Chapter3.This

🖇️ 명시적으로 this를 바인딩하는 방법

▼ call 메서드

```
Function.prototype.call(thisArg[, arg1[, arg2[, ...]]])
```

첫번째 인자를 this로 바인딩 하고, 이후의 인자들은 호출할 함수의 매개변수가 됩니다.

함수를 그냥 실행하면 this는 전역 객체를 참조하지만,

`call` 메서드를 이용하면 임의의 객체를 this로 지정할 수 있습니다.

```
// 함수
var function1 = function (a, b, c) {
  console.log(this, a, b, c);
};

function1(1, 2, 3); // Window 1 2 3
function1.call({ x: 1 }, 4, 5, 6); // { x: 1 } 4 5 6



// 메서드
var obj = {
  a: 1,
  method1: function (x, y) {
    console.log(this.a, x, y);
  }
};

obj.method1(2, 3); // 1, 2, 3
obj.method1.call({ a: 4 }, 5, 6); // 4, 5, 6
```

▼ apply 메서드

```
Function.prototype.apply(thisArg[, argsArray])
```

apply 메서드는 call 메서드와 기능적으로 완전 동일하며 `call()`은 인수 목록을, `apply()`는 인수 배열 하나를 받는다는점에서만 차이가 있습니다.

```
// call과 apply 비교
// 함수
var func1 = function (a, b, c) {
  console.log(this, a, b, c);
};

func1.call({ x: 1 }, 4, 5, 6); // { x: 1 } 4 5 6
func1.apply({ x: 1 }, [4, 5, 6]); // { x: 1 } 4 5 6
```

▼ call / apply 메서드의 활용

1. 유사배열객체에 배열 메서드를 적용

   객체에는 배열 메서드를 직접 적용할 수 없습니다. 그러나 유사배열객체(키가 0 또는 양의 정수인 프로퍼티가 존재하고 length 프로퍼티의 값이 0 또는 양의 정수인 객체)는 `call` 또는 `apply` 메서드를 이용해서 배열 메서드를 차용할 수 있습니다.

```
// 유사배열객체
var obj = {
  0: 'a',
  1: 'b',
  2: 'c',
  length: 3
};

obj.push('d'); // Uncaught TypeError: obj.push is not a function

Array.prototype.push.call(obj, 'd');
console.log(obj); // { 0: 'a', 1: 'b', 2: 'c', 3: 'd', length: 4 }

var arr = Array.prototype.slice.call(obj); // slice: 원본 배열의 복사본 반환
console.log(arr); // [ 'a', 'b', 'c', 'd' ];


// arguments 객체 이용(함수에 전달된 인수에 해당하는 Array 형태의 객체)
function myConcat(separator) {
  var args = Array.prototype.slice.call(arguments, 1);
  // args == ['red', 'orange', 'blue']
  return args.join(separator);
}
console.log(myConcat(', ', 'red', 'orange', 'blue')); // "red, orange, blue"


// nodeList 이용
document.body.innerHTML = '<div>a</div><div>b</div><div>c</div>';
var nodeList = document.querySelectorAll('div');
var nodeArr = Array.prototype.slice.call(nodeList);
nodeArr.forEach(function (node) {
  console.log(node); // (1) 출력값 아래 사진 참고
});
```

(1) 출력 결과

문자열에 대해서도 `call` / `apply` 를 이용해 모든 배열 메서드를 적용할 수 있습니다. 단, 문자열의 length는 읽기 전용이기 때문에 원본 문자열에 변경을 가하는 메서드 (push, pop, shift, unshift, splice 등)는 에러를 던집니다.

```
var str = 'abc def';

// 문자열에 직접 함수 사용이 불가능하기 때문
var newArr2 = str.map(function (char) {
  return char + '?';
}); // Uncaught TypeError: str.map is not a function

var newArr = Array.prototype.map.call(str, function(char) {
return char + '!';
});
console.log(newArr); // ['a!', 'b!', 'c!', ' !', 'd!', 'e!', 'f!']

// length 프로퍼티 값이 바뀌는 함수는 불가!
Array.prototype.push.call(str, ' gh'); // Uncaught TypeError: Cannot assign to read only property 'length' of obje

// concat: 인자로 주어진 배열이나 값들을 기존 배열에 합쳐서 새 배열을 반환
Array.prototype.concat.call(str, 'string'); // [String {'abc def'}, 'string']

// every: 배열 안의 모든 요소가 주어진 판별 함수를 통과하는지 테스트합니다.
Array.prototype.every.call(str, function(char) {
  return char !== ' ';
}); // false

// some: 배열 안의 어떤 요소라도 주어진 판별 함수를 적어도 하나라도 통과하는지 테스트합니다.
Array.prototype,some.call(str, function(char) {
  return char !== ' ';
}); // true

var newStr = Array.prototype.reduce.apply(str, [
  function(string, char, i) {
    return string + char + i;
  },'']);
console.log(newStr); // 'a0b1c2 3d4e5f6'
```

