

04

04-b. 클로저

JavaScript Object Oriented Programming

클로저

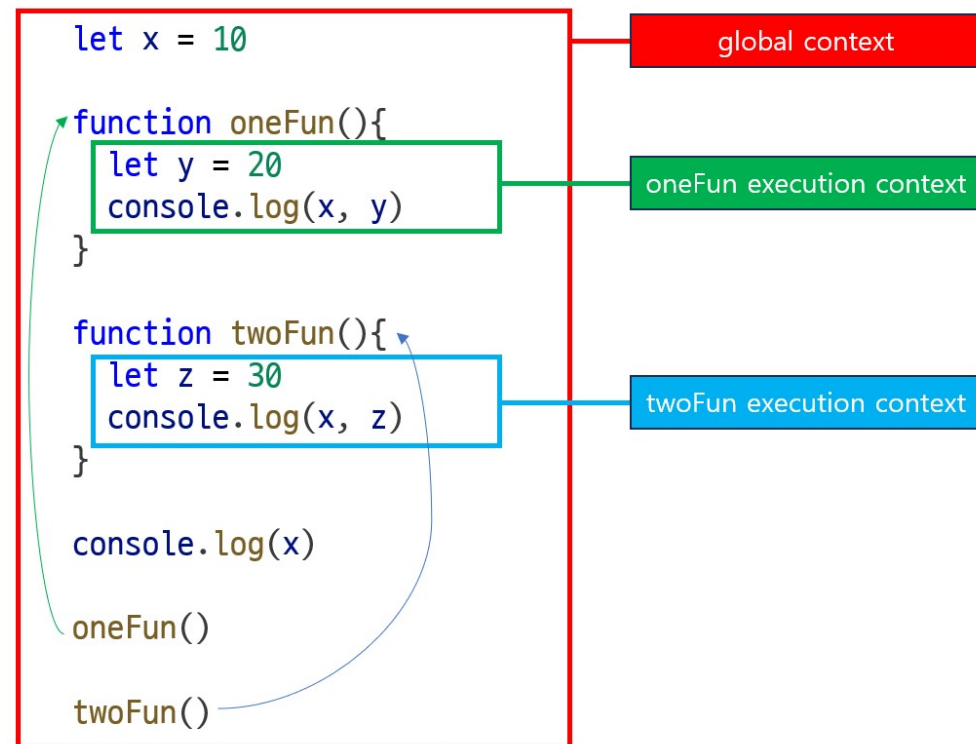
- 클로저(Closure)란 함수와 함수가 선언되었을 때의 렉시컬 환경(Lexical environment)의 조합을 의미합니다.
- 클로저는 자바스크립트에서 중요 개념이며 자바스크립트 뿐만 아니라 함수를 객체로 사용하는 대부분의 소프트웨어 언어에서 제공되는 개념입니다.
- 클로저는 자바스크립트에서 함수를 이용하기 위해서 자동으로 제공되는 개념이며 클로저를 위해 개발자가 어떤 코드적인 프로그램을 작성해야 하는 것은 아닙니다.
- 클로저 개념을 접하면 좀 복잡해 보이는데 실행 컨텍스트(Execution Context) 개념과 렉시컬 환경을 이해가 선행되어야 합니다.

클로저

실행 컨텍스트

- 실행 컨텍스트란 함수의 실행 환경이며 함수가 실행되기 위한 정보를 가지는 객체입니다.
- 함수가 호출되어 실행되려면 함수에 전달된 매개변수 값, 함수내에 선언된 로컬 변수, 로컬 함수등을 이용해야 합니다.
- 이 정보를 가지는 객체가 실행 컨텍스트입니다.
- 함수가 호출이 되면 자동으로 그 함수를 위한 실행 컨텍스트가 만들어지고 이 실행 컨텍스트내에 그 함수를 위한 매개변수 값, 로컬 변수등이 저장된다고 생각하면 됩니다.
- 그래서 함수내에서 매개변수를 참조하거나 로컬의 변수를 참조할 때 자동으로 함수의 실행 컨텍스트내에 등록된 것들이 이용되는 구조입니다

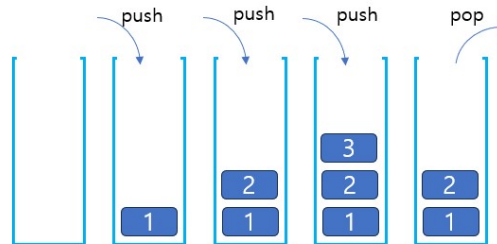
클로저





스택 구조

- 스택 구조란 흔히 Last In – First Out 으로 어떤 정보가 유지되는 구조를 의미합니다.
- 만들어진 순서대로 차곡차곡 쌓아놓았다가 가장 위에 것부터 제거하는 구조입니다.
- 그럼으로 가장 마지막에 추가된 것이 가장 먼저 제거된다고 해서 Last In – First Out 구조라고 합니다.



