클로저

클로저란?

클로저를 MDN에서는 **“클로저는 함수와 그 함수가 선언됐을 때의 렉시컬 환경과의 조합”** 으로 정의합니다. 물론 전혀 이해가 되지는 않죠. 그럼 이제 이해가 가게 해 봅시다.

클로저를 이해하기 위한 선수 지식은 **Scope(그 중에서도 Lexical Scope)** 와 **실행 컨텍스트** 입니다. 여기부터 이해를 해 봅시다.

Scope

기본적으로 변수를 생성하시면 전역 변수, 지역 변수 둘 중 하나로 나뉘게 된다는 것은 알고 있으실 겁니다. 예시를 들어보자면 아래와 같습니다.

```
var x = 'global value'

function example() {
  var x = 'local value'
  x = 'value changed'
}

example()
alert(x) // 'global value'
```

전역 변수는 **어디서든 쓸 수 있는 변수**, 지역 변수는 **해당 변수가 선언된 지역에서만 사용할 수 있는 변수**라는 뜻입니다. 그렇다면 여기서 example 함수 바깥의 변수 x가 **전역(global) 변수**, example 함수 내부의 변수 x를 **지역(local) 변수**가 됩니다.

위의 예제의 example 함수 안에서 `x = 'value changed'` 로 x 변수를 바꿔줬지만 이는 example 함수 내부의 지역 변수 x를 바꾼 것이지 전역 변수 x를 바꾼 것이 아닙니다. 그리고 지역변수 x는 example 함수 안에서만 사용이 가능하기 때문에 그 외부에서 `alert(x)` 를 하면 global value가 출력되는 것입니다.

이제까지 전역 변수, 지역 변수를 얘기 했고 그 동안 **~~에서 쓸 수 있는 변수다** 라고 얘기했습니다. 그리고 Scope란 그 범위를 의미합니다. 전역 변수의 범위(Scope)는 전체 즉, window 객체가 되는 것이고, 지역 변수의 범위는 해당 지역이 되는 것입니다.(위 예제에서는 example 함수)

Scope Chain

Chain 이라고 하면 '사슬, 연쇄' 라는 뜻을 가집니다. 그럼 Scope Chain 을 직역하면 '범위 사슬/연쇄' 이정도가 되겠네요. 전 연쇄작용보다는 사슬이라고 보는게 이해는 더 쉬웠습니다. 예시를 한 번 보겠습니다.

```
var name = 'zero';

function outer() {
  console.log('외부', name);

  function inner() {
    var enemy = 'nero';
    console.log('내부', name);
  }

  inner();
}

outer();
console.log(enemy); // undefined
```

outer 함수가 실행된 이후 enemy를 콘솔에 찍었고 이는 undefined 라는 결과를 내놓았습니다. 그리고 outer 함수가 실행되면서 inner 함수 안의 콘솔에서는 **'내부'zero**, outer 함수 안의 콘솔에서는 **'외부'zero** 라는 결과를 내놓았습니다.

outer 함수부터 보겠습니다. `console.log('외부', name)` 에서 name을 찾는 과정을 보면 먼저 자기 자신의 scope에서부터 name 변수를 찾습니다. 여기선 outer 함수 안에서 찾겠죠. 하지만 찾지 못했고 그 한단계 위의 전역 scope에서 name 변수를 찾아 이를 출력합니다.

inner 함수 역시 마찬가지로 inner 함수 scope에서부터 name을 찾아보고 없으면 outer 함수로, 여기에도 없으면 전역 scope 까지 올라가 name을 찾고 이를 출력합니다.

그리고 이렇게 scope를 거슬러 올라가면서 무엇인가를 찾는 것을 **Scope Chain** 이라고 합니다.

Lexical Scoping

그렇다면 이런 Scope는 언제 생길까요? 그냥 있으면 있는거지 뭘 언제 생기는지까지 찾고 있냐 라고 하시면 안됩니다. 개발에서 모든 생기는 순서는 실제 동작 순서를 결정하기 때문에 아무리 코드를 위에서 아래로 차근차근 실행한다고는 하지만 이와 다르게 동작도 많이 합니다.

