console.log(arguments); // [1, "Hello"]
    another(2, "World");
```

```
    function g() {
        // 화살표 함수 바깥의 함수(여기서는 g)의 arguments를 참조
        let showArg = () => console.log(arguments); // [2, "World"]
        showArg();
        let another = () => console.log(arguments); // [2, "World"]
        another(2, "World");
    }
    g(2, "World");
}

f(1, "Hello");
```

spread 문법

- **전개 문법(spread syntax)을 이용하여 배열을 가변 인자를 받는 함수에 전달 가능**
    - 점 세 개(...)를 사용하기 때문에 나머지 매개변수와 비슷해 보이지만, **전개 문법은 나머지 매개변수와 반대의 역할을 함을 유의**

```
alert( Math.max(3, 5, 1) ); // 5

let arr = [3, 5, 1];
// 배열 직접 전달은 불가 (max는 가변 인수를 전달받는 함수)
alert( Math.max(arr) ); // NaN

// "전개 문법"을 사용하여 배열의 내용을 가변 인자로 전달 가능
alert( Math.max(...arr) ); // 5

let arr1 = [1, -2];
let arr2 = [8, -8];

// 이터러블 객체 여러 개를 전달하는 것도 가능
alert( Math.max(...arr1, ...arr2) ); // 8
// 위의 코드는 결과적으로 아래와 같은 코드를 실행
// alert( Math.max( 1, -2, 8, -8 ) );
```

- 배열을 합치고자 할 때에도 전개 문법 사용 가능

```
let arr1 = ["Hello", 1234];
let arr2 = [5678, "World"];

let merged = [1, ...arr1, 2, ...arr2];
alert(merged); // [1,"Hello",1234,2,5678,"World"]
```

배열, 객체 복사 (Get a new copy of an array/object) (*)

- 전개 문법을 이용하여 배열, 객체의 내용을 복사 가능
    - 객체의 불변성을 이용하는 라이브러리에서 특히 자주 사용되는 기법

배열 복사

```
let arr = [1, 2, 3];
```

```
// 전개 연산자 써서 새 배열에 기존 배열의 내용을 복사 하기
let arrCopy = [...arr];
// 위의 코드가 같은 역할을 하는 밑의 코드(assign 함수 사용)보다 더 짧아서 자주 쓰임
// let arrCopy = Object.assign([], arr);

// 똑같은 내용을 가지지만
alert(JSON.stringify(arr) === JSON.stringify(arrCopy)); // true

// 같은 참조는 아닌 새 배열이 생성된 것을 확인 가능
alert(arr === arrCopy);  // false

// 참조 복사가 아니므로, 기존 배열의 변경이 복사된 배열에 영향을 끼치지 않음
arr.push(4);
alert(arr); // 1, 2, 3, 4
alert(arrCopy); // 1, 2, 3
```

객체 복사

```
let obj = { a: 1, b: 2, c: 3 };
let objCopy = { ...obj };
// 위의 코드가 같은 역할을 하는 밑의 코드(assign 함수 사용)보다 더 짧아서 자주 쓰임
// let objCopy = Object.assign({}, obj);

// 똑같은 내용을 가지지만
alert(JSON.stringify(obj) === JSON.stringify(objCopy)); // true

// 같은 참조는 아닌 새 객체가 생성된 것을 확인 가능
alert(obj === objCopy); // false (not same reference)

// 참조 복사가 아니므로, 기존 객체의 내용을 변경해도 복사된 객체에 영향을 끼치지 않음
obj.d = 4;
alert(JSON.stringify(obj)); // {"a":1,"b":2,"c":3,"d":4}
alert(JSON.stringify(objCopy)); // {"a":1,"b":2,"c":3}
```

- 객체 내부에 중첩된 객체가 있는 경우는 주의해서 사용해야 함

```
let person = {
    name: 'John',
    address: {
        city: 'Seoul',
        postal: '01234'
    }
};

// 기존 방식대로 복사 (personCopy1은 새로운 객체이지만, 내부 객체는 얕은 복사가 됨)
let personCopy1 = { ...person };
personCopy1.name = 'Jane';
console.log(personCopy1.name);          // Jane
console.log(person.name);               // John (영향 없음)
personCopy1.address.city = 'New York';
console.log(personCopy1.address.city);  // New York
console.log(person.address.city);       // New York (내부 객체 접근 시 원본 객체도
영향을 끼침)

// 중첩된 객체 복사 (personCopy2도 새로운 객체가 생성되며 복사되고 내부 객체(address)도
새로운 객체가 생성되며 내용이 복사됨)
```

```
let personCopy2 = { ...person, address: { ...person.address } };
personCopy2.address.city = 'Tokyo';
console.log(personCopy2.address.city);  // Tokyo
console.log(person.address.city);       // New York
```

변수의 유효범위와 클로저 (*****)

- 자바스크립트는 객체 지향 언어이면서 동시에 **함수 지향 언어의 특징**을 가짐
  - 함수를 **동적으로 생성** 가능
  - 생성한 함수를 값처럼 다른 함수에 인수로 넘길 수 있으며, 함수가 생성된 곳이 아닌 곳에서도 함수를 호출 가능 (가령, 함수에서 반환한 함수를 호출 가능 => 클로저)
  - 요약 => 함수를 **일급 객체**(https://en.wikipedia.org/wiki/First-class_citizen)로 취급
    - first-class function 항목 살펴보기

코드 블록

- 중괄호 코드 블록으로 진입하며 **새로운 접근 범위(scope)가 생성**되며, 접근 범위 안에서 선언한 변수는 **블록 안에서만 사용이 가능함**
  - 접근 범위 => 변수 저장할 수 있는 일종의 공간(혹은 객체)라고 이해하기