클로저

실행 컨텍스트

- 실행 컨텍스트는 함수 호출 순서대로 스택 구조로 유지됩니다.

```
let x = 10

function oneFun(){
  let y = 20
  console.log(x, y)
}

function twoFun(){
  let z = 30
  console.log(x, z)
}

console.log(x)

oneFun()

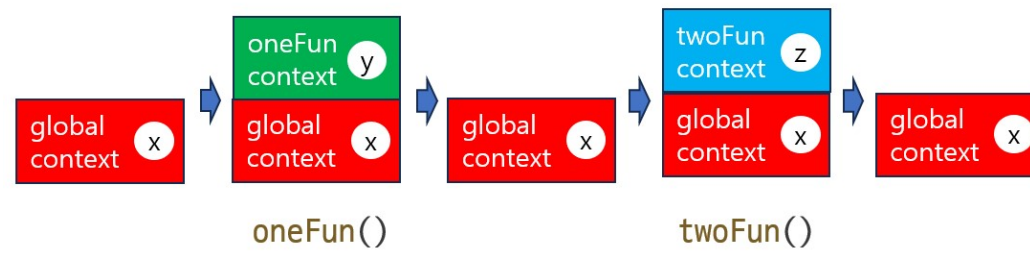
twoFun()
```

The diagram illustrates the execution context stack. It consists of three stacked boxes:

- oneFun context** (green box): Contains variable `y` (value 20).
- global context** (red box): Contains variable `x` (value 10).
- twoFun context** (blue box): Contains variable `z` (value 30).
- global context** (red box): Contains variable `x` (value 10).

Arrows indicate the call sequence: `oneFun()` calls `twoFun()`, and `twoFun()` calls `console.log(x)`.

클로저



클로저

실행 컨텍스트

- 함수를 위한 컨텍스트 정보가 유지됨으로 전역 위치에 선언한 변수는 모든 함수에서 사용이 가능한 것이며 특정 함수내에 선언한 로컬 변수들은 그 함수 호출이 끝나면 제거됨으로 그 함수가 실행되는 도중에만 사용할 수 있게 되는 것입니다.

클로저

렉시컬 환경이란

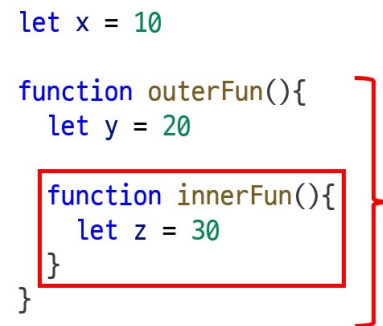
- 렉시컬(Lexical) 이란 영어 단어 뜻은 '허위', '구문' 이라는 뜻이 있습니다.
- 렉시컬 환경 혹은 렉시컬 스코프라고 하는데 이 의미는 함수가 선언된 위치를 의미합니다.
- 즉 함수가 실행되는 동적인 환경이 아닌 코드로 함수를 작성한 위치를 의미합니다.

클로저

렉시컬 환경이란

```
let x = 10

function outerFun(){
  let y = 20
  function innerFun(){
    let z = 30
  }
}
```



- innerFun() 을 outerFun() 함수 내에 선언했으므로 innerFun() 의 렉시컬 환경은 outerFun() 입니다.

클로저

클로저가 왜 필요한가?

- 함수를 선언하고 그 함수를 호출해서 실행시키는 모든 상황에서 클로저가 만들어지지 않습니다.
- 혹은 개발하면서 클로저를 모든 함수 호출시에 고려해야 하는 것은 아닙니다.
- 클로저가 필요한 경우는 함수를 호출하는 곳이 함수가 선언된 렉시컬 환경 내부가 아닌 경우입니다.