이렇게 call/apply를 이용해서 형변환하는것은 '**this를 원하는 값으로 지정해서 호출한다**'는 본래 메서드 의도와는 동떨어졌으며 코드를 보고 의도를 파악하기가 어렵습니다. 그래서 ES6에서는 순회 가능한 모든 종류의 데이터 타입을 배열로 전환하는 `Array.from` 메서드를 새로 도입했습니다.

```
var obj = {
  0: 'a',
  1: 'b',
  2: 'c',
  length: 3
}
var arr = Array.from(obj);
console.log(arr); // ['a', 'b', 'c']
```

2. 생성자 내부에서 다른 생성자를 호출

   생성자 내부에 다른 생성자와 공통된 내용이 있을 때 call/apply를 이용해서 다른 생성자를 호출하면 간단하게 반복을 줄일 수 있습니다.

```
function Person(name, gender) {
  this.name = name;
  this.gender = gender;
}

function Student(name, gender, school) {
  Person.call(this, name, gender);
  this.school = school;
}

function Employee(name, gender, company) {
  Person.apply(this, [name, gender]);
  this.company = company;
}

// 생성자 함수 내부의 this에는 생성자 함수가 생성할 인스턴스가 바인딩된다
var by = new Student('보영', 'female', '단국대');
var jn = new Employee('재난', 'male', '구골');
var ps = new Person('가', '나');

console.log('by: ', by);
console.log('jn: ', jn);
console.log('ps: ', ps);
```

```
by:  ▶ Student {name: '보영', gender: 'female', school: '단국대'}
jn:  ▶ Employee {name: '재난', gender: 'male', company: '구골'}
ps:  ▶ Person {name: '가', gender: '나'}
```

3. 여러 인수를 묶어 하나의 배열로 전달하고 싶을 때 - apply 활용

```
// apply 사용
var numbers = [10, 20, 3, 16, 45];
var max = Math.max.apply(null, numbers);
var min = Math.min.apply(null, numbers);
console.log(max, min); // 45 3

// 전개구문
var numbers = [10, 20, 3, 16, 45];
var max = Math.max(...numbers);
var min = Math.min(...numbers);
console.log(max, min); // 45 3
```

이처럼 `call` / `apply` 메서드는 별도의 this를 바인딩하며 함수 또는 메서드를 실행하는 훌륭한 방법이지만 오히려 이로 인해 this를 예측하기 어렵게 만들어 코드 해석을 방해한다는 단점이 있습니다.

▼ bind 메서드

```
Function.prototype.bind(thisArg[, arg1[, arg2[, ...]]])
```

bind는 call과 비슷하지만 즉시 호출하지는 않고 넘겨받은 this 및 인수들을 바탕으로 새로운 함수를 반환하기만 하는 메서드입니다.

bind는 함수에 this를 미리 적용하는 것과 부분적용 함수를 구현하는 두가지의 목적을 지닙니다.

```
var function1 = function (a,b,c,d) {
  console.log(this,a,b,c,d);
};

// this만을 지정한 bind
var bindFunc1 = function1.bind({x:1});
bindFunc1(5,6,7,8); // {x:1} 5 6 7 8
bindFunc1(1,2,3,4) // {x:1} 1,2,3,4


// this지정과 부분 적용 함수를 구현한 bind
var bindFunc2 = function1.bind({x:1}, 4, 5);
bindFunc2(6,7); // {x:1} 4 5 6 7
```

- name 프로퍼티

bind 메서드를 적용해서 새로 만든 함수는 name 프로퍼티에 동사 bound라는 접두어가 붙습니다.