```
// 중괄호 블록 추가
{
    // 지역 변수에 몇 가지 조작을 하면, 그 결과를 밖에선 볼 수 없음
    let message = "Hello";
    // 블록 내에서만 블록에서 선언한 변숫값에 접근 가능
    alert(message);
}

// 중괄호 바깥에서는 접근 불가
// ReferenceError: message is not defined
alert(message);
```

- 다음의 예제 코드 결과를 오해하지 않도록 유의

```
let message;
{
    // 여기서는 중괄호 바깥(바깥 범위)의 message 변수를 참조하는 것이고 지역 변수를 선언하는 것이 아님을 유의
    message = "Hello";
    alert(message);
}
alert(message);
```

- 이미 선언된 변수와 동일한 이름을 가진 변수를 **별도의 블록 없이 let으로 선언하면 에러가 발생**

```
let message = "안녕하세요.";
alert(message);

// 변수 중복 선언 불가
// SyntaxError: Identifier 'message' has already been declared
let message = "안녕히 가세요.";
alert(message);
```

- 다음 코드는 잘 작동함

```
let message = "Hello";

// 새 코드 블록 생성
{
    let message = "안녕하세요.";
    // 바깥 변수의 이름을 지역 변수가 가려버리는 "변수 쉐도잉(shadowing)" 현상 발생
    alert(message); // "안녕하세요." 출력
}

// 다른 코드 블록 생성
{
    // 정상 작동
    let message = "안녕히 가세요.";
    alert(message); // "안녕히 가세요." 출력
}

alert(message); // "Hello" 출력
```

- if, for, while과 같은 제어문의 중괄호 블록 안에서 선언한 변수는 오직 블록 안에서만 접근 가능
    - 즉, **중괄호가 생기면 매번 새로운 접근 범위가 생성**된다고 이해하기
        - https://dev.to/murithi/scope-in-javascript-e5e
        - https://stackoverflow.com/questions/59170277/javascript-understanding-let-scope-inside-for-loop

```
if(true)
// 새 블록 생성 (=새 접근 범위(scope) 생성)
{
    let phrase = "안녕하세요!";
    alert(phrase); // 안녕하세요!
}
// 블록을 벗어나면 접근 불가
// ReferenceError: phrase is not defined
// alert(phrase);

// for 옆 괄호 안에서 let 키워드를 이용하여 선언한 변수(i)는 블록에 속하는 "지역 변수"로 취급됨을 유의
// 또한 매번 블록이 반복되어 실행됨에 따라 i 변수도 "독립된 값으로 새로 생성"됨을 유의
// https://stackoverflow.com/questions/59170277/javascript-understanding-let-scope-inside-for-loop
for(let i = 0; i < 3; i++)
// 새 블록 생성
{
    // 변수 i는 for 블록 안에서만 접근 및 사용 가능
    alert(i);
}
```

```
// 블록을 벗어나면 접근 불가
// ReferenceError: i is not defined
// alert(i);
```

중첩 함수

- 함수 내부에서 선언한 함수를 **중첩(nested) 함수**라고 부름
  - 중첩 함수는 **코드를 정돈하는데 사용** 가능
    - 즉, 거대한 작업을 하는 큰 함수의 기능들을 쪼개서 중첩 함수로 정의 가능
  - 중첩 함수는 해당 함수가 정의된 **함수 내부에서만 접근 가능** (함수 호출 컨텍스트 생각해보기)

```
function sayHiBye(firstName, lastName)
// 새 블록 생성
{
    // 중첩 함수 정의
    function getFullName() {
        return firstName + " " + lastName;
    }

    // 중첩함수 호출
    alert( "Hello, " + getFullName() );
    alert( "Bye, " + getFullName() );
}

sayHiBye("철수", "김");
// 중첩 함수는 함수 바깥에서 접근 불가
// getFullName();
```

- 중첩 함수는 함수의 반환값으로 반환될 수 있음 => 즉, 함수에서 함수 반환 가능
  - 꼭 중첨 함수가 아니어도 **익명 함수, 화살표 함수 형태**로도 함수 반환 가능
  - 함수 내부에서 생성한 객체의 프로퍼티 형태(메서드)로도 반환 가능
  - ```
    function makeObjectWithClosureMethod(value) {
        let x = 100;
        let obj = {
            method(arg) {
                // 메서드 형태이지만, 함수와 똑같이 메서드가 정의된 시점의 바깥 환경을 기억 가능
                console.log(value, x, arg);
            }
        };
        return obj;
    }
    let o = makeObjectWithClosureMethod("Hello");
    o.method("World");
    ```

- 이렇게 반환된 함수는 어디서든 호출해서 사용 가능
  - **반환된 함수가 호출될 때 함수에서 참조하고 있는 외부 변수에도 접근이 가능함 (클로저)**

```
function makeCounter() {
    let count = 0;
```
```

```
    // (자바스크립트에서는 함수가 값으로 취급되므로) 함수에서 함수 반환 가능, 여기서는 익명 함수를 반환하고 있음
    // 또한, 함수는 자신이 정의된 시점에 존재하는 바깥 환경을 기억함
    return function() {
        // 반환된 이후에도 호출되어 외부 영역의 값(count)에 접근 가능 (클로저)
        return count++;
    };
    // 밑의 코드도 똑같은 역할을 하는 함수를 반환
    // (단, 화살표 함수이므로 자체적으로 this, arguments 값을 가지지 않음을 유의)
    // return () => count++;
}

