2023.03.05 보충(비동기 프로그래밍, this)

B: 내 담당 아님

1. 비동기(**Asynchronous**) 프로그래밍
    a. 비동기란, 둘 이상의 객체가 동시에 존재하지 않거나, 둘 이상의 이벤트가 동시에 발생하지 않는 것을 의미합니다. 프로그래밍에서는 두 가지 의미로 사용됩니다.
        i. 네트워크 및 통신: 동기는 통신을 즉시 주고받는다는 점에서 전화, 비동기는 통신을 즉시 주고받을 필요가 없다는 점에서 이메일에 비유할 수 있습니다. 즉, 비동기적 통신이라 함은, 다른 프로그램(서버 등)에 데이터를 요청하고 응답을 기다리면서 다른 작업을 할 수 있습니다. (AJAX 등)
        ii. 소프트웨어 설계: 작업이 완료되는 것을 기다릴 필요 없이, 기존의 작업과 함께 처리되도록 요청할 수 있게 코드를 작성하는 것입니다.
    b. 동기 프로그래밍 vs 비동기 프로그래밍
        i. 동기 프로그래밍은 프로그램을 **작성한 순서대로 실행**합니다. 즉, 기존의 작업이 완료되는 것을 기다렸다가 다음 작업을 진행합니다. 따라서, 비효율적인 로직이 포함된 코드라면 작업이 모두 완료될 때까지 오랜 시간이 걸릴 수 있고, 그동안 프로그램이 응답하지 않을 것(**Bloking**)입니다.
        ii. 비동기 프로그래밍은 **시간이 오래 걸리는 작업을 시작**하고, 이 작업이 수행되는 동안 기존의 코드로 복귀하여 **다른 이벤트에 응답**합니다. 시간이 오래 걸리는 작업이 끝나면 결과를 알려 줍니다.
    c. Javascript에서 비동기 프로그래밍이 구현되는 방법
        i. **이벤트 핸들러**: 이벤트가 발생할 때마다 호출되는 핸들러를 제공함으로써 비동기 프로그래밍을 구현합니다. '비동기 작업 완료' 이벤트가 발생하면 이를 사용하여 호출 결과를 알릴 수 있습니다.
            1. `XMLHttpRequest`
                a. JavaScript에서 원격 서버에 HTTP 요청을 할 수 있는 API로, `XMLHttpRequest` 객체에 연결해 진행 상태 및 완료에 대한 알림을 수신합니다.
                b. 아래 예시에서는, 순차적으로 프로그램이 시작되어 `XMLHttpRequest` 객체에 GET 요청을 보냅니다. 이후 'start'가 먼저 출력되고,

`XMLHttpRequest` 객체로부터 응답을 받으면 `loadend` 이벤트가 실행되어 ‘done’이 출력됩니다.

```
const xhr = new XMLHttpRequest();
xhr.addEventListener('loadend', () => {
  console.log('done');
});

xhr.open('GET', 'url');
xhr.send();
console.log('start');
```

ii. **Callback**

1. Callback은 적절한 시점에 호출될 것으로 예상하여 다른 함수로 전달되는 함수입니다(이벤트 핸들러는 Callback의 한 종류입니다). JavaScript에서 비동기 함수를 구현하는 주요 방식이었으나, Callback이 깊게 중첩되면 가독성이 낮아지는 문제점이 발생했습니다(Callback Hell).

iii. **Promise**

1. Promise는 비동기 작업이 맞이할 미래의 완료 또는 실패와 그 결과를 나타냅니다.
2. Promise는 `pending`, `fulfilled`, `rejected` 중 하나의 상태를 가집니다. `pending`은 이행되기 전의 초기 상태이며 `fulfilled`는 성공, `rejected`는 실패한 상태로, 각각 `then`과 `catch`를 통해 이후의 작업을 처리할 수 있습니다. `fulfilled` 또는 `rejected` 상태가 되면 안정된 또는 처리된 (`settled`) 상태가 되었다고 합니다.

iv. **Async Function**

1. async function은 Promise를 반환하는 보다 편리한 방법입니다.
2. 직접 Promise를 반환하지 않아도 async function을 만들고, 비동기 작업이 필요한 부분에 `await` 키워드를 사용하면 해당 부분에서 Promise가 반환됩니다.

v. **Generator**

1. 실행 도중에 함수의 제어권을 외부로 양도(`yield`)할 수 있는 함수입니다.
2. `function*` 키워드를 통해 선언할 수 있고, Iterator 객체를 반환합니다.
    a. Iterator 객체는 next 메서드를 가집니다.

