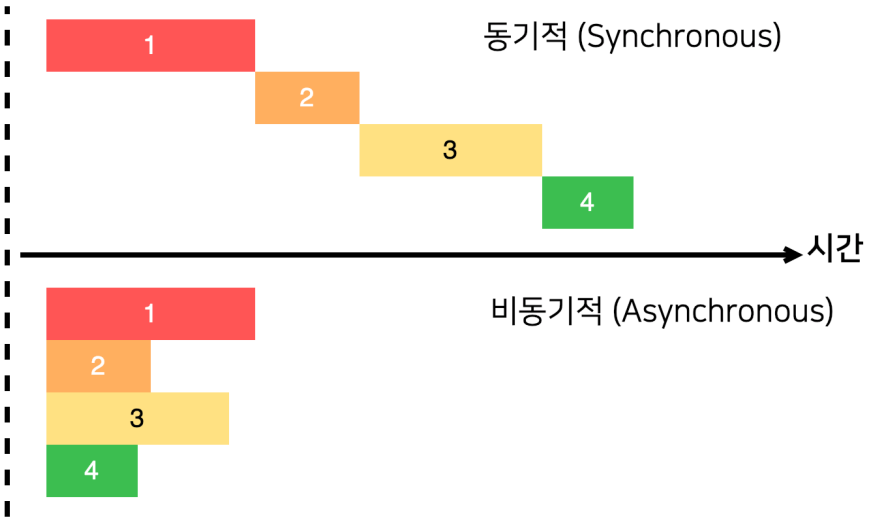
자바 스크립트 심화

1. 비동기 프로그래밍이란?

일단 일반적으로 자바스크립트는 동기적인 언어입니다. 한가지 작업이 다 끝나야만 다음 작업으로 넘어갈 수 있다는 뜻이죠.

그러나 이런 동기적인 방식이 항상 좋은 것은 아닙니다. 오히려 문제가 생기는 경우도 있는데요, 예를 들어 네트워크에서 데이터를 가져오는 등의 작업은 시간이 많이 걸리기도 합니다. 근데 이 작업이 수행되는 동안 자바스크립트는 다른 작업을 수행하지 못하게 되죠.



그러나 위의 이미지처럼 데이터를 가져오는 동안 다른 작업을 같이 하게 된다면 훨씬 일을 빠르게 마칠수도 있습니다.

setTimeout 함수

동기, 비동기에 대한 예시를 한번 보겠습니다.

자바스크립트에서 `setTimeout`은 비동기 프로그래밍의 기본 예시 중 하나입니다.

`setTimeout`은 주어진 지연 시간이 경과한 후에 특정 코드나 함수를 실행하도록 예약하는데 사용됩니다. 쉽게 말해, 몇 초 뒤에 어떤 코드를 실행하고 싶을 때 사용할 수 있죠.

```
console.log(1);
console.log(2);
console.log(3);
console.log(4);
```

```
1
2
3
4
```

우선, 위 코드를 실행하면 오른쪽과 같이 로그가 나올 겁니다. 코드 순서대로 실행되는 것을 확인할 수 있습니다.

```
console.log(1);
console.log(2);
setTimeout(() => {console.log(3)}, 5000); // 5000은 5초와 같습니다.
console.log(4);
```

```
1
2
4
undefined
3
```

위 코드에서 `setTimeout`을 사용한 코드는 '5초 뒤에 숫자 3을 출력해' 라는 의미를 지니고 있습니다. 만약 이 코드가 '동기적'으로 동작한다면 2가 출력되고, 5초뒤에 3이 출력, 마지막으로 4가 출력될 것입니다.

그러나 앞에서 말했다시피 `setTimeout`은 비동기로 작동하기 때문에 코드 순서와 상관 없이 실행과 동시에 작동합니다. 따라서 먼저 기다림 없이 출력될 수 있는 1,2,4가 먼저 출력되고 5초 후에 3이 마지막으로 출력되는 결과가 나타나는 것이죠.

비동기 너무 어색하지 않나요? 더 효율적인 작업이 가능하다는 건 이해가 가지만 위의 예시처럼 난 5초 기다리는걸 명령하고 싶은데!! 코드는 마음대로 비동기로 처리해버릴 수 있다는 게 처음에는 황당했어요... 특히 파이썬에 익숙하신 분들이라면 더더욱 어려운 개념일 수도 있습니다.

아마 저와 비슷한 사람이 많았는지, 이렇게 원치 않는 비동기 처리를 쉽게 다룰 수 있는 방법이 자바스크립트에는 있습니다! 대표적으로 `콜백(callback), Promise, async/await` 입니다.