// 함수 반환
let counter = makeCounter();

// 반환받은 함수를 호출
alert( counter() ); // 0 (함수 호출 과정에서 함수가 정의되던 시점의 외부 환경 값(count)을 기억하여 접근 가능하므로 계속 count값이 증가함)
alert( counter() ); // 1
alert( counter() ); // 2
```

렉시컬 환경

단계 1. 변수

- 실행 중인 함수, 임의의 중괄호 코드 블록, 스크립트 전체는 **렉시컬 환경(Lexical Environment) 객체**라 불리는 내부 숨김 연관 객체(internal hidden associated object)를 갖음
  - 외부에는 숨겨진 객체이므로 코드를 이용한 접근은 불가
- 렉시컬 환경 객체는 **두 부분으로 구성**
  - **환경 레코드(Environment Record)** – 모든 지역 변수를 프로퍼티로 저장하고 있는 객체로 this 값과 같은 기타 정보도 여기에 저장됨
  - **외부 렉시컬 환경(Outer Lexical Environment)**에 대한 참조로 이 참조를 통해서 외부 코드와 연결이 가능
    - **전역 렉시컬 환경**에서는 외부 참조를 갖지 않기 때문에 null 값이 할당됨

```
// 전역 렉시컬 환경 (외부 렉시컬 환경(null)에 접근 불가)
console.log(this); // window

{
    // 여기(임의의 중괄호 코드 블록)서는 외부 렉시컬 환경에 접근 가능
    // ...
}

function f() {
    // 여기(실행 중인 함수)서는 외부 렉시컬 환경에 접근 가능
    // ...
}

f();
```

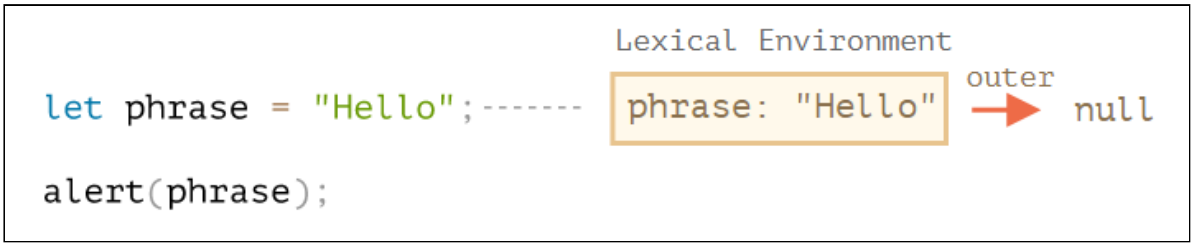
- 변수는 특수 내부 객체인 **환경 레코드의 프로퍼티**일 뿐이며 변수를 가져오거나 변경하는 것은 환경 레코드의 프로퍼티를 가져오거나 변경함과 동일한 의미를 가짐
- 스크립트 전체와 관련된 렉시컬 환경을 **전역 렉시컬 환경(Global Lexical Environment)**이라고 부름
- 렉시컬 환경은 명세서에서 자바스크립트가 어떻게 동작하는지 설명하는 데 쓰이는 **이론상의 객체**로 코드를 사용해 직접 렉시컬 환경에 접근하거나 조작하는 것은 불가능함

script.js

```
<script>
let phrase = "Hello";
alert( phrase );
</script>
```

위의 코드가 스크립트 전체라고 가정하면 아래의 그림과 같은 렉시컬 환경 생성 (전역 렉시컬 환경)

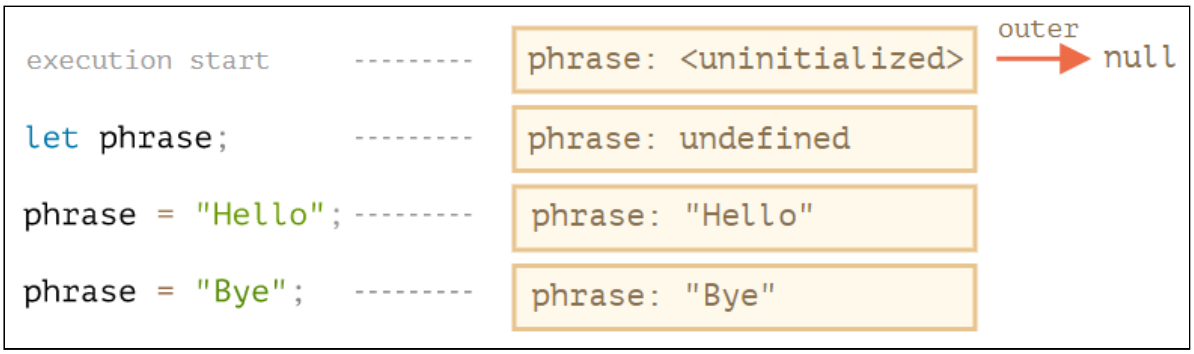
- 전역 호출 컨텍스트와 연관지어 설명하자면, **전역 호출 컨텍스트에서 전역 렉시컬 환경을 참조**할 수 있다고 이해하기 (당연히 함수 호출 컨텍스트에서는 해당 함수가 호출될 때 생성된 렉시컬 환경을 참조 가능)
- 네모 상자는 변수가 저장되는 환경 레코드
- 붉은 화살표는 **외부 참조(outer reference)**
  - 전역 렉시컬 환경이므로 여기서는 null을 가리킴



- 코드가 실행되는 과정(실행 단계)에서 렉시컬 환경은 변화하게 됨

