Kakao Cloud(Day 2)

Java Script → 객체지향언어,객체기반언어

→ Java Script 구성하는 대부분은 객체

메모리에 왜 Undefined위에 안덮고 새로운 공간에 할당할까?

같은 공간에 지우고 할당하면 기존에 있는게 지워지기 때문에

어디에서 잘못됬는지 찾아낼 수 가 없다

추적하기가 쉽다

객체는 왜 변경가능하지??

→ 객체는 Primitive Type Value보다 사이즈가 크기때문에

새로운 공간에 계속 할당하면 메모리 효율성이 떨어진다.

객체란?

0개 이상의 Property의 집합이고 각 Property는 Key,Value의 쌍으로 구성!

```
var Person = {

name : ‘홍길동’,

age : 20 }

name,age ⇒ Property Key

‘홍길동’, 20 ⇒ Property Value
```

property의 key : 문자열, symbol값

property의 value : JavaScript에서 Value로 인식되는 모든것

→ 함수가 나올 수 있다 = method

객체를 생성하려면 어떻게 해야 하나요?

1. 객체 literal → 가장 간단한 방법❗
2. Object 생성자 함수
3. 사용자가 만든 생성자 함수 (User defined)
4. Object.create() method
5. ES6 version → Class를 이용해서 만들어 낼 수 있다.

Property의 동적 추가 && 삭제

• (dot rotation) → 일반적인 Property 추가

[ ](bracket rotation) ⇒ naming rule에 맞지 않는 Property 추가할 때

```
var obj = {
  myName: "홍길동",
};

obj.myAge = 20;
obj["!myPhone"] = "010-1234-5678";
// naming rule에 맞지 않는 property를 추가할때는 [] bracket rotation을 써야한다.
console.log(obj);
```

기타 다른 형태 → 22.07.05 code(propertyEtc.js)

→ key value를 중복해서 사용하게 되면 먼저 선언했던 값이 날아간다.

```
var obj = {
  10: 100,
  let: "권장되지 않아요!",
  myName: "홍길동",
  "!myName": "김길동",
  myName: "김연아",
};
console.log(obj);
console.log(obj.myAddress); // Undefined
```

ES6에서 추가된 객체 literal 확장 code로 확인!(축약표현)

→ 22.07.05(literalExpansion.js)

```
let x = 1;
let y = 2;

const obj = { x, y }; // 확장표현방식
// 식별자를 property key로 가질때 축약형
// 식별자를 가져다가 키로쓰고 그 식별자가 가지고있는 값을 property value로 쓴다

console.log(obj);

/* let myObj = {
    name : '홍길동',
    printName: function () {
        console.log(this.name);
    }
}
*/
let myObj = {
  name: "홍길동",
  printName() {
    console.log(this.name); //축약형
  },
};
myObj.printName();
```

Primitive Value(원시값) VS 객체

→ (immutable) (mutable)

유사 배열 객체 → 22.07.05 code (Array-like-Object.js)

```
let myStr = "Hello";

// primitive type을 마치 객체(배열)처럼 사용
console.log(myStr[0]); // H
console.log(myStr.length); // 5

myStr[0] = "h";

console.log(myStr); // Hello
// 내부슬롯에 있는 인덱스값이 변경되기 때문에 원래의 값은 변경되지 않는다.
```

function(함수)

일련의 과정을 수행하기 위한 독립적인 실행단위(statement)를 {}를

이용해서 하나의 실행단위로 만들어 놓은 것.

→ 반복적인 code를 함수화 ⇒ 유지보수 up

→ 오류가 발생할 여지가 낮아진다.(code 신뢰도 상승)

함수를 정의(definition) → 호출 (call, invoke)

기명함수(named function)

→ function add(함수이름) (parameter(매개변수)) {

    return x+y; ⇒ return value

}

add(argument(인수));

함수 literal

함수는 객체!!!!

```
var func = function add(x,y) {

            return x+y; } -> 함수 literal
```

함수이름은 식별자(identifier)

→ 함수 이름은 함수 내부에서만 사용이 가능(외부로 노출 X)

→ 함수는 식별자로 호출, 그래서 대부분 익명함수(anonymous)로 사용

→ literal로 변수에 저장할 때 22.07.05 code(function.js)

```
var myFunc = function add(x, y) {
  return x + y;
};

console.log(myFunc(3, 5)); //  8
console.log(add(3, 5)); // 불가능
```

함수를 정의(definition)하는 방법

1.함수 선언문

```
ex) function add(x,y){
    return x+y;
```

```
}
```

2.함수 표현식

```
ex) var myFunc = function(x,y) {
    return x+y;
};
```

3.Function 생성자 함수 이용(대문자 F)

```
ex) var add = newFunction('x','y','return x+y') ⇒ 권장하지 않는다.
```

4.ES6 화살표 함수(Arrowfunction)

```
var add = (x,y) => x + y;
```

```
foo(); // 호출되요!
add(); // Error
//선언적 함수, 함수 표현식이냐에 따라서 함수 hoisting이 달라진다.

// 함수 선언문
function foo() {
  console.log("foo 함수"); // 눈에는 보이지 않지만 foo라는 변수를 묵시적으로 생성
}


// 함수 표현식
(function bar() {
  console.log("bar 함수");
}); // ()안에 있으면 계산하라는 얘기, 즉 평가
// foo();  가능 'foo 함수'
// bar();  불가능
```

함수 호출

Java Script의 함수는 Overloading이 발생하지 않아요!