2. Callback

콜백 함수의 동작 방식을 쉽게 비유하자면, 식당 예약과 같습니다. 맛집을 가게 되면 사람이 많아 자리가 없습니다. 그래서 대기자 명단에 이름을 쓴 다음 자리가 날 때까지 주변을 돌아다니죠. 만약 식당에 자리가 생기면 전화로 자리가 났다고 연락이 옵니다. 그 전화를 받는 시점이 여기서 콜백 함수가 호출되는 시점과 같습니다. 손님 입장에서는 자리가 날 때까지 근처 가게에서 쇼핑을 할 수도 있고, 다른 식당 자리를 알아볼 수도 있습니다.

자리가 났을 때만 연락이 오기 때문에 미리 가서 기다릴 필요도 없고, 직접 식당 안에 들어가서 자리가 비어 있는지 확인할 필요도 없습니다. 자리가 준비된 시점, 즉 데이터가 준비된 시점에서만 저희가 원하는 동작을 수행할 수 있습니다.

콜백 지옥(Callback Hell)

콜백 지옥은 비동기 처리 로직을 위해 콜백 함수를 연속해서 사용할 때 발생하는 문제입니다. 아래와 같은 코드를 살펴보겠습니다.

```
$.get('url', function(response) {  ///url에서 데이터를 가져옴
    parseValue(response, function(id) {   ///데이터를 파싱해서 id 추출
        auth(id, function(result) {    ///id를 이용해 인증 수행(auth)
            display(result, function(text) {    ///인증 결과를 이용해 최종 텍스트 생성(dis
                console.log(text);    /// 최종 결과를 콘솔에 출력
            });
        });
    });
});
```

앞에서 언급했다시피 웹 서비스를 개발할 때는 서버에서 데이터를 받아와 화면에 표시하기까지 인코딩, 사용자 인증 등 처리해야하는 경우가 있습니다. 만약 이 모든 과정을 비동기로 처리해야 한다면 위와 같이 콜백안에 콜백을 계속 무는 형식으로 코딩을 하게 됩니다. 이러한 코드 구조는 가독성도 떨어지고 로직을 변경하기도 어렵습니다. 그래서 이러한 코드 구조를 흔히 콜백 지옥이라고 칭합니다.

이런 콜백 지옥을 단순히 패턴만 바꿔서 해결하고자 한다면 각 콜백 함수를 아래와 같이 분리해주면 됩니다.

```
function parseValueDone(id) {
  auth(id, authDone);
}
function authDone(result) {
  display(result, displayDone);
}
function displayDone(text) {
  console.log(text);
}
$.get('url', function(response) {
  parseValue(response, parseValueDone);
});
```

그러나 이런 방식보다 훨씬 편하게 구현할 수 있는 방법이 존재합니다. 실제로도 콜백 함수는 쉽게 가독성이 떨어져 버릴 수 있는 문제로 최근에는 잘 사용하지 않습니다. 이를 해결하는 방법이 앞서 언급했던 `Promise, async(await)` 입니다.

3. Promise

Promise

promise는 자바스크립트에서 비동기 작업을 더 편리하게 다루기 위한 객체입니다.

promise는 주로 서버에서 받아온 데이터를 화면에 표시할 때 사용합니다. 일반적으로 웹을 구현할 때 우리는 서버에서 데이터를 요청하고 받아오기 위해 API라는 것을 사용합니다. (API에 대한 구체적인 설명은 이후에 배우게 될 예정입니다.) API가 실행되면 서버에 '~~이런 데이터를 하나 보내주세요' 라고 요청을 보내게 됩니다. 그런데 여기서 데이터를 받아오는 건 시간이 조금 걸립니다. 그러나 데이터를 받아오기도 전에 마치 데이터를 다 받아온 것 마냥 화면에 데이터를 표시하려고 하면 오류가 발생하거나 빈 화면이 뜨겠죠?