```
let phrase;
phrase = "Hello";
phrase = "Bye";
```

위의 코드가 실행되는 양상 그림으로 그려보기



execution start (아직 코드 실행 전)

- 스크립트가 시작되면 스크립트 내에서 선언한 **변수 전체가 렉시컬 환경에 저장됨 (pre-populated, 호출 컨텍스트 생성 단계의 호이스팅 개념으로도 이해 가능)**
- 이때 변수의 상태는 특수 내부 상태(special internal state)인 **uninitialized 상태**가 됨

- 자바스크립트 엔진은 uninitialized 상태의 변수를 인지하긴 하지만, let 키워드를 이용한 변수 선언문을 만나기 전까지 변수 참조는 불가능함 (참조시 **ReferenceError 발생**)
    - let, const 키워드를 이용한 변수, 상수값 선언문을 만나기 전까지의 구역을 해당 값에 대한 Temporal Dead Zone이라고 부름
        - var 키워드를 이용해서 선언된 변수의 경우 호이스팅되는 시점에 바로 undefined 값으로 초기화되므로 TDZ와 같은 개념이 없음

1번째 라인 실행 후

- 변수 선언문(let phrase)을 만나게 되고, 아직 값을 할당하기 전이기 때문에 undefined 값으로 초기화되고 이 시점 이후부터는 phrase 값에 접근이 가능해 짐 (TDZ을 벗어난 상황)

2번째 라인 실행 후

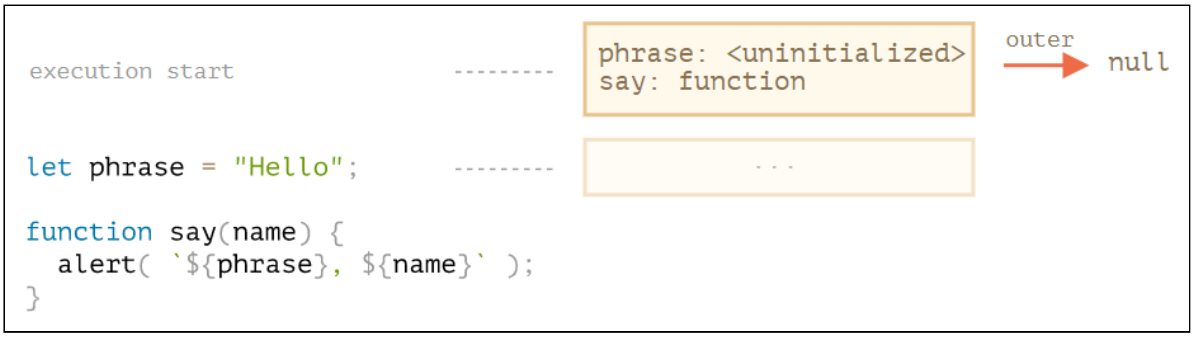
- phrase에 "Hello" 값이 할당

3번째 라인 실행 후

- phrase에 "Bye" 값이 할당

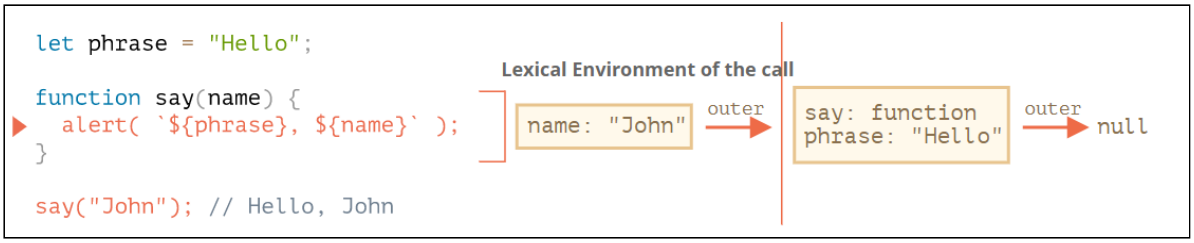
단계 2. 함수 선언문

- 함수는 변수와 마찬가지로 값으로 취급됨
- 단, **함수 선언문(function declaration)으로 선언한 함수는 일반 변수와는 달리 바로 초기화**된다는 점에서 차이
    - 함수 선언문으로 선언한 함수는 **렉시컬 환경이 만들어지는 즉시 사용 가능** (execution start 시점)
    - 함수 표현식(function expression)으로 정의한 함수는 let, const 키워드를 만난 후 대입이 이루어지기 전까지는 사용이 불가능함
    - 왜냐하면 **근본적으로는 함수가 저장된 변수, 상수**이기 때문에 처음 호이스팅되는 시점에는 uninitialized 값이 부여됨
        - 따라서, 변수에 함수를 할당하는 시점 이후부터 정상적인 사용(호출)이 가능
- 밑의 그림을 보면 코드 실행 이전(execution start) 시점(생성 단계)에 이미 say 함수가 렉시컬 환경에 올라오게 됨을 확인 가능



단계 3. 내부와 외부 렉시컬 환경

- **함수가 호출되는 시점에 새로운 렉시컬 환경이 자동으로 생성됨**
    - 함수 호출 컨텍스트가 새로 생성되며 동시에 연관된 렉시컬 환경도 새로 생성되는 것으로 이해하기
    - 렉시컬 환경엔 함수 호출 시 **넘겨받은 매개변수와 함수의 지역 변수**가 저장됨



- 여기서 say 함수가 호출 중인 동안은 **호출 중인 함수(say)를 위한 내부 렉시컬 환경과 내부 렉시컬 환경이 가리키는 외부 렉시컬 환경(여기서는 전역 렉시컬 환경) 두 개**를 갖게 됨
- 코드에서 변수에 접근할 땐, **먼저 내부 렉시컬 환경을 검색 범위**로 잡고, 내부 렉시컬 환경에서 원하는 변수를 찾지 못하면 **검색 범위를 내부 렉시컬 환경이 참조하는 외부 렉시컬 환경으로 확장**함
  - **확장 과정은 검색 범위가 전역 렉시컬 환경으로 확장될 때까지 반복**되며 전역 렉시컬 환경에서도 변수를 찾지 못하면,
    - 엄격 모드에선 **에러가 발생**함
    - 비엄격 모드에선 정의되지 않은 변수에 값을 할당하려고 하면 **에러가 발생하는 대신 새로운 전역 변수가 생성**됨
      - 이러한 코드는 문제가 될 소지가 많지만, 이전 코드가 실행되지 않는 불상사를 막기 위해서(하위 호환성을 위해서) 남아있는 기능임
  - 이러한 탐색 과정을 **"범위 체인(scope chain)"을 따라가며 검색이 이루어진다고 표현**하기도 함

엄격 모드에서 에러 발생