→ 인자의 개수가 틀려도 호출 가능

→ arguments의 내부객체를 이용한다. 22.07.05 code(arguments.js)

```
// 함수 선언문

function add() {
  // arguments  [2,3,4]
  let sum = 0;
  // arguments는 유사배열객체(Array-like Object)
  // 모든 유사배열객체는 length property를 가지고 배열처럼 index를 이용해서
  // access가능. 당연히 순환가능
  for (let i = 0; i < arguments.length; i++) {
    sum += arguments[i];
  }
  return sum; // -> return구문이 없으면 Undefined가 return된다.
}

console.log(add(2)); // 2
console.log(add(2, 3, 4)); // 9
```

IIFE

Immediately Invoked Function Expression ⇒ 즉시 실행 함수

→ 함수를 선언함과 동시에 호출

→ 함수의 재사용이 불가능.

→ 재사용이 불가능 하기때문에 대부분 익명함수로 사용한다.

```
// IIFE (즉시 실행 함수)
(function add() {
  let x = 10;
  let y = 20;

  console.log(x + y); // 30
})();
```

왜 쓰나요⁉

가장 대표적인 예로 전역변수를 지역변수화 할 수 있다.

first-class citizen,object (일급함수)

1 익명의 literal로 생성가능 → 동적으로 생성가능

2 객체가 변수나 자료구조에 저장 가능

3 객체를 다른 함수의 인자로 전달 가능

4 함수의 리턴값으로 객체를 사용

⇒ Java Script 함수는 일급객체❗

Callback function

```
// 잘 만든 함수가 존재!
// 그런데 이 함수의 기능을 변경(추가)
// 1. 원래 있는 함수를 수정!
// 2. 함수를 새로 추가해서 만들어요!
// 3. 함수를 추상화 시켜서 인자로 받아서 사용!

function repeat(n, f) { // 고차함수(Higher-ordered Function)
  for (var i = 0; i < n; i++) {
    f(i);
  }
}

let logAll = function (i) { // callback 함수
  console.log(i);
};

let logOdd = function (i) { // callback 함수
  if (i % 2) {
    console.log(i);
  }
};

repeat(3, logAll);
// repeat(5, logOdd);
```

Scope

→ 식별자가 유효한 범위❗

JavaScript Engine이 identifier을 찾을 때 사용하는 메커니즘

⇒ Scope Chain

*** JavaScript Engine은 코드를 실행할 때 문맥(context)를 고려해서 실행한다.***

→ 현재 실행중인 code가 어디에 있는 code이고, code주변 정보를 파악해서 실행

⇒ lexical Environment → 이것을 실제로 구현해 놓은것 ⇒ excution context

함수가 호출 되었을 때

Scope를 함수가 호출된 곳을 기준으로 설정 ⇒ dynamic scope(동적)

함수가 정의된 곳을 기준으로 설정 ⇒ static scope(정적) = lexical scope(정적)

Java Script = lexical scope (거의 대부분의 언어가 정적 scope)

```
var x = 1;

function loo() {
  var x = 10;
  bar();
}

function bar() {
  console.log(x); // 1이 출력된다.
}
loo();
```

진짜 머리가 터질 것 같아,, 내 조그만 뇌로 감당할 수가 없다,,,😭

전역변수

1 가독성이 나빠진다 (오류의 여지가 많다)

2 메모리 resource를 소모

3 변수를 늦게 찾는다 (효율 Bad)

4 다른파일과 변수 충돌이 생길 수 있다.

⇒ 사용을 줄여야 한다. ⇒ IIFE 를 이용해 전역변수를 줄일 수 있다.

객체는 Property의 집합

→ Property는 Property attribute를 가져요❗❗

→ Property attribute → Property를 생성할 때 해당 property의 상세를 나타내는 값(기본적으로 정의된다.)

Property의 상세

1 property의 값 → [[Value]]

2 property의 값을 수정할 수 있는지 여부 → [[Writable]]

3 해당 property가 열거될 수 있는지 여부 → [[Enumerable]]

4 해당 property attribute를 재정의 할 수 있는지 여부 → [[Configurable]]

⇒ 전부 다 내부슬롯이라서 직접적인 접근을 할 수가 없다.

```
// Property Attribute를 확인해 보아요!!

const person = {
  name: "Lee",
  age: 20,
};
console.log(Object.getOwnPropertyDescriptor(person, "name"));

result = { value: 'Lee', writable: true, enumerable: true, configurable: true }

// 다 가져 올려면
console.log(Object.getOwnPropertyDescriptors(person));
 result =>
 name: {
    value: 'Lee',
    writable: true,
    enumerable: true,
    configurable: true
  },
  age: {
    value: 20,
    writable: true,
    enumerable: true,
    configurable: true }
```

Property Defined

Object.defineProperty()

```
const person = {
  age: 20,
};

// person.name = '홍길동';

Object.defineProperty(person, "name", { // property attribute 재정의
  value: "Shin",
  writable: false,
```

```
  enumerable: false,
  configurable: true,
});

console.log(person);
console.log(Object.getOwnPropertyDescriptor(person, "name"));

person.name = "아이유";
console.log(person);
console.log(Object.keys(person));

for (let idx in person) {
  // 열거 시키는 구문
  console.log(person[idx]); // property key
}
for (let value in person) {
  // 열거 시키는 구문
  console.log(value); // property value
}
```

내부 slot, method

Java Script 객체는 내부 slot → [[…]]

내부 method → [[…]]

내부 method는 개발자각 직접적으로 사용할 수 없어요❗❗

(Java Engine에 의해 사용됩니다!!)

obj .__proto__ => [[Prototype]]