다시 본론으로 들어와서!! 결론적으로 Scope는 함수를 **호출할 때가 아니라 함수를 선언할 때** 생깁니다. 예시를 보겠습니다.

```
var name = 'zero';
function log() {
  console.log(name);
}

function wrapper() {
  name = 'nero';
  log();
}
wrapper();
```

결과는 **nero** 입니다.

log 함수가 **선언되는 시점**에 log함수의 scope는 **[log 함수, 전역]** 으로 정해졌고, wrapper 함수가 실행되면서 전역 변수 name을 nero로 재할당 했으니 그 후에 실행되는 log 함수는 log 함수 내에서 name을 찾고, 못찾았기 때문에 다음으로 전역 환경에서 name을 찾아 콘솔에 nero를 찍게 되는 것입니다.

다른 예시를 보겠습니다.

```
var name = 'zero';
function log() {
  console.log(name);
}

function wrapper() {
  var name = 'nero';
  log();
}
wrapper();
```

이번에 결과값은? **zero** 입니다.

log 함수가 wrapper 안에서 **실행(호출)**됐는데 그럼 scope가 [log 함수, wrapper 함수, 전역] 이렇게 돼야 하는거 아닌가요? 라고 하실 수도 있지만 위에서도 말했다시피 scope는 함수 선언 시점에 결정되기 때문에 log 함수의 scope는 **[log 함수, 전역]** 이 되고 log 함수 내부에서 name 변수를 찾을 수 없었기 때문에 그 다음 scope인 전역 환경으로 가 name을 찾아 출력하게 되는 것입니다.

그리고 이렇게 **Scope는 선언 시점에 결정된다** 라는 개념이 **Lexical Scoping** 입니다.

실행 컨텍스트

여기까지 해서 Closure를 이해하기 위한 중요한 2가지 개념 중 하나인 Lexical Scoping 까지 공부했습니다. 그렇다면 이제 실행 컨텍스트를 공부할 차례입니다.

Context를 번역하면 문맥이라고 합니다. 그리고 문맥은 **문장과 문장과의 관계**를 의미합니다. 하지만 개발자 입장에서 Context는 환경 이라고 생각하는 것이 더 편합니다. 즉, 실행 컨텍스트는 실행 환경이 되는 것이죠. 간단한 예시를 보겠습니다.

```
var name = 'zero';
function wow(word) {
  console.log(word + ' ' + name);
}

function say () {
  var name = 'nero';
  console.log(name);
  wow('hello');
}

say();
```

위 코드에서 결과는 어떻게 될까요? say 함수 안에서 **nero**가 출력되고 wow 함수 안에서 **hello zero** 가 출력되는 것을 알 수 있습니다. 그럼 위의 코드들이 어떻게 실행되는지 내부에서의 관점에서 살펴보겠습니다.

처음 코드가 실행되는 순간 모든 것을 포함하는 **전역 컨텍스트**, 즉 전역 환경이 생깁니다. 그 안에는 갖가지 모든 내용들이 들어 있고, 이런 전역 컨텍스트는 **페이지가 종료될 때까지 유지**됩니다. 그리고 함수가 **호출**될 때마다 **함수 컨텍스트**가 생깁니다. 여기서 중요한건 **호출** 입니다. Scope는 선언 시점에서 결정되지만 함수 컨텍스트는 함수가 호출될 때 생겨납니다. 그리고 함수가 종료되면 해당 컨텍스트도 사라지게 됩니다. (물론 예외가 있고 그 예외는 **클로저** 입니다.)

이런 컨텍스트에는 4가지 원칙이 있습니다.

1. 전역 컨텍스트 하나 생성 후, 함수 호출 시마다 컨텍스트가 생깁니다.
2. 컨텍스트 안에는 **변수객체(arguments, variable), scope chain, this** 가 있습니다.
3. 컨텍스트 생성 후 함수가 실행되는데, 사용되는 변수들은 변수 객체 안에서 값을 찾고, 없다면 scope chain을 따라 올라가며 찾습니다.

4. 함수 실행이 마무리되면 해당 컨텍스트는 사라집니다. 페이지가 종료되면 전역 컨텍스트가 종료됩니다.