클로저


클로저가 왜 필요한가?

```
let x = 10

function outerFun(){
  let y = 20

  function innerFun(){
    let z = 30
    console.log(x, y, z)
  }
  innerFun()
}

outerFun()
```



The diagram illustrates the lexical environment stack during the execution of the code. It consists of three stacked boxes: a green box at the top labeled 'innerFun context' containing the variable 'z', a blue box in the middle labeled 'outerFun context' containing the variable 'y', and a red box at the bottom labeled 'global context' containing the variable 'x'. A double dash '--' is positioned to the left of the 'outerFun context' box, indicating the current execution context.

- 위의 경우는 함수의 실행이 자신이 선언된 렉시컬 환경내에서 실행된 경우입니다.
- 클로저가 만들어지지 않습니다.

클로저

클로저가 왜 필요한가?

- 모든 함수의 실행이 렉시컬 환경에서 실행된다고 볼 수 없습니다.

```
let x = 10

function outerFun() {
  let y = 20

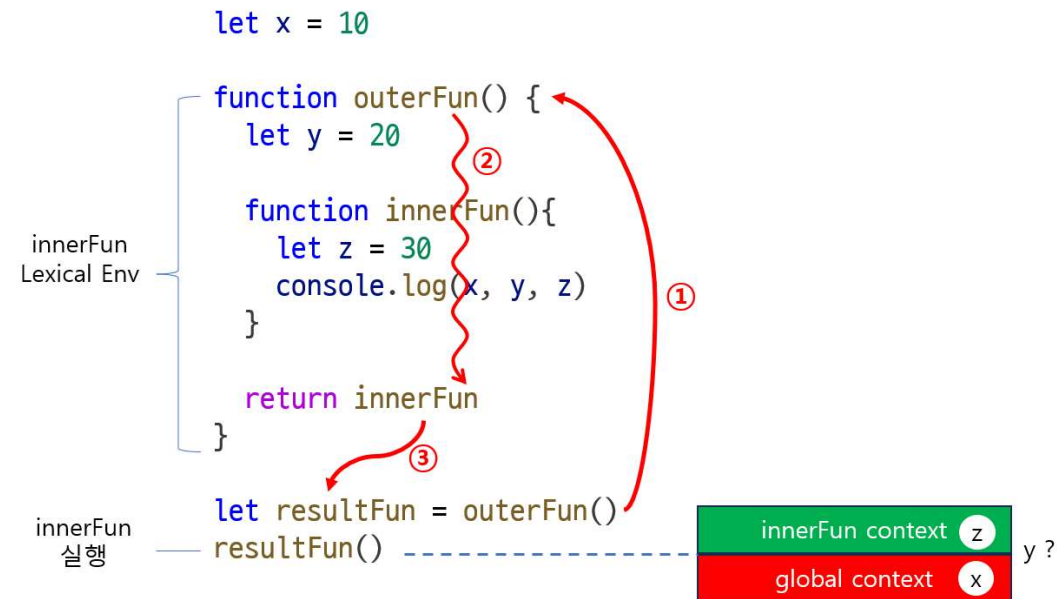
  function innerFun(){
    let z = 30
    console.log(x, y, z)//10, 20, 30
  }

  return innerFun
}

let resultFun = outerFun()
resultFun()
```

클로저

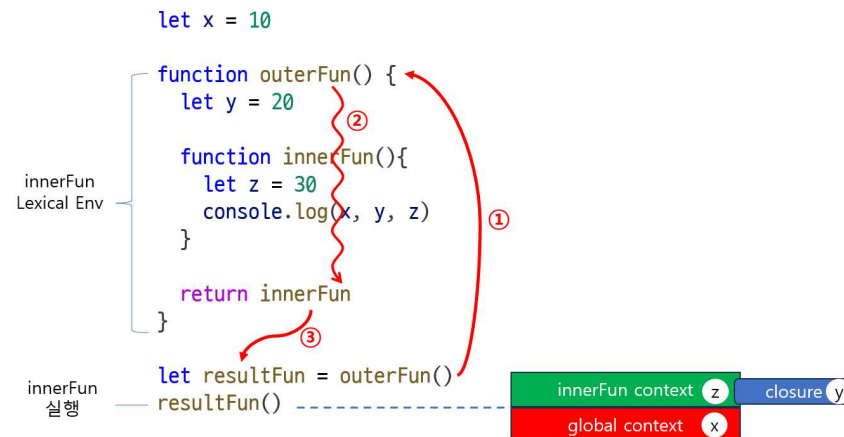
클로저가 왜 필요한가?