```
// 엄격 모드
"use strict"

function oops() {
    // notExist 변수 탐색 불가 => 에러 발생
    // Uncaught ReferenceError: notExist is not defined
    notExist = "?";
}

oops();
```

비엄격 모드에서는 전역 변수가 생성됨

```
function oops() {
    // 아마도 var 혹은 let 키워드를 실수로 인해서 생략한 코드
    notExist = "?";
}

oops();
// 실수로 인해서 전역 변수가 생김
alert(window.notExist);
```

Q1) func2 함수가 호출되는 시점의 렉시컬 환경 그려보기

```
let x = 100;
function func1(y) {
    function func2(z) {
        // alert 함수를 호출하는 시점에서 렉시컬 환경을 그려보기
        alert(`${x},${y},${z}`);
    }
    func2(300);
}
func1(200);
```

func2 함수 호출 시점의 렉시컬 환경

```
[ func2 함수 호출 시점의 렉시컬 환경 ]
z: 300
outer => [ 외부 렉시컬 환경 (func1) ]
         y: 200
         func2: function
         outer => [ 외부 렉시컬 환경 (전역) ]
                  x: 100
                  func1: function
                  outer => null
```

Q2) func2에서 x, y 값에도 접근할 수 있는 이유는?

Q3) 코드보고 질문 답변하기 (힌트 => 임의의 중괄호 코드 블록도 렉시컬 환경을 생성함)

```
let x = 100;
{
    let y = 200;
    {
        let z = 300;
        console.log(x);
        console.log(y);
        console.log(z);
    }
    console.log(x);
    console.log(y);
    // ReferenceError 발생
    // console.log(z);
}
console.log(x);
// ReferenceError 발생
// console.log(y);
// console.log(z);
```

1. 가장 많이 중첩된 중괄호 블록 내부에서 x, y, z 변수에 모두 접근한 가능한 이유는? => 가급적 스코프 체인, 외부 렉시컬 환경, 전역 렉시컬 환경같은 단어가 포함되도록 답변해보기
2. ReferenceError 에러가 발생하는 이유는?

단계 4. 반환 함수

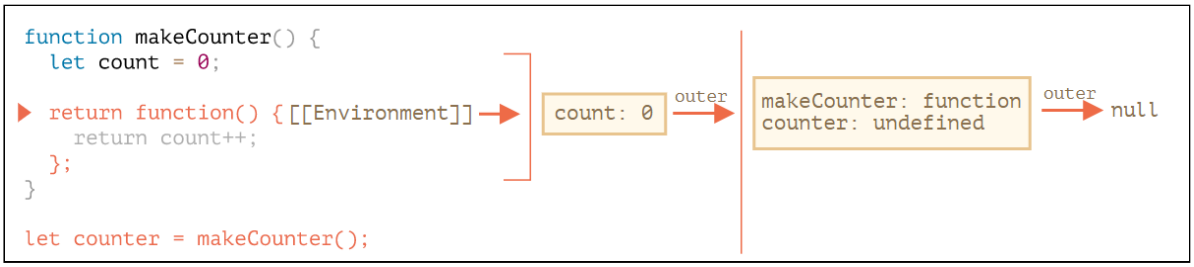
- 함수(여기서는 makeCounter)를 호출하면 **호출할 때마다 (새로운, 독립된 해당 함수 호출과 관련된) 렉시컬 환경 객체가 생성**됨
    - 렉시컬 환경엔 함수를 실행하는데 필요한 변수들(여기서는 count 변수 하나만 포함된 렉시컬 환경이 생김)이 저장됨

```
function makeCounter() {
    let count = 0;

