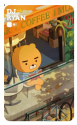

⭐Kakao Cloud(Day 4)⭐

Object.prototype.__proto__

Object의prototype을 instance가 상속받기 때문에

instance에서 __proto__를 사용할 수 있다.

만드는순간 프로토타입 객체 만들어지고

오브젝트.prototype에 대한 링크를 가리킨다.

⭐"__proto__" 접근자 property⭐

→ Object prototype이 가지고 있는 property

```
const obj = {};

const parent = { x: 1 };

obj.__proto__ = parent;

console.log(obj.x); // 1
```

```
const obj = {}; // 객체 literal로 만든 객체

console.log(obj.__proto__.constructor.name); // Object
```

⇒ **객체 literal로 만든객체는 Object를 가리키고 있다.**

그런데 "__proto__"라는 표현이 code에 직접 나오는건 권장하지 않아요

```
// 객체 생성하는 방법 중 하나
// 객체를 생성할 때 객체의 상위 prototype객체를 직접 지정
const obj = Object.create(null); // 상위 prototype 객체가 없다고 지정

console.log(obj.__proto__); // undefined

// 그럼 이런 경우를 위해 어떻게 사용하는것이 좋을까요?
// Object가 가지고 있는 method를 이용하는게 좋아요!
```

```
console.log(Object.getPrototypeOf(obj)); // null
// -> 이 method를 사용하는것을 권장한다.
```

Object.getPrototypeOf ⇒ method 이용 권장!!

⭐**prototype property**⭐

⇒ constructor만 가질 수 있는 property

constructor ⇒ instance를 만들어 낼 수 있는 능력을 가진 것

→ 함수선언문, 함수표현식, class ⇒ 생성자 함수로 사용

3개만 constructor를 가지고 있다.

따라서 **일반객체**(객체 literal)

non-constructor → Arrow function

→ ES6 method

→ **prototype property를 가지지 않아요!!**

⭐**prototype EX1**⭐

```
// 함수표현식

var foo = function () {};
console.log(foo.__proto__ === Function.prototype); // true

console.log(foo.prototype.__proto__ === Object.prototype); // true

console.log(Object.prototype.__proto__); // null

console.log(foo.constructor === Function); //  true
```

⭐**literal**⭐

{} → 객체 literal Object.protype

→ Object

함수 literal → Function.prototype

→ Function

[1,2,3] → 배열 literal

→ Array.prototype

⭐prototype 객체는 언제 생성되나요?⭐

생성자함수와 같이 만들어져요

built-in(전역)함수 → Java Script Engine이 기동하면 생성된다.

⇒ Object,Function,Array,Math,JSON,Number,String … 등등

⭐Overriding, property shadowing⭐

```
function Person(name) {
  this.name = name;

  //instance method
  this.getName = function () {};
}

// prototype method

Person.prototype.sayHello = function () {
  console.log(`안녕하세요. ${this.name}`);
};
// instance를 생성
const me = new Person("홍길동");

// overriding
me.sayHello = function () {
  console.log(`Hello ${this.name}`);
};
// 만약 overriding이 발생하면 이 발생된 overriding에
// 의해서 숨겨진 prototype method를 property shadowing
// 되었다고 말한다.
me.sayHello(); // 안녕하세요. 홍길동
```

⭐Strict mode 사용시 주의점⭐

this ⇒현재 사용하는 객체에 대한 reference

1. 묵시적 전역을 사용할 수 없다.
2. 변수,함수,매개변수가 delete로 삭제가 안된다
3. this의 의미가 달라진다
4. 일반한수에서 this → window 를 가리키지만

Strict mode 에서는 undefined를 가리킨다.

⭐Closure(클로저)⭐

→ Java Script 고유의 개념이 아니다.

→ 함수형 언어들이 가지고 있는 특징❗❗

First-class citizen

object ⇒ (일급객체)에 대한 이해 선행

⭐일급객체란⭐

1. 익명으로 생성이 가능해야 한다.(runtime에 생성 가능)
2. 값으로 저장이 가능해야 한다.(변수나 자료구조에 저장가능)
3. 함수의 매개변수로 전달이 가능해야 한다.
4. 함수의 return 값으로 사용이 가능해야 한다.

Java Script의 함수는 4가지 조건을 모두 만족한다.

⇒ 일급객체,일급함수로 취급된다.

closure는 함수와 그 함수가 선언된 lexical 환경의 조합

Q. lexical environment가 뭔가요???

A. 나도 몰라요!! excution context를 알아야한대요

⭐ECMA Script 명세⭐

1. 전역 code → 전역에 존재하는 source code, 전역함수의 내부코드는 포함 X
2. 함수 code → 함수 내부에 존재하는 code, 중첩함수의 내부코드는 포함 X

– –

3. eval code
4. 모듈 code

⭐전역코드⭐

전역변수를 관리하기 위해 전역 scope를 생성, var keyword로 선언된

식별자 찾고 window객체 만들고 식벽자를 window객체에 binding

→ 이런 작업을 하기위해 전역 실행 컨텍스트를 생성

⭐함수내부⭐

함수내부에서 사용하는 지역변수,매개변수,arguments가 관리 되어야되고 지역 scope를 생성한 다음 이 지역 scope를 전역 scope와 연결해서 scope 체인을 만든다.

→ 이런 작업을 하기위해 함수 실행 컨텍스트를 생성

⭐Closure(클로저) 예제⭐

```
const x = 1;
function outer() {
  const x = 10;
  const inner = function () { // 중첩함수
    console.log(x); // 외부함수의 식별자를 참조한다
  };
  return inner; // 리턴되는 중첩함수가 외부함수의 식별자 const x = 10을 참조
}
const innerFunc = outer(); // 외부함수 실행으로 execution context stack에서 제거되지만
// lexical 환경은 메모리에 남아있어서 중첩함수에 의해 const x = 10 이 사용될 수 있다.
innerFunc(); // 10
```

1. **closure는 중첩함수이다.**
2. **이 중첩함수가 외부함수의 결과값으로 return**
3. **return되는 중첩함수가 외부함수의 식별자를 참조한다.**
4. **return되는 중첩함수의 life cycle이 외부함수보다 길어야 한다.**
5. **이때 중첩함수에서 외부함수에 대한 참조가 남아있기 때문에 외부함수의 실행은 execution context stack에서 제거되지만 외부함수의 lexical 환경은 메모리에 남아있어서 중첩함수에 의해 사용될 수 있다.**

⇒ 이 조건을 모두 만족해야 closure 함수이다.

Q. 자 그럼 대체 closure는 어디에 쓰는 건가요???

A. 답은 코드에서 찾아보자

```
const increase = (function () {
  let num = 0;
  return function () {
    return ++num;
  };
})();

console.log(increase()); // 1
console.log(increase()); // 2
console.log(increase()); // 3
```

```
const counter = (function () {
  let num = 0;

  return {
    increase() {
      return ++num;
    },
    decrease() {
      return --num;
    },
  };
})();
console.log(counter.increase()); // 1
console.log(counter.increase()); // 2
console.log(counter.decrease()); // 1
console.log(counter.decrease()); // 0
```