이와 같은 문제점을 해결하기 위한 방법 중 하나가 프로미스 입니다.

promise의 세가지 상태

프로미스를 사용할 때 알아야 하는 가장 기본적인 개념이 바로 프로미스의 상태입니다. 여기서 말하는 상태란 프로미스의 처리 과정을 의미합니다. 프로미스를 생성하고 종료될 때까지 3가지 상태를 가지게 되는데요

- pending(대기): 비동기 처리 로직이 아직 완료되지 않은 상태
- fulfilled(이행): 비동기 처리가 완료되어 프로미스가 결과 값을 반환해준 상태
- Rejected(실패): 비동기 처리가 실패하거나 오류가 발생한 상태

즉, 비동기 처리가 완료 되지 않았다면 `Pending`, 완료 되었다면 `Fulfilled`, 실패하거나 오류가 발생하였다면 `Rejected` 상태를 갖습니다.

promise 사용법

우선 아래와 같이 new Promise() 메서드를 호출하면 대기(pending) 상태가 됩니다.

```
new Promise()
```

new Promise() 메서드를 호출할 때 콜백함수를 선언할 수 있고, 콜백 함수의 인자는 resolve, reject입니다.

```
const condition = true;
const promise = new Promise((resolve, reject) => {
    // 보통 여기에 비동기 함수를 넣습니다!
  if (condition) {
    resolve('resolved'); // condition이 참
  } else {
    reject('rejected'); // condition이 거짓
  }
});
```

예시에서는 condition 값에 따라 promise 반환 값이 결정되고 있습니다. 값이 참이면 resolve를 호출하고, 아니라면 reject를 호출합니다.

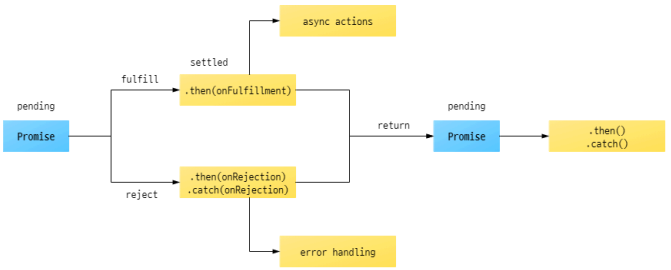
```
promise
  .then((res) => {
    console.log(res);
```

```
  })
  .catch((error) => {
    console.error(error);
  });
```

또한 위의 예시처럼 `then()` 과 `catch()` 를 통해 비동기 로직의 성공 여부에 따라 다르게 처리가 가능합니다.

- `resolve` 함수 호출 → `then()` 실행
- `reject` 함수 호출 → `catch()` 실행

- **resolve 함수**
  1. 이 함수가 호출되면 Promise의 상태는 "Pending"에서 "Fulfilled"로 변경됩니다.
  2. resolve 함수는 비동기 작업의 결과를 인자로 받아, 이를 Promise 객체에 전달합니다.
  3. resolve 함수가 호출되면 Promise에 등록된 "then" 콜백 함수가 실행됩니다.
- **reject 함수**
  1. 이 함수가 호출되면 Promise의 상태는 "Pending"에서 "Rejected"로 변경됩니다.
  2. reject 함수는 비동기 작업에서 발생한 오류를 인자로 받아, 이를 Promise 객체에 전달합니다.
  3. reject 함수가 호출되면 Promise에 등록된 "catch" 콜백 함수가 실행됩니다.



최종 코드

```
const condition = true;
const promise = new Promise((resolve, reject) => {
  // 보통 여기에 비동기 함수를 넣습니다!
```

```
  if (condition) {
    resolve('resolved'); // condition이 참
  } else {
    reject('rejected'); // condition이 거짓
  }
});

promise
  .then((res) => {
    console.log(res);
  })
  .catch((error) => {
    console.error(error);
  });
```

4. async/await

`promise`가 편리해진거 같지만 아쉽게도 `promise` 또한 `then() 지옥`이라는 것이 빠질 수 있습니다.

```
const userInfo = {
  id: 'likelion@google.com',
  pw: '****'
};

function parseValue() {
  return new Promise({
    // ...
  });
}
function auth() {
  return new Promise({
    // ...
  });
}
function display() {
  return new Promise({
```

```
    // ...
  });
}

getData(userInfo)
  .then(parseValue)
  .then(auth)
  .then(diaplay);
```

앞서 콜백함수 지옥을 설명할 때 사용 했던 코드를 promise를 이용해 보면 이처럼 then()를 이용하여 길어질 수 있습니다. 그래서 `async/await`가 등장하게 됩니다.