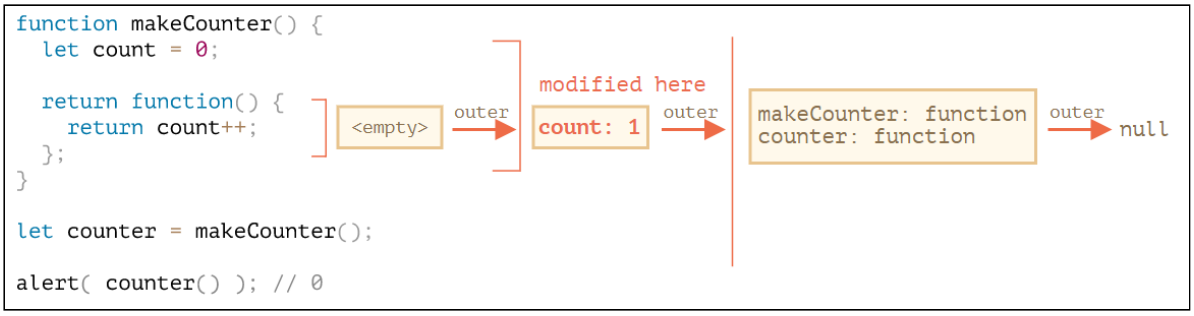
    // 이 익명 함수는 생성되며 [[Environment]]에 함수가 생성된 시점의 외부 렉시컬 환경
(makeCounter의 렉시컬 환경)을 저장함
    return function() {
        return count++;
    };
}

let counter = makeCounter();
```

- **모든 함수는 함수가 생성된 곳의 렉시컬 환경을 기억함**
    - 함수는 **[[Environment]]라고 불리는 숨김 프로퍼티**를 가지며, 여기에 **함수가 만들어진 곳의 렉시컬 환경에 대한 참조가 저장**됨
        - 즉, "익명함수.[[Environment]]"엔 해당 익명 함수가 생성된 (count 값이 존재하는) 렉시컬 환경에 대한 참조가 저장됨
            - "makeCounter.[[Environment]]"엔 전역 렉시컬 환경이 저장됨 (왜냐하면 makeCounter는 전역 환경에서 정의된 함수이므로)
        - 호출 장소와 상관없이 함수가 **자신이 생성된 곳을 기억**할 수 있는 건 바로 [[Environment]] 프로퍼티 덕분이며 **[[Environment]]는 함수가 생성될 때 딱 한 번 그 값이 설정되고 영원히 변하지 않음**
    - 자바스크립트의 함수는 숨김 프로퍼티인 [[Environment]]를 이용해 자신이 어디서 만들어졌는지를 기억
        - 함수 내부의 코드는 [[Environment]]를 통해서 함수가 생성된 시점의 외부 변수에 접근 가능
            - 익명 함수의 outer => "익명함수.[[Environment]]" => makeCounter의 렉시컬 환경
            - makeCounter 함수의 outer => "makeCounter.[[Environment]]" => 글로벌 렉시컬 환경



- counter 변수를 통해서 함수를 호출하면, 호출할 때마다 새로운 함수 호출 렉시컬 환경이 만들어지고 이 렉시컬 환경은 "익명함수.[[Environment]]"에 저장된 렉시컬 환경을 외부 렉시컬 환경으로서 참조함
    - 여기서 함수 호출 시 전달받은 인수나 선언한 지역 변수가 없으므로 렉시컬 환경은 비어있고, 결과적으로 **count 값은 "익명함수.[[Environment]]"에 저장된 렉시컬 환경에서 찾게 됨**



- 변숫값(count)의 갱신은 **변수가 저장된 렉시컬 환경에서 이루어짐**을 유의

Q) a2 함수 호출 시점에 함수 호출 렉시컬 환경 및 outer("익명함수.[[Environment]]")에 저장된 렉시컬 환경을 그려보기

```
function makeAppender(init = "", delimiter = "") {
    let str = init;

    return function(s) {
        if(delimiter && str.length === 0) {
            str += s;
        } else {
            str += (delimiter + s);
        }
        return str;
    }
}

let a1 = makeAppender();
console.log( a1("Hello") ); // "Hello"
console.log( a1("World") ); // "HelloWorld"

let a2 = makeAppender("010", "-");

console.log( a2("1234") ); // "010-1234"
// 다음 코드를 통해서 함수가 호출 실행되는 시점의 렉시컬 환경 그려보기
// ▼▼▼
console.log( a2("5678") ); // "010-1234-5678"
// ▲▲▲

let csv = makeAppender("김철수", ",");
csv("남자");
csv("20");
let ret = csv("대학생");
console.log( ret ); // "김철수,남자,20,대학생"
```

a2 함수 호출 시점의 렉시컬 환경

```
[ a2 함수 호출 시점의 렉시컬 환경 ]
s: "5678"
outer => [ 외부 렉시컬 환경 (makeAppender 함수 호출 렉시컬 환경) ]
         init: "010"
         delimiter = "-"
         str = "010-1234"
         outer => [ 외부 렉시컬 환경 (전역 렉시컬 환경) ]
                  a1: func
                  a2: func
                  makeAppender: func
                  csv: <uninitialized>
                  ret: <uninitialized>
                  outer => null
```

클로저(closure)

- **클로저(혹은 클로저 함수) 정의 => 외부 변수를 기억하고 이 외부 변수에 접근할 수 있는 함수**
  - https://poiemaweb.com/js-closure
- 위의 정의에 따르면 **자바스크립트에선 모든 함수가 자연스럽게 클로저**가 됨
  - 클로저 => 함수 + 함수가 참조하는 외부 렉시컬 환경
  - 왜냐하면 모든 함수가 생성되는 시점에 외부 렉시컬 환경을 저장하는 **[[Environment]]가 생성되고, 이를 통해 외부 렉시컬 환경을 참조**할 수 있기 때문
    - 결과적으로 이로 인하여 **외부 변수에도 접근**할 수 있으므로 모든 함수는 클로저의 특성을 지니게 됨

```
프론트엔드 개발자 채용 인터뷰에서 "클로저가 무엇입니까?"라는 질문을 받으면, 클로저의 정의를 말하고 자바스크립트에서 왜 모든 함수가 클로저인지에 관해 설명하면 될 것 같습니다. 이때 [[Environment]] 프로퍼티와 렉시컬 환경이 어떤 방식으로 동작하는지에 대한 설명을 덧붙이면 좋습니다.

