스코프 체인(Scope Chain)

실행 컨텍스트를 자바스크립트 객체로 표현

```
ExecutionContext = {
  LexicalEnvironment: {
    EnvironmentRecord: {
    },
    outerLexicalEnvironment: null
  }
}
```

스코프 체인(Scope Chain)

실행 컨텍스트(Execution Context)

- 코드가 실행되기 위해 필요한 정보를 가지고 있다.
- Execution Context = LexicalEnvironment + EnvironmentRecord + OuterLexicalEnvironment
  실행 컨텍스트 렉시컬 환경 환경 레코드 외부 렉시컬 환경
- ex) 전역 코드, 함수코드, eval, module 코드
- 전역코드 -> 전역 컨텍스트 -> 전역코드 -> 실행 컨텍스트 -> 함수내용 실행 -> 스택에 쌓임 -> 실행종료 -> 스택에서 제거

렉시컬 환경(LexicalEnvironment)

- 환경 레코드(EnvironmentRecord) + 외부 렉시컬 환경(OuterLexicalEnvironment)

환경레코드(EnvironmentRecord)

- EnvironmentRecord = { key : value }

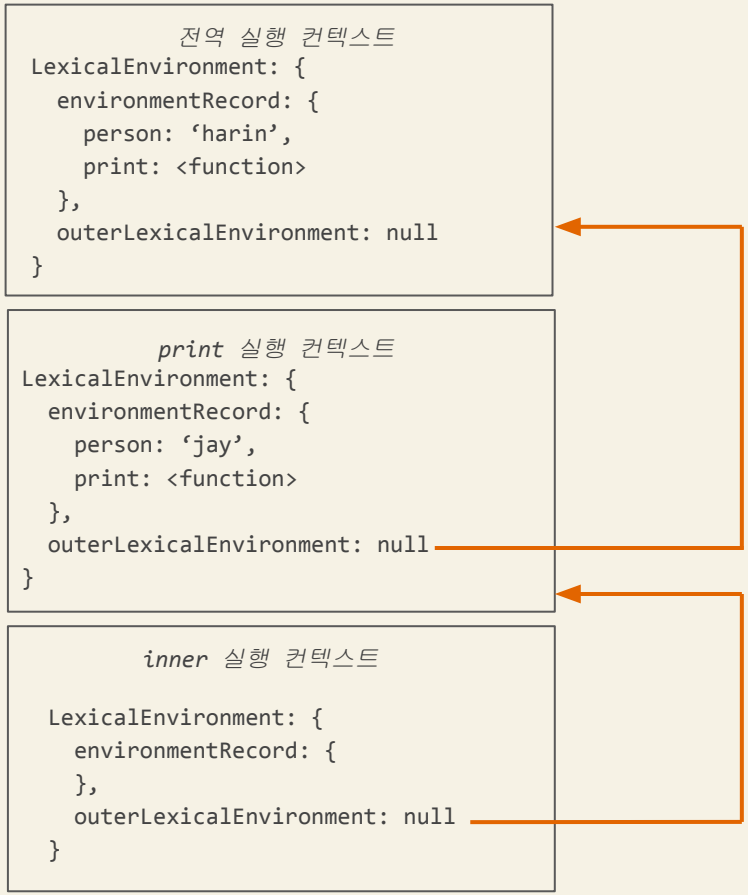
스코프 체인(Scope Chain)

```
var person = "harin";

function print(){
      var person2 = "jay";

      function innerPrint(){
        console.log(person);
        console.log(person2);
      }

      innerPrint();
      console.log("print finished");
}

print();
console.log("finished");
```

- innerPrint() 함수가 호출될 때 두 변수 person과 person2를 자신의 실행 컨텍스트의 렉시컬 환경에서 찾는다.
- 하지만 자신의 렉시컬 환경에 없기 때문에 innerPrint()함수의 렉시컬 환경 레코드에는 아무런 키-값도 존재하지 않는다.
- 이렇게 자신의 실행 컨텍스트에 없으면 외부 렉시컬 환경의 참조를 통해 연결된 print 실행 컨텍스트에서 해당 식별자를 찾게된다.
- 이 때 person2을 print 실행 컨텍스트의 환경 레코드에서 찾아서 'jay'를 출력하게 된다. 마찬가지로 person는 전역 실행 컨텍스트 까지 가서 찾아 'harin'값을 출력한다.