클로저

클로저가 왜 필요한가?

- 클로저를 정의할 때 "클로저(Closure)란 함수와 함수가 선언되었을 때의 렉시컬 환경(Lexical environment)의 조합"이라고 했습니다.
- 어떤 함수가 그 함수가 선언된 위치 외부에 전전달되 실행될 때 자신이 선언된 렉시컬 환경의 정보를 이용할 수 있게 클로저가 자동으로 만들어져서 함수에 추가되게 됩니다.





캡슐화(Encapsulation)

- 캡슐화는 객체지향 프로그래밍에 자주 등장하는 용어입니다.
- 캡슐내에 내용을 담듯이 관련된 프로퍼티, 메서드를 하나로 묶어서 작성한다는 의미에서 캡슐화라고 부릅니다.
- 자바스크립트 측면에서는 생성자 함수를 이용하거나 클래스를 이용해 캡슐화 프로그램을 작성합니다.
- 캡슐화를 하는 주된 이유는 두가지입니다.
- 관련된 것을 묶어서 하나의 프로그램 단위로 개발하고 관리해 개발 생산성 및 유지보수성을 증대하겠다는 이유와 캡슐로 내용을 묶고 외부에 캡슐 내의 내용을 감추어 정보 은닉을 하고자 하는 목적입니다.

즉시 실행 함수

- 즉시 실행 함수란 선언하자마자 호출해 사용하는 함수를 의미합니다.
- 일반적으로 함수는 선언한 후 필요할 때 해당 함수를 반복적으로 호출해서 사용합니다.
- 그럼으로 함수 이름을 지정해야 하고 그 이름으로 호출해 사용됩니다.
- 그런데 즉시 실행 함수는 이름을 가지지 않습니다. 그럼으로 선언과 동시에 호출하지 않는 한 다시 호출할 수 없게 됩니다.
- 즉시 실행 함수는 () 안에 `function() { }` 형태로 선언되며 이렇게 선언하자마자 () 로 호출하여 이용됩니다.



즉시 실행 함수

- 즉시 실행 함수가 이용되는 대부분의 이유는 어떤 변수, 함수가 이용되는 범위를 즉시 실행 함수로 묶어 즉시 실행 함수 내에서만 사용되게 하고자 하는 목적입니다.
- 즉 변수, 함수가 전역에서 사용되지 못하고 즉시 실행 함수내에서만 사용되게 하고자 하는 목적입니다.

전역 변수, 함수

```
1 let data = 10
2 function myFun(){
3   console.log(data)
4 }
5 myFun();//10
6
7 data = 20
8 myFun();//20
```

```
(function () {
  let data = 10;
  function myFun() {
    console.log(data);
  }
  myFun(); //10
})();

// data = 20;//error
// myFun(); //error
```

즉시 실행 함수

- 즉시 실행 함수를 이용해 특정 영역을 선언하고 그 영역내에서만 변수, 함수가 이용되게 하는 기법은 여러 상황에서 유용한데 하나의 예가 라이브러리 코드 개발자 입장입니다.
- 라이브러리는 그 라이브러리를 누군가의 코드에서 이용한다는 개념인데 라이브러리내에 선언된 변수, 함수명이 그 라이브러리를 이용하는 곳까지 영향을 미쳐 의도치 않게 문제가 발생할 수 있습니다.
- 라이브러리를 만들면서 라이브러리내에서만 사용하고자 하는 변수, 함수가 있다면 이후 이 라이브러리를 이용하는 곳에 영향을 주지 않게 하기 위해서 즉시 실행함수로 라이브러리 내에서만 사용하게 할 수 있습니다.

즉시 실행 함수

- 즉시 실행 함수가 이용되는 유용한 또다른 예가 여러 함수에서 이용되는 변수를 선언하는데 그 변수가 특정 함수들 내에서만 사용되게 하고자 할 때 이용할 수 있습니다.

```
let count = 0

function increment(){
  count++
}
function decrement(){
  count--
}
```

즉시 실행 함수

```
const counter = (function () {  
  let count = 0;  
  return function (argFun) {  
    count = argFun(count);  
    return count;  
  };  
})();  
let increment = (no) => ++no  
let decrement = (no) => --no
```



감사합니다

단단히 마음먹고 떠난 사람은
산꼭대기에 도착할 수 있다.
산은 올라가는 사람에게만 정복된다.



윌리엄 셰익스피어
William Shakespeare