Q) 클로저가 무엇입니까?
A) 클로저는 함수가 호출된 시점의 외부 환경을 기억하는 특수한 함수인데, 자바스크립트에서는 함수가 생성되는 시점에 함수의 [[Environment]] 숨김 프로퍼티가 설정되고, 해당 프로퍼티를 통해서 함수가 정의된 외부 렉시컬 환경을 참조할 수 있으므로, 엄밀히 말하면 모든 함수가 클로저라고 할 수 있습니다.
```

가비지 컬렉션

- 기본적으로 함수 호출이 끝나면 함수 호출 스택 정리 과정에서 **함수에 대응하는 렉시컬 환경이 메모리에서 제거**됨
  - 함수와 관련된 인수, 지역 변수들은 이때 모두 사라짐
  - 자바스크립트에서 모든 객체는 **도달 가능한 상태일 때만 메모리에 유지**되므로 함수 호출이 끝나면 관련 변수를 참조할 수 없게 됨
- 만약 함수가 반환되는 과정에서 **반환된 함수가 외부 변수를 참조할 경우 [[Environment]] 프로퍼티에 외부 렉시컬 환경에 대한 정보가 저장되므로 도달 가능한 상태**가 되어 가비지 컬렉션 대상이 되지 않게 됨

```
function f() {
    let value = 123;

    return function() {
        // 여기서 value를 참조하므로 [[Enviroment]]에 value에 대한 참조가 저장됨
        alert(value);
    }
}

// g 변수가 참조를 잃지 않는 동안엔 연관 렉시컬 환경인 g.[[Enviroment]]도 메모리에 value 값이 계속 존재
let g = f();
// null 대입으로 인해서 g.[[Enviroment]]에 도달할 수 없는 상황이 되므로, 이 시점 이후 메모리에서 value 값이 삭제됨
g = null;
```

외부 변수를 참조하는 함수를 반환하지 않는 코드

```
function sum(a, b) {
    // 함수 호출에 의해서 새 렉시컬 환경이 생성되고 환경에 a, b, c 값이 저장됨
    let c = 100;
    return a + b + c;
}

// 함수 호출 이후 지역 변수인 a, b, c에는 도달할 수 없으므로 메모리에서 삭제
let result = sum(1, 2);
```

추가 예시

Q1) 콘솔 출력 결과는?

```
let a = 100;
function func1() {
    let b = 200;
    function func2() {
        let c = 300;
        return function(amount = 1) {
            a += amount; b += amount; c += amount;
            console.log(a, b, c);
        }
    }
    return func2();
}

let f1 = func1();
f1();
f1();
f1();

let f2 = func1();
f2(2);
f2(3);
f2(4);
```

Q2) 콘솔 출력 결과는?

```
let a = 100;
function func1() {
    let b = 200;
    function func2() {
        return function() {
            let c = 300;
            a++; b++; c++;
            console.log(a, b, c);
        }
    }
    return [func2(), func2()];
}

let fArr = func1();
fArr[0]();
fArr[1]();
```

오래된 var (@)

- var로 선언한 변수는 let으로 선언한 변수와 유사하게 동작함
    - 즉, 대부분의 경우 let을 var로, var를 let으로 바꿔도 큰 문제 없이 동작함

var는 블록 스코프가 없습니다

- var로 선언한 변수의 스코프는 **함수 스코프이거나 전역 스코프**
    - 블록 기준으로 스코프가 생기지 않기 때문에 **블록 밖에서 값에 접근이 가능**

```
// 조건문 관련 블록 스코프 생성
if (true) {
    // let 대신 var를 사용
    var test = true;
}

// 블록 밖에서 값에 접근 가능
alert(test); // true
```

- var는 코드 블록을 무시하기 때문에 test는 전역 변수가 되고, 따라서 전역 스코프에서 이 변수에 접근하게 됨
- **반복문**에서도 블록 스코프를 생성하지 않으므로 다음과 같은 문제 발생

```
// let 대신 var를 사용
for (var i = 0; i < 10; i++) {
    // ...
}

// 반복문이 종료되었지만 'i'는 전역 변수이므로 여전히 접근 가능
alert(i); // 10
```

- 단, 코드 블록이 함수 안에 있다면, var는 (함수 내부에서만 접근 가능한) **함수 레벨 변수**가 되어 외부에서 접근은 불가
    - 즉, 예외적으로 함수 호출 과정에서 생긴 **함수 수준의 블록 스코프는 존중**하는 것을 알 수 있음

```
// 함수 레벨에서 블록 스코프 작성
function sayHi() {
    // 여전히 블록 스코프를 무시
    if (true) {
        var phrase = "Hello";
    }

    // 제대로 출력됨
    alert(phrase);
}

sayHi();