`async/await` 는 가장 최근의 나온 비동기 처리 문법으로, 기존의 `callback` 이나 `Promise` 의 단점을 해소하고자 만들어졌습니다.

async/await 기본 문법

```
const 함수명 = async () => {
  await 비동기_처리_메서드_명();
};
```

기본 형태를 위와 같습니다. 함수를 호출하면 비동기 처리 메서드가 `await`를 기다렸다가 실행될 것입니다. 한 가지 주의할 점은 비동기 처리 메서드(Async)가 항상 `promise 객체`를 반환해야 `await`가 의도한대로 동작한다는 것입니다.

await는 async 함수 안에서만 동작합니다. 즉 async 함수가 아닌데 await를 사용하면 문법 에러가 발생할 겁니다. 또한 `'변수 = await promise'`인 형태에서는 프로미스가 resolve된 값이 변수에 저장됩니다. 만약 `'변수 = await 값'`인 경우 그 값이 변수에 저장됩니다.

async/await 간단 예제

```
const fetchItems = () => {
  return new Promise((resolve, reject) => {
    let items = [1, 2, 3];
    resolve(items);
  });
};
```

```
const logItems = async () => {
  let resultItems = await fetchItems();
  console.log(resultItems); // [1,2,3]
};

logItems();
```

먼저 `fetchItems()` 함수는 프로미스 객체를 반환하는 함수입니다. 프로미스는 '자바크스립트 비동기 처리를 위한 객체'라고 배웠었죠. fetchItems() 함수를 실행하면 프로미스가 resolve 되며 결과 값은 `items` 배열이 됩니다.

그리고 이제 `logItems()` 함수를 보겠습니다. logItems() 함수를 실행하면 fetchItems() 함수의 결과 값인 items 배열이 `resultItems` 변수에 담깁니다. 따라서, 콘솔에는 [1,2,3]이 출력되죠.

`await`를 사용하지 않았다면 데이터를 받아온 시점에 콘솔을 출력할 수 있게 콜백 함수나 .then()등을 사용해야 했을 겁니다. 하지만 async await 문법 덕택에 비동기에 대한 사고를 하지 않아도 되는 것 입니다.

5. Try / Catch

지금까지 비동기 처리에 대한 정말 많은 예시들을 살펴봤는데요. 이해가 잘 안 가실수도 있습니다... 사실 자바스트립트를 처음 배우는 시점에서 비동기 처리를 한번에 완벽히 이해한다는건 쉽지 않아요. 앞선 내용들의 개념을 제대로 못 이해했다 하더라도! 앞으로 나올 `try/catch` 코드 하나만 건지고 가면 좋겠습니다!

아래의 코드가 실제로 저희가 자주 쓰는 코드의 형태입니다. 이것만 배우고 가셔도 일단은 충분합니다.

```
const fetchData = async () => {
  try {
    const response = await axios.get('https://api.example.com/data');
    ///url의 api 엔드 포인트로 get 요청을 보낸 후 await를 사용하여 응답을 받을 때까지 
    const data = response.data;  ///실제 API에서 받아온 데이터를 추출 후 data로 지
    console.log(data);
  } catch (error) {    ///API 요청이 실패하면 어떤 에러인지 출력
```

```
    console.error('Error:', error);
  }
}

fetchData();
```

비동기 작업 중 발생할 수 있는 에러를 처리하기 위해 `try/catch` 문을 사용할 수 있습니다.

`try/catch`는 `async/await`와 함께 사용하여 `Promise`가 rejected 상태가 되었을 때 발생하는 에러를 잡아낼 수 있습니다.

- **try**: 이 블록에는 에러가 발생할 가능성이 있는 코드를 넣습니다.
- **catch**: 이 블록은 `try` 블록에서 에러가 발생했을 때 실행됩니다. 에러 객체를 인자로 받아 이를 처리할 수 있습니다.

6. 이벤트 다루기 (Event)

이제부턴 조금 다른 이야기를 해볼 예정입니다. ~~(지긋지긋했던 비동기에서 벗어나)~~

이벤트(Event) 라는 것에 대해서 배워 볼건데요. 이벤트는 웹 브라우저와 사용자 사이에 상호작용이 발생하는 특정 시점을 의미합니다. 여기서 말하는 상호작용에는 마우스 클릭, 키보드 입력, 스크롤 등과 같은 행위를 말합니다.

이벤트가 발생하면 이벤트 종류에 따라 어떤 작업을 하거나 미리 등록한 함수를 호출하는 등의 조작을 자바스크립트로 지정할 수 있습니다.

웹 브라우저에서 사용자와의 상호작용으로 발생하는 이벤트는 200여가지가 넘습니다... 모든 이벤트를 다루기란 사실상 불가능이겠죠? 그래서 저희는 가장 많이 사용하는 중요 이벤트 몇가지만 다루겠습니다.

