2주 4회차

1. 밑의 코드를 실행해보고 이유를 생각해보시오.

```
const original = [
[18, 18, 18, 18],
[19, 19, 19, 19],
[20, 20, 20, 20],
[21, 21, 21, 21],
];
const copy = original.slice();
console.log(JSON.stringify(original) === JSON.stringify(copy));
copy[0][0] = 99;
copy[2].push("02");
console.log(JSON.stringify(original) === JSON.stringify(copy));
console.log(original);
console.log(copy);
```

- 출력결과

```
true
true
[
  [ 99, 18, 18, 18 ],
  [ 19, 19, 19, 19 ],
  [ 20, 20, 20, 20, '02' ],
  [ 21, 21, 21, 21 ]
]
[
  [ 99, 18, 18, 18 ],
  [ 19, 19, 19, 19 ],
  [ 20, 20, 20, 20, '02' ],
  [ 21, 21, 21, 21 ]
]
```

- `const copy = original.slice();` : 원래 배열의 안의 배열은 얕은 복사를 하므로 `copy`와 `original`은 서로 다른 배열이지만 그 안에 배열의 경우 기존 변수의 배열을 참조한다.
- `console.log(JSON.stringify(original) === JSON.stringify(copy));` : 두 배열이 동일한 내용을 참조하고 있으므로 true를 출력한다.

- `copy[0][0] = 99;` : `copy` 배열의 첫 번째 배열의 첫 번째 값을 변경하면 `original` 배열에 반영된다. 즉, `copy`와 `original` 배열의 첫 번째 배열의 첫 번째 값은 `99`로 변경한다.
- `copy[2].push("02");` : `copy` 배열의 세 번째 배열에 값을 `push`하면 이 값도 `original`에 반영된다. 즉, `copy`와 `original` 배열의 두 번째 배열의 마지막 값에 `02`를 추가한다.
- `console.log(JSON.stringify(original) === JSON.stringify(copy));` : 두 배열이 동일한 내용을 참조하고 있으므로 true를 출력한다.
- `console.log(original);` , `console.log(copy);` : 두 객체를 출력하면 같은 값을 참조하므로 같은 값을 출력한다.

2. 밑에 있는 코드들을 실행해 보시고 이유를 생각해보세요.

- 코드 1

```
const obj = { a: 1 };
const newObj = Object.assign({}, obj);
newObj.a = 2;
console.log(obj);
console.log(obj === newObj);
```

- 출력결과

```
{ a: 1 }
false
```

- `const newObj = Object.assign({}, obj);` : `Object.assign()` 메서드를 사용하여 새로운 객체 `newObj`를 생성한다. 이때, 빈 객체 `{}`에 `obj`의 속성을 복사하여 할당한다.
- `newObj.a = 2;` : `newObj`의 속성 `a` 값을 `2`로 변경한다.
- `console.log(obj);` : `obj`의 값이 변경되지 않았으므로 `{ a: 1 }`을 출력한다.

- `console.log(obj === newObj);` : 원본 객체와 복사된 객체는 서로 다른 참조를 가진 객체이므로 `false`를 출력한다.

- 코드 2

```
const obj = {
a: 1,
b: {
c: 2,
},
};
const newObj = Object.assign({}, obj);
newObj.b.c = 3;
console.log(obj);
console.log(obj.b.c === newObj.b.c);
```

- 출력결과

```
{ a: 1, b: { c: 3 } }
true
```

- `const newObj = Object.assign({}, obj);` : `Object.assign()` 메서드를 사용하여 새로운 객체 `newObj`를 생성한다. 이때, 빈 객체 `{}`에 `obj`의 속성을 복사하여 할당한다.
- `newObj.b.c = 3;` : `newObj`의 속성 `b.c`값을 `3`으로 변경한다. 이때, `c`는 2차원 객체이므로 얕은 복사가 됐으므로 원본 객체 `obj`의 속성 `b.c` 값도 `3` 으로 변경한다.
- `console.log(obj);` : 원본 객체인 `obj`의 값이 변경되었으므로 `{ a: 1, b: { c: 3 } }`을 출력한다.
- `console.log(obj.b.c === newObj.b.c);` **:** 원본 객체와 복사된 객체는 서로 같은 값을 참조하므로 `true`를 출력한다.