이제까지 1, 3, 4번은 설명이 됐으니 이제 2번을 중심으로 위 코드를 실행해보겠습니다.

1. 전역 컨텍스트

가장 먼저 전역 컨텍스트가 실행됩니다. 그리고 2번 원칙에 따라 변수 객체, scope chain, this가 생성됩니다. 여기서 변수 객체 안의 **arguments는 함수의 인자, variable은 본인 scope 내부의 변수들** 입니다. 그렇게 되면 전역 컨텍스트의 arguments는 없을거고, variable은 **name, wow, say** 가 됩니다. 함수를 왜 변수로 놓냐! 라고 하실 수도 있지만 JavaScript는 유일하게 함수를 변수에 할당해줄 수 있는 언어입니다.

그리고 scope chain은 상위에 뭐가 없기 때문에 자기 자신만을 가지고, this는 따로 설정하지 않으면 window가 됩니다. this의 경우 bind를 하거나 new를 호출하지 않는 이상 반드시 window입니다.

전역 컨텍스트를 객체 형식으로 표현하면 아래와 같아집니다.

```
'전역 컨텍스트': {
  변수객체: {
    arguments: null,
    variable: ['name', 'wow', 'say'],
  },
  scopeChain: ['전역 변수객체'],
  this: window,
}
```

이렇게 전역 컨텍스트가 만들어지고 난 이후, 해당 컨텍스트의 코드들이 실행됩니다. wow와 say의 경우 호이스팅 때문에 선언과 동시에 할당이 일어나며, name은 ‘zero’가 할당되어 variable이 `variable: [{ name: 'zero' }, { wow: Function }, { say: Function }]` 와 같이 변합니다.

say 함수 컨텍스트

그렇게 코드가 실행되다가 say 함수가 호출되는 순간(`say()`) say 함수에 대한 함수 컨텍스트가 생성됩니다. 절대 선언하는 순간이 아닙니다!!

say 함수 컨텍스트는 arguments가 없고, variable은 name이 있습니다. scope chain은 선언시를 기준으로 만들어 지기 때문에 [say, 전역] 이 될 것이고, this는 그대로 window가 됩니다. 그리고 이걸 객체 형식으로 표현하면 아래와 같아집니다.

```
'say 컨텍스트': {
  변수객체: {
    arguments: null,
    variable: ['name'],
  },
  scopeChain: ['say 변수객체', '전역 변수객체'],
  this: window,
}
```

이렇게 함수 컨텍스트가 생기고 난 이후 say 함수 내부의 코드들이 실행됩니다. variable에 있는 name이라는 변수에 ‘nero’를 할당해주고, console.log(name)을 찍습니다. 이때, name은 say 컨텍스트 안에서 찾게 되는데 variable에 name이 있기 때문에 해당 내용을 그대로 출력해줍니다. 그리고 wow(’hello’) 라는 코드가 실행되면서 wow 함수가 호출되게 됩니다.

wow 함수 컨텍스트

wow 함수가 호출되었으니 이제 wow에 대한 함수 컨텍스트가 생깁니다. wow 함수 컨텍스트는 아래와 같이 객체로 표현 가능합니다.

```
'wow 컨텍스트': {
  변수객체: {
    arguments: [{ word : 'hello' }],
```

```
    variable: null,
  },
  scopeChain: ['wow 변수객체', '전역 변수객체'],
  this: window,
}
```

wow 컨텍스트 정도는 이제 해석이 가능하실 것 같습니다.

이렇게 모든 컨텍스트가 만들어지고 코드가 실행된 후, 해당 컨텍스트의 코드가 종료되면 컨텍스트가 사리집니다. 즉, 코드 실행 이후, wow → say 컨텍스트가 순서대로 사라지고, 페이지가 종료되면 전역 컨텍스트가 사라집니다.

클로저

예제를 먼저 보겠습니다.

```
function outerFunc() {
  var x = 10;
  var innerFunc = function() { console.log(x) };
  return innerFunc;
}

var inner = outerFunc();
inner();
```

실행 컨텍스트적으로 분석을 해보면 outerFunc는 innerFunc을 반환하고 실행 컨텍스트 스택에서 제거됩니다. 그리고 outerFunc이 사라졌으니 그 안의 변수 x또한 유효하지 않게 됩니다. 하지만 위 코드를 실행해보면 10이 출력되는 것을 확인할 수 있습니다.