i. next가 실행될 때마다 `yield` 전까지 제너레이터 함수가 실행되고, `yield` 키워드를 만나면 함수 외부의호출자에게 { value, done } 객체와 함께 제어권을 양도합니다.

ii. { value, done }에는 각각 `yield`를 통해 전달하는 값과, generator 함수의 종료 여부가 할당됩니다.

vi. **Web Worker**

1. 스크립트의 연산을 주 실행 스레드와 분리된 별도의 백그라운드 스레드에서 실행하는 기술입니다.
2. window와 다른 Global 맥락에서 실행됩니다. DOM을 직접 조작하거나, window의 일부 method, property를 사용할 수 없으나 대부분은 사용 가능합니다.

2. This

a. this는 프로그래밍 언어에서 해당 this가 포함된 객체를 가리킵니다. JavaScript에서도 기본적으로 이는 마찬가지입니다. 그러나 JavaScript에서 this는 기본적으로 Lexical하지 않습니다. 즉, this가 가리키는 객체가, **코드가 작성된(Lexical) 순간이 아닌, 문맥(context)에 따라 결정**된다는 의미입니다.

```
const test = {
  prop: 42,
  func: function() {
    return this.prop;
  },
};
console.log(test.func()); // 42
```

b. 전역 문맥에서의 this는 this가 호출된 전역 객체 window를 가리킵니다.

c. 함수 문맥에서의 this는 함수가 호출되는 방식에 따라 다른 객체를 가리킵니다.

i. 단순 호출

1. 아래 코드에서 this가 호출되는 것은 fn함수의 return 시점입니다. 즉, this가 가리키는 대상이 fn 함수 내부에서 결정되지 않고, 호출된 시점인 전역 문맥에서 결정된 것입니다.

```
function fn() { return this; }
console.log(fn()) // window
```

ii. 엄격 모드(strict mode)에서 호출

1. 엄격 모드에서는 lexical하게 동작하므로, fn 함수 내에서 this값을 지정하지 않은 것으로 되어 undefined를 반환합니다.

```
function fn() {
  'use strict';
  return this;
}
console.log(fn()) // undefined
```

d. call, apply, bind, arrow function

i. call, apply

1. call과 apply는 this의 값을 다른 문맥으로 넘겨 주는 역할을 합니다.
2. 아래 코드에서, checkThis 함수의 this는 호출 시점인 전역 문맥에 따라 'global hello'를 반환합니다. 그러나, call 또는 apply를 사용하면 checkThis 함수의 this를 첫 번째 파라미터인 obj로 설정하기 때문에, ‘hello’를 반환합니다.

```
const obj = { a: 'hello' };
const a = 'global hello';

function checkThis() {
  return this.a;
}
checkThis(); // global hello
checkThis.call(obj); // 'hello'
checkThis.apply(obj); // 'hello'
```

ii. bind

1. bind는 this가 lexical하게 (첫 번째 파라미터로 넘겨 받은 객체로) 고정된 동일한 내용의 새로운 함수를 반환합니다.

```
const obj = { a: 'hello' };
const a = 'global hello';

function checkThis() {
  return this.a;
}
checkThis(); // global hello
const binded = checkThis.bind(obj);
console.log(binded()); // 'hello'
```

iii. arrow function

1. **화살표 함수에서 this는 lexical**하게 작동합니다. 즉, 호출 시점이 아닌 코드의 작성 시점에 이미 자신을 감싼 정적 범위로 this가 결정되어 있으며, 따라서 전역 코드에서 화살표 함수가 정의되었다면 this 또한 전역 객체를 의미합니다.

e. 객체 문맥에서의 this는 해당 객체를 가리킵니다.

i. 객체의 프로토타입 체인 내에서 사용된 this 역시 프로토타입 객체가 아닌 해당 객체를 가리킵니다.

ii. `getter`와 `setter`에서도 마찬가지로 해당 객체를 가리킵니다.

iii. 생성자에서도 마찬가지로 해당 객체(새로 생성되는 객체)를 가리킵니다.

f. 이벤트 핸들러 문맥에서의 this는 이벤트를 발생시키는 요소를 가리킵니다.

i. 즉, `this === event.currentTarget`입니다.

ii. 이벤트가 전파될 경우, event.target은 event.currentTarget과 달라질 수 있기 때문에, `this === event.target`은 True가 아닌 경우도 있습니다.