- 코드 3 (코드 1과 같다)

```
const obj = { a: 1 };
const newObj = Object.assign({}, obj);
newObj.a = 2;
console.log(obj);
console.log(obj === newObj);
```

- 출력결과

```
{ a: 1 }
false
```

- 코드 4 (코드 3을 Spread 연산자를 사용해 나타내보았다)

```
const obj = { a: 1 };
const newObj = { ...obj };
newObj.a = 2;
console.log(obj);
console.log(obj === newObj);
```

- 출력결과

```
{ a: 1 }
false
```

- `const newObj = { ...obj };` : Spread 연산자를 사용하여 새로운 객체 `newObj`를 생성한다. 이 때, `{}`에 `obj`의 속성을 복사하여 할당한다.
- `newObj.a = 2;` : `newObj`의 속성 `a` 값을 `2`로 변경한다.
- `console.log(obj);` : `obj`의 값이 변경되지 않았으므로 `{ a: 1 }`을 출력한다.
- `console.log(obj === newObj);` : 원본 객체와 복사된 객체는 서로 다른 참조를 가진 객체이므로 `false`를 출력한다.

- 코드 5

```
const obj = {
a: 1,
b: {
c: 2,
},
};
const newObj = { ...obj };
newObj.b.c = 3;
console.log(obj);
console.log(obj.b.c === newObj.b.c);
```

- 출력결과

```
{ a: 1, b: { c: 3 } }
true
```

- `const newObj = { ...obj };` : Spread 연산자를 사용하여 새로운 객체 `newObj`를 생성한다. 이 때, `{}`에 `obj`의 속성을 복사하여 할당한다.
- `newObj.b.c = 3;` : `newObj`의 속성 `b.c`값을 `3`으로 변경한다. 이때, `c`는 2차원 객체이므로 얕은 복사가 됐으므로 원본 객체 `obj`의 속성 `b.c` 값도 `3` 으로 변경한다.
- `console.log(obj);` : 원본 객체인 `obj`의 값이 변경되었으므로 `{ a: 1, b: { c: 3 } }`을 출력한다.
- `console.log(obj.b.c === newObj.b.c);` : 원본 객체와 복사된 객체는 서로 같은 값을 참조하므로 `true`를 출력한다.

3. 아래 함수를 작동 시켜보세요(?).

```
function deepCopy(obj) {
    if (obj === null || typeof obj !== "object") {
        return obj;
    }
    let copy = {};
    for (let key in obj) {
        copy[key] = deepCopy(obj[key]);
    }
    return copy;
}
const obj = {
    a: 1,
    b: {
    c: 2,
    },
    func: function () {
        return this.a;
    },
};
const newObj = deepCopy(obj);
newObj.b.c = 3;
console.log(obj);
console.log(obj.b.c === newObj.b.c);
```

- 출력 결과

```
{ a: 1, b: { c: 2 }, func: [Function: func] }
false
```

- `deepCopy` : 재귀적으로 객체의 속성을 복사해 새로운 객체를 생성하는 함수다. (깊은 복사를 한다.)
- `if (obj === null || typeof obj !== "object") { return obj; }` : 입력된 `obj`가 `null` 이거나 객체가 아닌 경우 그대로 반환한다.
- `for (let key in obj) { copy[key] = deepCopy(obj[key]); }` : `for` 문을 사용하여 `obj` 의 모든 속성들을 순회한다. 해당 속성의 값이 원시 타입일 경우 그대로 할당하고, 객체일 경우 재귀적으로 이 함수를 호출하여 깊은 복사를 한다.
- `return copy;` : 모든 속성들이 복사된 새로운 객체를 반환한다.
- `const newObj = deepCopy(obj);` : `deepCopy` 함수를 사용하여 새로운 객체 `newObj`를 생성한다.
- `newObj.b.c = 3;` : `newObj`의 속성 `b.c` 값을 `3`으로 변경한다.
- `console.log(obj);` : `obj`의 값이 변경되지 않았으므로 `{ a: 1, b: { c: 2 }, func: [Function: func] }`을 출력한다.
- `console.log(obj.b.c === newObj.b.c);` : 원본 객체와 복사된 객체가 서로 다른 `b.c` 값을 참조하므로 `false`를 출력한다.

4. Lodash 라이브러리에 대해서 조사하고 cloneDeep 메서드도 조사해보세요.

- Loadash 라이브러리
  - 자바스크립트 라이브러리로 객체, 배열 등의 데이터 구조를 쉽게 변환하여 사용하게 도와준다.
  - handling 할 때 유용하고 빠른 작업과 간결한 코드를 사용할 수 있다.
  - _라는 기호를 사용한다.

- cloneDeep 메서드
  - Loadash의 라이브러리로 깊은 복사를 할 수 있다.
  - 더 쉽고 안전하게 깊은 복사를 할 수 있으며, 일반적인 개발에는 효율적이다.
  - 코딩 테스트에는 사용할 수 없고 외부 종속성이 추가되는 것이 단점이다.