스코프 체인(Scope Chain)

스코프 체인(Scope Chain)



클로저(Closure)

함수가 실행된 이후에도 메모리에 값이 존재해 외부에서 접근하여 사용할 수 있는 방법

- 내가 정의한 내용 -

함수가 특정 스코프에 접근할 수 있도록 의도적으로 그 스코프에서 정의하는 방법,
이를 보통 클로저(Closure)라고 한다.
스코프를 함수 주변으로 좁히는(closing) 것이라고 생각해도 된다.

- Learning JavaScript -

함수와 함수가 선언된 어휘적 환경(Lexcial Environment)과의 조합이다.

- MDN developer.mozilla.org -

클로저(Closure)

함수를 실행한 이후에도 메모리에 존재해서 외부에서 참조해서 사용할 수 있는 방법

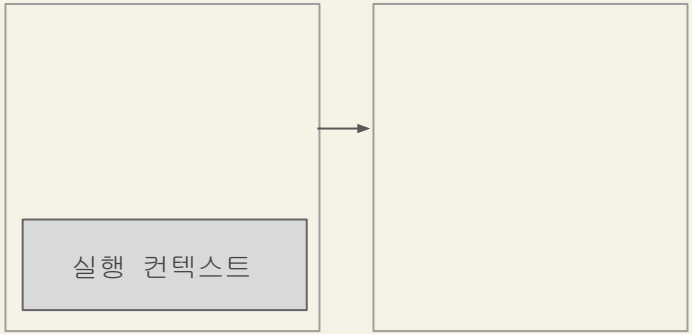
함수를 호출하고 나면 대체로 실행 컨텍스트가 실행되는 동안 컨텍스트 스택에 존재하다가 사라진다.

그런데 클로저(closure) 형태로 만들어 지면 실행이 되고 난 이후에도 실행 컨텍스트가 존재해 참조할 수 있다.

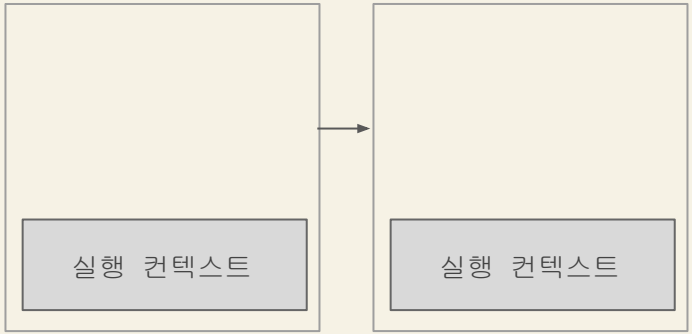
함수안에 내부함수를 선언해서 내부함수내에 있는 주변에 있는 변수에 접근하는 방법

함수와 함수가 선언된 어휘적 환경(Lexical Environment)과의 조합이다.

일반적 함수 호출



클로저(Closure) 함수 호출



클로저(Closure)

클로저(Closure) 형태의 선언 방법

1. 함수 안의 함수(중첩 함수)

```
function outerFn(){
      let x = 10;

      // closure
      return function innerFn(y){
        return x = x + y;
      }
}
```

2. 전역에 선언한 변수를 블록 안에서 함수로 정의하고 전역에서 호출

```
let globalFunc;
{
      let x = 10;

      // closure
      globalFunc = function(y){
        return x = x + y;
      }
}
globalFunc();
```

클로저(Closure)

클로저(Closure) 형태를 사용하는 이유(용도)

- 일반적으로 접근할 수 없는 것에 접근하는 효과도 있다.
- 자바스크립트의 모든 함수는 클로저(closure)를 정의한다.
- 변수를 은닉하여 지속성을 보장

클로저(Closure)

```
function createCounterClosure(){
    let count = 0;
    return {
        increase: function(){
            count++;
        },
        getCount: function(){
            return count;
        }
    }
}

const counter1 = createCounterClosure();
const counter2 = createCounterClosure();

counter1.increase();
counter1.increase();
console.log('counter 1의 값:'+ counter1.getCount());

counter2.increase();
console.log('counter 2의 값:'+ counter2.getCount());
```

count1과 count2의 메소드들이 다른 count에 접근하는 것은 서로 다른 렉시컬 환경의 환경 레코드에서 count에 접근하기 때문이다.
또한, 이러한 현상이 가능한 이유는 바로 클로저(closure) 때문이다.