이벤트 처리기

먼저 그 전에 이벤트 처리기에 대해 짚고 가겠습니다. 이벤트 처리기라는 것은 이벤트가 발생했을 때 이에 반응하여 수행할 동작이 작성된 함수에 연결되도록 하는 처리기 입니다. 즉 쉽게 말해 해당 이벤트가 발생할 때 이에 대응하여 동작하는 `콜백함수`를 말합니다.

이벤트 처리기는 두 종류가 있는데요, 하나는 이벤트 핸들러, 또 하나는 이벤트 리스너입니다. 두가지 모두 사용자가 어떤 동작을 했을 때 실행할 코드를 지정해주는 방법입니다. 그러나 가장 큰 차이점은 하나의 요소에 이벤트를 여러개 등록할 수 있는지 입니다. 이벤트 핸들러는 한가지 이벤트 밖에 등록하지 못합니다. 따라서 이미 이벤트가 있는 요소에 새로운 이벤트를 등록하면 새 이벤트가 기존 것을 덮어버립니다. 그러나 이벤트 리스너는 여러개의 이벤트를 처리할 수 있죠.

```
const btn = document.getElementById('btn');

btn.onclick = () => { ... }

btn.addEventListener('click', () => { ... });
btn.addEventListener('hover', () => { ... });
btn.addEventListener('mouserdown', () => { ... });
```

위의 예시를 보면 한눈에 들어오죠??

	eventHandler	addEventlistener()
형태	속성(HTML attribute or JS property)	메서드(Method)
이벤트 중복 등록	불가능(새 이벤트가 기존 것을 덮음)	가능(새 이벤트가추가됨)
이벤트 제거 여부	불가능	removeEventListener()로 제거 가능
기능 확장성	제한적	더 유연하고 확장 가능

그럼 아까 말했던 수 많은 이벤트들을 간략하게 중요한 것만 보겠습니다.

이벤트 핸들러 속성 종류

마우스 이벤트	onclick	마우스로 클릭하면 발생
	onmouseover	마우스 포인터를 올리면 발생
	onwheel	마우스 휠을 움직이면 발생
키보드 이벤트	onkeypress	키보드 버튼을 누르고 있는 동안 발생
	onkeydown	키보드 버튼을 누른 순간 발생

	onkeyup	키보드 버튼을 눌렀다가 뗀 순간 발생
포커스 이벤트	onfocus	요소에 포커스가 되면 발생
	onchange	요소의 값이 바뀌었을 때 발생
폼 이벤트	onselect	텍스트 필드 등의 텍스트를 선택했을 때 발생
	onsubmit	폼이 전송될 때 발생

이벤트 핸들러 - 인라인방식

자바 스크립트에서는 이벤트를 다룰 때 `이벤트 핸들러`를 사용하는데, 이벤트 핸들러는 '이벤트가 발생 했을 때 실행되는 함수'를 말합니다.

인라인 방식은 HTML 요소의 속성으로 직접 이벤트 핸들러를 할당하는 방식입니다.

```
<button onclick="clickEvent()">버튼</button>

<script>
   function clickEvent(){
   alert("click");
   }
</script>
```

이렇게 작성하면, ' 버튼'이 클릭되었을 때 clickEvent()라는 함수가 실행됩니다.

이런 인라인 방식, 이벤트 핸들러의 단점은 HTML와 자바스크립트를 뒤죽박죽 섞어서 사용하기 때문에 분석하기 어려워집니다. 또한 한번에 하나의 이벤트 핸들러만 등록될 수 있다는 문제가 있죠.

이벤트 핸들러 - 비인라인방식

```
let btn = document.getElementById('myBtn');
/// id="myBtn"을 가진 버튼 요소를 JavaScript에서 가져와 btn 변수에 저장.

// 이벤트 핸들러 프로퍼티에 함수 지
btn.onclick = function() {
///onclick 프로퍼티에 익명 함수를 할당하여 클릭 이벤트를 처리.
```

```
    alert('clicked!');
}
```

이벤트 리스너

이벤트 리스너란 자바스크립트 코드에서 이벤트를 동적으로 처리하는 방식을 말합니다.

addEventListener 메소드를 이용해 이벤트를 대상 요소에 연결하고, 이에 따라 실행될 함수(이벤트 핸들러)를 전달합니다.

```
let btn = document.getElementById('myBtn');
/// id="myBtn"을 가진 버튼 요소를 JavaScript에서 가져와 btn 변수에 저장.

// addEventListener 메서드 사용
btn.addEventListener('click', function() {
  alert('clicked!');
});
```

위의 코드는 버튼 클릭 이벤트를 두 가지 방식(이벤트 핸들러 vs 이벤트 리스너)로 처리하는 방법을 보여드렸습니다