어떤 함수의 name 프로퍼티가 `bound [함수명]` 이라면 이는 [함수명]이 원본함수이며 이에 bind 메서드를 적용한 새로운 함수라는 의미이기 때문에 call/apply보다 코드를 추적하기에 더 수월합니다.

```
var func = function (a,b,c,d) {
  console.log(this,a,b,c,d);
};

var bindFunc1 = func.bind({ x: 1 });
console.log(func.name); // func
console.log(bindFunc1.name); // bound func
```

- 상위 컨텍스트의 this를 내부함수나 콜백 함수에 전달하기

  메서드의 this와 메서드 내부의 중첩 함수 또는 콜백 함수의 this가 불일치하는 문제를 해결하기 위해 bind가 유용하게 사용됩니다.

```
var obj = {
  outer: function () {
    console.log(this); // obj
    var innerFunc = function () {
      console.log(this); // window
    };
    innerFunc();
  }
}
obj.outer();


// bind
var obj = {
  outer: function () {
    console.log(this); // obj
    var innerFunc = function () {
      console.log(this); // obj
    }.bind(this);
    innerFunc();
  }
}
```

```
obj.outer();


// call
var obj = {
  outer: function () {
    console.log(this); // obj
    var innerFunc = function () {
      console.log(this); // obj
    };
    innerFunc.call(this);
  }
}
obj.outer();
```

또한 콜백 함수를 인자로 받는 함수나 메서드 중에서 기본적으로 '콜백 함수 내에서의 this'에 관여하는 함수 또는 메서드에 대해서도 bind 메서드를 이용하면 this값을 사용자 입맛에 맞게 바꿀 수 있습니다.

```
var obj = {
  logThis: function () {
    console.log(this);
  },
  logThisLater1: function () {
    setTimeout(this.logThis, 500);
  },
  logThisLater2: function () {
    setTimeout(this.logThis.bind(this), 100);
  }
};

obj.logThis(); // obj
obj.logThisLater1(); // Window
obj.logThisLater2(); // obj
```

`setTimeout()` 이 실행하는 코드는 `setTimeout()` 을 호출했던 함수와는 다른 실행 맥락에서 호출됩니다. 호출 함수의 `this` 키워드 값을 설정하는 일반적인 규칙이 여기서도 적용되며, `this` 를 호출 시 지정하지도 않았고 `bind` 로 바인딩하지도 않은 경우 기본 값인 `window` (또는 `global`) 객체를 가리키게 됩니다. 따라서 `setTimeout()` 을 호출한 함수의 `this` 값과는 다르게 되는 것입니다.

다음 코드를 살펴보세요.

```
const myArray = ['zero', 'one', 'two'];
myArray.myMethod = function (sProperty) {
  console.log(arguments.length > 0 ? this[sProperty] : this);
};

myArray.myMethod();  // "zero,one,two" 기록
myArray.myMethod(1); // "one" 기록
```

위의 코드는 `myMethod` 를 호출할 때, 호출로 인해 `this` 가 `myArray` 로 설정되기 때문에 정상적으로 동작합니다. `this[sProperty]` 가 `myArray[sProperty]` 와 동일함을 확인하세요. 그러나, 다음의 코드도 살펴보세요.

```
setTimeout(myArray.myMethod, 1.0*1000);      // 1초 후 "[object Window]" 기록
setTimeout(myArray.myMethod, 1.5*1000, '1'); // 1.5초 후 "undefined" 기록
```

`myArray.myMethod` 를 `setTimeout` 에 전달했고, 타이머 만료 후 호출 시점에 `this` 가 따로 설정되지 않으므로 기본 값인 `window` 객체를 가리키게 돼 정상적인 동작을 하지 않습니다.

▼ 화살표 함수의 예외 사항

ES6에 새롭게 도입된 화살표 함수는 실행 컨텍스트 생성 시 this를 바인딩하는 과정이 제외 됐기 때문에 함수 내부에는 this가 아예 없으며, 접근하고자 하면 스코프체인상 가장 가까운 this에 접근하게 됩니다.

