함수 심화학습

(번외) call/apply 정리

- **call 메서드**
  - 함수나 메소드 객체에는 call 메서드가 제공됨
  - 함수(메소드)를 호출하는 과정에서 this 값을 지정하며 호출하고 싶을 경우 사용
  - call(this, ...args)
    - 첫 번째 인자값으로 this로 사용할 값을, 그 뒤에는 **가변인자 형태**로 함수, 메서드로 전달할 함수 인자 제공 가능

```
function printThis() {
    console.log(this);
}

let person = { name: "John", age: 20 };
let dog = { name: "Sam", age: 5 };

// printThis 함수를 호출하되, this 값을 person 객체로 지정
printThis.call(person); // { name: "John", age: 20 }
// printThis 함수를 호출하되, this 값을 dog 객체로 지정
printThis.call(dog); // { name: "Sam", age: 5 }

function printName(prefix, times=1) {
    console.log(this);
    for(let i=0;i<times;i++) {
        console.log(prefix, this.name);
    }
}

// printName 함수를 호출하되, this 값을 person 객체로 지정하며 prefix 값으로 "Hello",
times 값으로 2를 전달
printName.call(person, "Hello", 2);
printName.call(dog, "Good");
```

- **apply 메서드**
  - call과 비슷한 용도로 사용하지만 **메서드에 인자를 전달하는 방식에 차이**가 있음
  - apply(this, args)
    - args 값은 함수 호출시 전달할 **인자값이 담긴 배열**

```
function printThis() {
    console.log(this);
}

let person = { name: "John", age: 20 };
let dog = { name: "Sam", age: 5 };

// 만약 단순히 this 값만 정해주고 싶은 상황이라면 call 메서드와 다른점 없음
```

```
// printThis 함수를 호출하되, this 값을 person 객체로 지정
printThis.apply(person); // { name: "John", age: 20 }
// printThis 함수를 호출하되, this 값을 dog 객체로 지정
printThis.apply(dog); // { name: "Sam", age: 5 }

function printName(prefix, times=1) {
    console.log(this);
    for(let i=0;i<times;i++) {
        console.log(prefix, this.name);
    }
}

// 그러나 call 메서드와 달리 인자를 전달할 때 가변 인자가 아니라 인자값이 담긴 "배열"을 전달한다는 차이점이 존재
printName.apply(person, ["Hello", 2]);
printName.apply(dog, ["Good"]);
// 굳이 배열을 전달하면서 call 메서드를 써야 한다면, 전개 연산자(...) 써서 배열을 가변 인자로 바꿔 전달해주기
printName.call(person, ...["Hello", 2])
```

함수 바인딩

- 객체 메서드가 객체 내부가 아닌 **다른 곳에 전달되어 호출되면 this 값 사라지는 현상**이 발생

```
let user = {
    firstName: "John",
    sayHi() { alert(`Hello, ${this.firstName}!`); }
};

// "Hello, John!" 출력
user.sayHi();

// "Hello, undefined!" 출력
setTimeout(user.sayHi, 1000);

// 위의 코드와 같은 똑같은 상황을 재현한 코드
/*
let f = user.sayHi;
setTimeout(f, 1000);
*/
```

메서드를 전달할 때, this 값도 제대로 유지하는 방법

방법 1: 래퍼

```
let user = {
    firstName: "John",
    sayHi() { alert(`Hello, ${this.firstName}!`); }
};

setTimeout(function() {
    // 주체(user)가 있는 상태에서 출력 (this가 보존)
    user.sayHi();
}, 1000);
```

방법 2: bind

- 모든 함수는 **this를 수정(바인딩)하게 해주는 내장 메서드 bind를 제공**
- func.bind(context)는 **함수처럼 호출 가능한 특수 객체(exotic object)를 반환**함
    - 객체를 호출하면 this가 context로 고정된 함수 func가 반환됨

```
let user = { firstName: "John" };

function func() {
    alert(this.firstName);
}

// func.bind(user)는 func의 this를 user로 바인딩한 변형된 함수
let funcUser = func.bind(user);
// "John"
funcUser();
```

객체의 메소드에 bind 적용 예제

```
let user = {
    firstName: "John",
    sayHi() {
        alert(`Hello, ${this.firstName}!`);
    }
};

// sayHi 변수에 저장된 바인딩된 함수의 this 값은 이 시점부터 user로 고정됨
// 단, "반환된 새로운 함수"의 this 값이 고정되는 것이고 user.sayHi가 바뀌는 것이 아님을 유의
let sayHi = user.sayHi.bind(user);
// sayHi에 저장된 함수는 user.sayHi가 아닌 "완전히 새로운, this가 user로 고정된 함수"
console.log(sayHi === user.sayHi); // 참조가 같지 않으므로 false 반환

// 이제 점(.) 앞에 주체가 되는 객체 없이 호출해도 this 값을 기억함
sayHi(); // Hello, John!

// 1초 이내에 user 값이 변화해도 sayHi는 기존 값을 사용
setTimeout(sayHi, 1000); // Hello, John!
user = { sayHi() { alert("또 다른 사용자!"); } };
```