5. 소개한 방법 이외에도 깊은복사, 얕은복사 방법을 찾아보세요
가능하다면 언어 별로 정리해도 좋고, JS만 하셔도 좋고, 단일 언어 하나만 하셔도 좋습니다.

5.1 array.concat()

- 얕은 복사다.
- 파라미터로는 배열 또는 값을 사용할 수 있다.
- 파라미터로 받은 값들을 기존의 배열과 병합해 새로운 배열을 만들어 리턴한다.
- 1차원 배열은 깊은 복사가 이행되지만 2차원 배열부터는 경우에는 얕은 복사가 이행된다.

- 예시코드 1 (1차원)

```
var arr = ['a', 'b', 'c', 'd'];
var newArr = [].concat(arr);
newArr[4] = 'f';
console.log(arr);
console.log(arr === newArr);
```

- 출력결과

```
[ 'a', 'b', 'c', 'd' ]
false
```

- 예시코드 2 (2차원)

```
var arr = ['a', 'b', 'c', ['d']];
var newArr = [].concat(arr);
newArr[3][0] = 'e';
console.log(arr);
console.log(arr[3][0] === newArr[3][0]);
```

- 출력결과

```
[ 'a', 'b', 'c', [ 'e' ] ]
true
```

5.2 Array.from()

- 얕은 복사다.
- 파라미터는 `arrayLike[, mapFn[, thisArg]]`로 `arrayLike`는 배열로 변환하고자 하는 유사 배열 객체나 반복 가능한 객체, `mapFn`는 배열의 모든 요소에 대해 호출할 맵핑 함수, `thisArg`는 `mapFn` 실행 시에 this로 사용할 값이다.
- 유사 배열 객체나 반복 가능한 객체를 얕게 복사해 새로운 배열 객체를 만들어 리턴한다.
- 1차원 배열은 깊은 복사가 이행되지만 2차원 배열부터는 경우에는 얕은 복사가 이행된다.

- 예시코드 1 (1차원)

```
var arr = ['a', 'b', 'c', 'd'];
var newArr = Array.from(arr);
newArr[4] = 'f';
console.log(arr);
console.log(arr === newArr);
```

- 출력결과

```
[ 'a', 'b', 'c', 'd' ]
false
```

- 예시코드 2 (2차원)

```
var arr = ['a', 'b', 'c', ['d']];
var newArr = Array.from(arr);
newArr[3][0] = 'e';
console.log(arr);
console.log(arr[3][0] === newArr[3][0]);
```

- 출력결과

```
[ 'a', 'b', 'c', [ 'e' ] ]
true
```

5.3 Object.create()

- 얕은 복사다
- 파라미터는 `proto[, propertiesObject]`로 `proto`는 새로 만든 객체의 프로토타입이어야 할 객체, `propertiesObject`는 자신의 속성이 열거가능한 객체는 해당 속성명으로 새로 만든 객체에 추가될 속성 설명자를 지정한다.
- 지정된 프로토타입 객체 및 속성을 갖는 새 객체를 만들어 리턴한다.
- 1차원 배열은 깊은 복사가 이행되지만 2차원 배열부터는 경우에는 얕은 복사가 이행된다.

- 예시코드 1 (1차원)

```
var arr = ['a', 'b', 'c', 'd'];
var newArr = Object.create(arr);
```

```
newArr[4] = 'f';
console.log(arr);
console.log(arr === newArr);
```

- 출력결과

```
[ 'a', 'b', 'c', 'd' ]
false
```

- 예시코드 2 (2차원)

```
var arr = ['a', 'b', 'c', ['d']];
var newArr = Object.create(arr);
newArr[3][0] = 'e';
console.log(arr);
console.log(arr[3][0] === newArr[3][0]);
```

- 출력결과

```
[ 'a', 'b', 'c', [ 'e' ] ]
true
```

5.4 Ramda 라이브러리의 R.clone() 메서드

- 깊은 복사다.
- 입력된 값의 깊은 복사본을 생성한다.
- 배열을 전부 복사하여 새 주소에 담기 때문에 참조를 공유하지 않는다.

- 예시코드 (2차원)

```
import * as R from 'ramda'
const obj = {
a: 1,
```

```
b: {
c: 2,
},
func: function () {
return this.a;
},
};
const newObj = R.clone(obj);
newObj.b.c = 3;
console.log(obj);
console.log(obj.b.c === newObj.b.c);
```

- 출력결과

```
{ a: 1, b: { c: 2 }, func: [Function: func] }
false
```