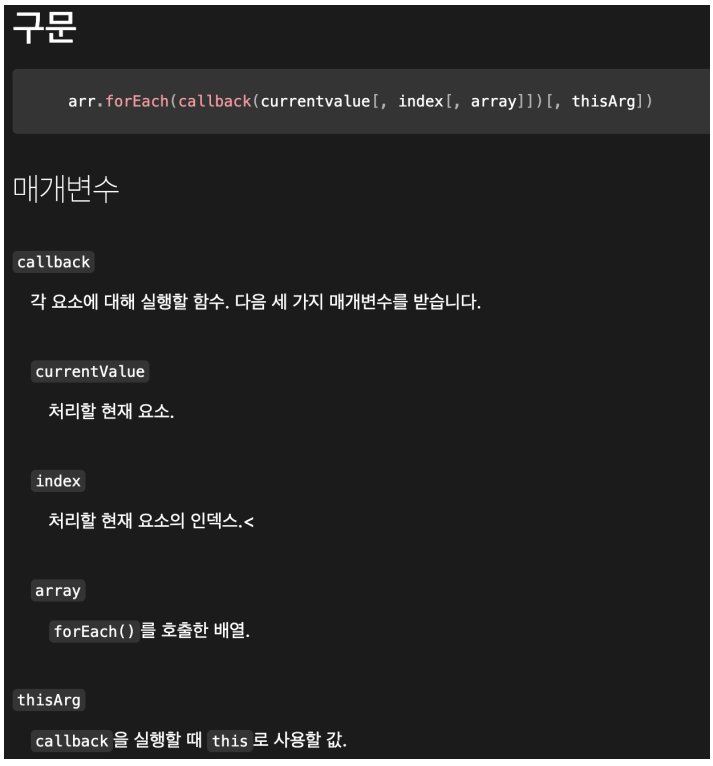
```
// 함수 선언식
var obj = {
  outer: function () {
    console.log(this); // obj
    function innerFunc() {
      console.log(this); // window
    };
    innerFunc();
  },
};
obj.outer();


// 화살표 함수
var obj2 = {
  outer: function () {
    console.log(this); // obj2
    var innerFunc = () => {
      console.log(this); // obj2
    }
    innerFunc();
  },
};
obj2.outer();
```

▼ 별도의 인자로 this를 받는 경우(콜백함수 내에서의 this)

콜백함수를 인자로 받는 메서드 중 일부는 추가로 this로 지정할 객체를 인자로 지정할 수 있습니다.

이러한 형태는 내부 요소에 대해 같은 동작을 반복 수행해야 하는 배열 매서드에 많이 포진되어 있으며, 같은 이유로 Set, Map 등의 메서드에서도 일부 존재합니다.



```
// 비교시 vscode
var report = {
  sum: 0,
  count: 0,
  add: function () {
    var args = Array.prototype.slice.call(arguments);
    args.forEach(function (entry) {
      this.sum += entry;
      ++this.count;
    }, this);
  },
  average: function () {
    return this.sum / this.count;
  }
};
report.add(60, 85, 95);
console.log(report.sum, report.count, report.average()); // 240 3 80
```

이 외에도 thisArg를 인자로 받는 메서드는

```
Array.prototype.map(callback[, thisArg])
Array.prototype.filter(callback[, thisArg])
Array.prototype.some(callback[, thisArg])
Array.prototype.every(callback[, thisArg])
Array.prototype.find(callback[, thisArg])
Array.prototype.findIndex(callback[, thisArg])
Array.prototype.flatMap(callback[, thisArg])
Array.prototype.from(arrayLike[, callback[, thisArg]])
Set.prototype.forEach(callback[, thisArg])
Map.prototype.forEach(callback[, thisArg])
```

🔥마지막 정리🔥

- **전역 공간에서의 this**는 전역객체(브라우저에서는 window, Node.js에서는 global)을 참조합니다.
- 어떤 함수를 **메서드로 호출한 경우 this**는 메서드 호출 주체(메서드명 앞의 객체)를 참조합니다.
- 어떤 함수를 **함수로서 호출한 경우 this**는 전역객체를 참조합니다. 메서드 내부에서도 동일합니다.
- **콜백함수 내부에서의 this**는 해당 콜백함수의 제어권을 넘겨받은 함수가 정의한 바에 따르며, 정의하지 않은 경우에는 전역객체를 참조합니다.
- **생성자 함수에서의 this**는 생성될 인스턴스를 참조합니다.

- **call, apply** 메서드는 this를 명시적으로 지정하면서 함수 또는 메서드를 호출합니다
- **bind** 메서드는 this 및 함수에 넘길 인수를 일부 지정해서 새로운 함수를 만듭니다.
- 요소를 순회하면서 콜백 함수를 반복 호출하는 내용의 일부 메서드는 별도의 인자로 this를 받기도 합니다.