- bind 메소드를 호출한 이후에 **다시 bind를 호출해도 이전 this 값은 변경되지 않음**

```
let o1 = { a: 100 };
let o2 = { a: 200 };
function printA() { alert(this.a); }

// 처음 bind 함수(or 메서드) 호출하여 o1으로 바인딩
let o1BindPrintA = printA.bind(o1);
o1BindPrintA(); // 100
// 이미 바인딩된 함수(or 메서드)는 다시 bind를 호출해도 o2로 바인딩되지 않음
let bindAgain = o1BindPrintA.bind(o2);
bindAgain(); // 100
```

부분 적용 (partial application)

- bind 메서드를 통해서 this 뿐만 아니라 **함수로 전달할 인수도 바인딩이 가능함**
    - 부분 적용을 사용하면 **기존 함수의 매개변수 값이 고정된 새로운 함수를 생성** 가능

```
function mul(a, b) { return a * b; }

// 첫 번째 전달 인수가 2로 고정된 mul(a=2, b) 함수 반환
let double = mul.bind(null, 2);
alert( double(3) ); // = mul(2, 3) = 6
alert( double(4) ); // = mul(2, 4) = 8
alert( double(5) ); // = mul(2, 5) = 10

// 첫 번째 전달 인수가 3으로 고정된 mul(a=3, b) 함수 반환
let triple = mul.bind(null, 3);
alert( triple(3) ); // = mul(3, 3) = 9
alert( triple(4) ); // = mul(3, 4) = 12
alert( triple(5) ); // = mul(3, 5) = 15

// 모든 전달 인수가 고정된 mul 함수 반환
let six = mul.bind(null, 2, 3);
alert( six() ); // = mul(2, 3) = 6
```

화살표 함수 다시 살펴보기

화살표 함수에는 this가 없습니다

- 화살표 함수 본문에서 this에 접근하면 **함수가 호출된 외부 환경에서 this 값을 가져옴**

```
let group = {
    title: "1모둠",
    students: ["보라", "호진", "지민"],

    showList() {
        // 화살표 함수가 정의되고 호출된 showList 함수의 this 값이 화살표 함수 내부의
this 값으로 사용됨
        this.students.forEach(student => alert(this.title + ': ' + student));
    }
};

// showList 호출 시점에 this는 group 이므로 화살표 함수 내부에서도 this는 group
group.showList();
```

일반 함수를 쓴 버전

```
let group = {
    title: "1모둠",
    students: ["보라", "호진", "지민"],

    showList() {
        // forEach로 전달된 콜백 함수가 this 값 참조가 사라진 상태에서 호출되므로 this는
undefined
        this.students.forEach(function(student) {
            // TypeError: Cannot read property 'title' of undefined
            alert(this.title + ': ' + student);
        });
    }
};

group.showList();
```

Q) 일반 함수를 쓰고 화살표 함수는 쓰지 않으면서 정상적으로 작동하게 하려면? (힌트: 클로저)

```
let group = {
    title: "1모둠",
    students: ["보라", "호진", "지민"],

    showList() {
        let self = this;
        // 클로저 특성을 활용하여 해결
        this.students.forEach(function(student) {
            alert(self.title + ': ' + student);
        });
    }
};

group.showList();
```

- (this가 없으므로) 화살표 함수는 **new 키워드와 함께 호출 불가** (즉, **생성자 함수로 사용할 수 없음**)

```
// 화살표 함수로 생성자 함수 정의해보기
const User = (name, age) => {
    this.name = name;
    this.age = age;
}

// Uncaught TypeError: User is not a constructor
let user = new User("John", 30);
```

화살표 함수엔 arguments가 없습니다

- 화살표 함수는 일반 함수와는 다르게 모든 인수에 접근할 수 있게 해주는 **유사 배열 객체 arguments를 지원하지 않음**

```
function f() {
    // 화살표 함수 바깥의 함수(f)의 arguments를 참조 (자체 arguments를 가지지 않음)
    let showArg = () => console.log(arguments); // [1, "Hello"]
    showArg();

    // 인자값을 여럿 전달했지만 바깥의 arguments를 참조
    let another = () => console.log(arguments); // [1, "Hello"]
    another(2, "World");

    function g() {
        // 화살표 함수 바깥의 함수(여기서는 g)의 arguments를 참조
        let showArg = () => console.log(arguments); // [2, "World"]
        showArg();
        let another = () => console.log(arguments); // [2, "World"]
        another(2, "World");
    }
    g(2, "World");
}

f(1, "Hello");
```

화살표 함수 특징 요약

```
1. this를 가지지 않습니다.
2. arguments를 지원하지 않습니다.
3. new와 함께 호출할 수 없습니다.
4. super가 없습니다.
```

- 화살표 함수는 컨텍스트가 있는 긴 코드보다는 **자체 컨텍스트(this)가 없는 짧은 코드를 담을 용도로 사용**됨