이처럼 자신을 포함하고 있는 외부 함수보다 내부 함수가 더 오래 유지되는 경우, **외부 함수 밖에서 내부 함수가 호출되더라도 외부 함수의 지역 변수에 접근할 수 있는 함수를 클로저**라고 부릅니다.

다시 MDN의 정의로 돌아가보면 **“클로저는 함수와 그 함수가 선언됐을 때의 렉시컬 환경과의 조합”** 에서 **함수**란 **반환된 내부 함수**를 의미하고, **그 함수가 선언될 때의 렉시컬 환경**이란 **내부 함수가 선언되었을 때의 스코프**를 의미합니다.

즉, **클로저**는 **반환된 내부함수가 자신이 선언됐을 때의 환경인 스코프를 기억해 자신이 선언됐을 때의 환경(스코프) 밖에서 호출되어도 그 환경에 접근할 수 있는 함수** 를 의미합니다. 이를 간단히 말하면 **자신이 생성될 때의 환경을 기억하는 함수** 라고 말할 수도 있습니다.

클로저에 의해 참조되는 외부 함수의 변수 즉, 위의 예시에선 outerFunc 함수의 변수 x를 자유변수 라고 부릅니다.

실행 컨텍스트의 관점에서 보면, 내부함수가 유효한 상태에서 외부함수가 종료돼 외부 함수의 실행 컨텍스트가 반환되어도, 외부 함수 실행 컨텍스트 내의 변수 객체는 **내부 함수에 의해 참조되는 한 유효**해 내부함수가 스코프 체인을 통해 참조할 수 있는 것을 의미합니다.

즉, 외부 함수가 이미 반환되었어도 외부 함수 내의 변수는 이를 필요로 하는 내부함수가 하나 이상 존재하는 경우 계속 유지됩니다. 이때, 내부함수가 외부 함수에 있는 변수의 복사본이 아니라 실제 변수에 접근한다는 것만 주의하면 됩니다.

클로저의 활용

클로저는 렉시컬 환경을 기억하고 있어야 하므로 메모리 차원에서 손해를 볼 수 있지만, 항상 적극적인 사용이 권유됩니다.

```
<!DOCTYPE html>
<html>
<body>
  <button class="toggle">toggle</button>
  <div class="box" style="width: 100px; height: 100px; background: red;"></div>

  <script>
    var box = document.querySelector('.box');
```

```
    var toggleBtn = document.querySelector('.toggle');

    var toggle = (function () {
      var isShow = false;

      return function () {
        box.style.display = isShow ? 'block' : 'none';
        // ③ 상태 변경
        isShow = !isShow;
      };
    })();

    toggleBtn.onclick = toggle;
  </script>
</body>
</html>
```

위 예시에서 즉시 실행 함수는 함수를 반환하고 바로 소멸합니다. 하지만 해당 즉시 실행 함수가 반환한 함수는 자신이 생성됐을 때의 렉시컬 환경에 속한 변수 isShow를 기억하는 클로저 함수입니다. 클로저가 기억하는 변수 isShow는 box 요소의 표시 상태를 나타냅니다.

클로저가 제거되지 않는 한 클로저가 기억하는 렉시컬 환경의 변수 isShow는 소멸되지 않고, 이는 전역변수 사용 없이 상태 유지가 가능한 것이 됩니다. 전역 변수는 누구나 접근할 수 있고 변경할 수 있기 때문에 많은 부작용이 있어 위와 같이 클로저를 활용하는게 좋은 방법입니다.

위와 같이 코드가 짜여지면 isShow라는 변수는 toggleBtn의 onclick 함수를 제외하고는 어디서도 접근이 불가능한 private한 변수가 됩니다. 그리고 이는 정보의 은닉이 가능하다는 것입니다.

javascript에는 private이란 기능이 없지만 클로저를 활용하면 이를 흉내낼 수 있습니다.