// 단, 외부(함수 바깥)에서는 접근 불가
// Error: phrase is not defined
alert(phrase);
```

var로 변수를 여러번 선언해도 무방합니다

- let으로 두 번 똑같은 식별자를 가진 변수를 선언할 경우에는 다음과 같이 에러가 나지만

```
let user;
// SyntaxError: 'user' has already been declared
let user;
```

- var를 사용해서 선언하면 아무런 문제도 생기지 않음

```
var user = "Pete";
// 여기서 var는 그냥 무시됨 (즉, 변수 선언이 아닌, 단순한 대입문처럼 처리됨)
// 또한, 에러를 발생시키지도 않음
var user = "John";

alert(user); // John
```

선언하기 전 사용할 수 있는 var

- var 변수 선언은 **함수가 시작될 때 처리**됨 (호이스팅되어 undefined 값으로 설정)
    - 전역에서 선언한 변수라면 **스크립트가 시작될 때 처리**
- 함수 본문 내에서 var로 선언한 변수는 선언 위치와 상관없이 함수 본문이 시작되는 지점에서 정의

```
function sayHi() {
    phrase = "Hello";
    alert(phrase);
    var phrase;
}
sayHi();
```

위의 코드와 똑같이 작동하는 코드

```
function sayHi() {
    var phrase;
    phrase = "Hello";
    alert(phrase);
}
sayHi();
```

- 코드 블록은 무시되기 때문에, 아래 코드 역시 동일하게 동작

```
function sayHi() {
    // (*)
    phrase = "Hello";
    if (false) {
        // 이 코드가 (*)의 위치로 끌어 올려진다고 생각하기
        var phrase;
    }
    alert(phrase);
}
sayHi();
```

- 이렇게 **변수가 끌어올려 지는 현상을 호이스팅(hoisting)이라고 부름**
    - var로 선언한 **모든 변수는 함수의 최상위로 호이스팅 됨**(=최상위로 끌어 올려짐)
- 단, **선언은 호이스팅 되지만 할당은 호이스팅 되지 않음**을 유의

```
function sayHi() {
    alert(phrase);
    var phrase = "Hello";
}
sayHi();

// 위의 코드가 실제로 실행되는 방식은 아래와 같음
function sayHi() {
    // 선언은 함수 시작 시 처리 (선언은 호이스팅됨)
    var phrase;
    // undefined
    alert(phrase);
    // 할당은 실행 흐름이 해당 코드에 도달했을 때 처리 (할당은 호이스팅되지 않음)
    phrase = "Hello";
}
sayHi();
```

- 모든 var 선언은 함수 시작 시 처리되기 때문에, **var로 선언한 변수는 어디서든 참조 가능**
    - 그러나 변수에 무언가를 할당하기 전까진 값이 undefined임을 유의

즉시 실행 함수 표현식 (@)

- var도 블록 레벨 스코프를 가질 수 있게 여러가지 방안을 고려
    - **즉시 실행 함수 표현식(immediately-invoked function expressions, 줄여서 IIFE)**을 사용

IIFE 코드

```
// 함수 레벨에서는 외부로 영향을 주지 않는 스코프가 생긴다는 특징을 이용
(function() {
    // 함수 호출
    let message = "Hello";
    alert(message); // Hello
})();
```

- 자바스크립트는 "function"이라는 키워드를 만나면 **함수 선언문이 시작될 것이라 예상**함
    - 그런데, 함수 선언문으로 함수를 만들 땐 반드시 함수의 이름이 있어야 하므로 아래와 같이 익명 함수를 선언과 함께 실행하면 에러가 발생

```
// 함수를 선언과 동시에 실행하려고 함
function() { // <-- Error: Function statements require a function name
    let message = "Hello";
    alert(message); // Hello
}();
```

- 따라서 **소괄호로 익명 함수 정의 부분을 감싸서** 해당 익명 함수가 표현식으로 평가되어 함수를 반환하고 난 뒤에 함수 실행이 되도록 위의 IIFE 코드와 같이 써주어야 함

```
// step 1) 익명 함수를 표현식으로 평가
(function() { /* ... */ })();

// step 2) 표현식 평가가 끝나서 함수가 반환(f)되고 이어서 함수가 즉시 호출됨
// (function() { /* ... */ }) => f
f();
```

IIFE 미사용 코드 (전역 공간 오염)

```
// 주로 전역 공간을 오염시키지 않기 위해서 사용됨
var multiplier = 2;
var a = 10 * multiplier;
var b = 20 * multiplier;
var c = a + b;
console.log(c); // 60
// a, b, c 모두 전역 객체에 정의됨
console.log(window.multiplier, window.a, window.b, window.c);
```

IIFE 사용 코드

```
// (가급적, 내부 블록을 통해 전역 공간 오염 없이 연산을 처리하고 결과만 가져오자는 전략으로 이해)
var result = (function(multiplier) {
    var a = 10 * multiplier;
    var b = 20 * multiplier;
    return a + b;
})(/* multiplier */ 2);

// 물론, 이렇게 해도 전역 공간에 result가 생기긴 함
console.log(window.result);
```

객체를 반환하는 IIFE 사용 코드

```
var result = (function() {
    var 어쩌구 = "Hello";
    var 저쩌구 = "World";

    // 뭔가 프로그램 내부에서 사용할 객체 초기화 관련 작업 진행
    // ...

    return {
        a: 100,
        b: 어쩌구
    }
})();

// 물론, 이렇게 해도 전역 공간에 result가 생기긴 함
console.log(window.result);
```