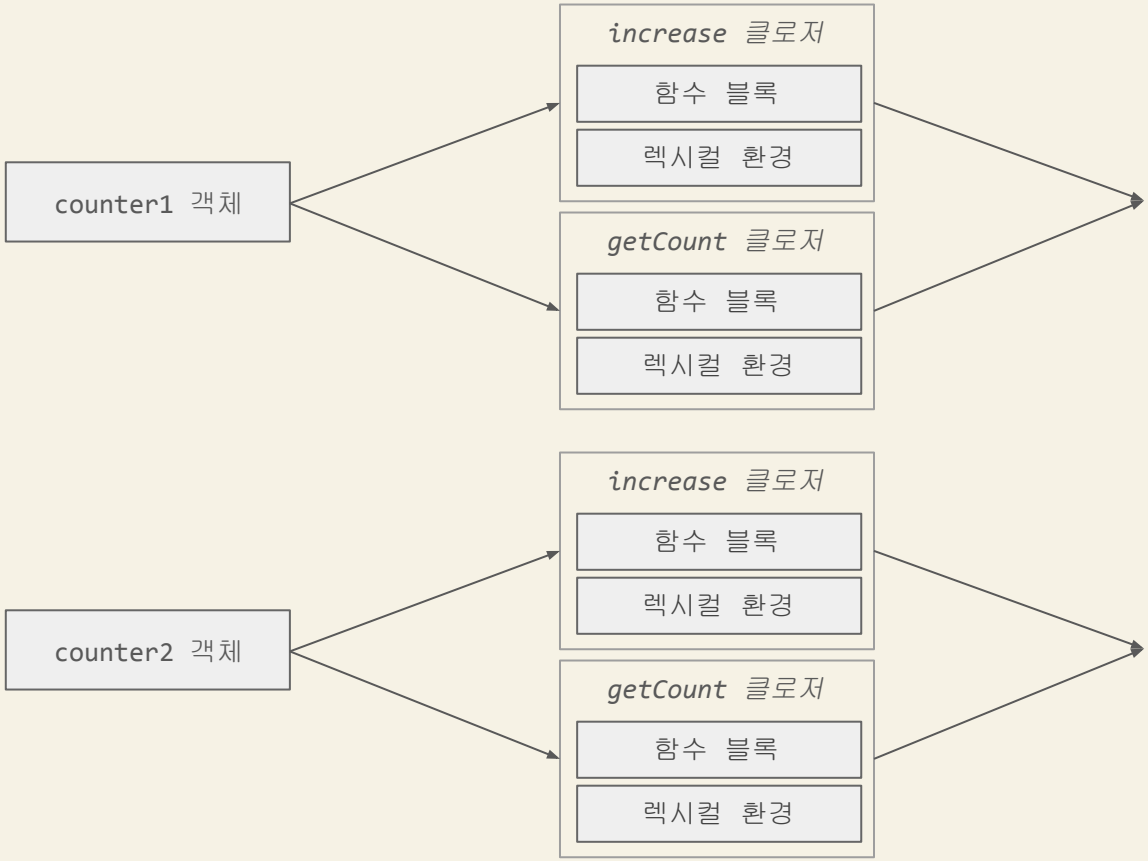
increase와 getCount 함수가 정의될 때의 렉시컬 환경은 createCounterClosure 실행 컨텍스트의 렉시컬 환경이다.
이 실행 컨텍스트는
`const counter1 = createCounterClosure();` 이나
`const counter2 = createCounterClosure();`
를 호출 할 때 만들어 진다.

그래서 increase 함수와 getCount 함수는 createCounterClosure
실행 컨텍스트의 렉시컬 환경을 기억하고 있는 클로저가 된다.

대체로 실행 컨텍스트가 컨텍스트 스택(Stack)에서 제거되면 해당 환경은 사라지기 마련이다.
그런데 클로저가 만들어지면 해당 환경(실행 컨텍스트)은 사라지 않는다.

왜냐하면 해당 참조가 존재하기 때문이다.

클로저(Closure)


counter1 객체
increase 클로저
함수 블록
렉시컬 환경
getCount 클로저
함수 블록
렉시컬 환경
createdCounterClosure 실행 컨텍스트
LexicalEnvironment: {
environmentRecord: {
count: 2
}
}
counter2 객체
increase 클로저
함수 블록
렉시컬 환경
getCount 클로저
함수 블록
렉시컬 환경
createdCounterClosure 실행 컨텍스트
LexicalEnvironment: {
environmentRecord: {
count: 1
}
}