Promise, async & await

비동기는 왜 사용해야 하는가?

- 코드의 정확한 실행 시점을 알 수 없는 경우 (사용자의 입력: 마우스, 인풋 이벤트 등등)
- 처리에 시간이 많이 걸리는 동작의 경우 (네트워크 요청, 파일의 입출력 등등)

자바스크립트는 싱글쓰레드로 동작하기 때문에 위와 같은 처리가 필요한 경우, 처리되는 동안 다른 동작을 할 수가 없게됩니다. 우리는 일상생활에서 비동기식 행동을 자주합니다. 항상 어떻게 하면 시간을 아끼면서 여러가지 일을 동시에 처리할 수 있을까 고민하게 됩니다.

예를 들어 만약

1. 컵라면을 먹는다

2. 드라마를 본다.

이 두 가지 행동을 처리한다고 할 때,

컵라면을 다 끓이고 드라마를 보는게 처리 시간에 있어서 효율적일까요?

1. 컵라면 물을 끓인다. (+3분)
2. 물이 끓으면 컵라면을 까서 스프를 넣고 뚜껑을 닫는다.(+1분)
3. 3분뒤 컵라면을 먹는다.(+3분)
4. 티비로 가서 보고 싶은 드라마를 고른다.(+1분)
5. 드라마를 결정하고 재생한다.(+1분)
6. 드라마를 보면서 먹는다.(종료 : 총 9분)

두 가지를 동시에 진행하는게 효과적일까요?

1. 컵라면 물을 끓인다. + 2) 티비로 가서 보고 싶은 드라마를 고른다.(+3분)
2. 물이 끓으면 컵라면을 까서 스프를 넣고 뚜껑을 닫는다.(+1분)
3. 드라마를 결정하고 재생한다. + 5) 3분뒤 컵라면을 먹는다. (+3분)
4. 드라마를 보면서 먹는다.(종료 : 총 7분)

결과적으로 2분 더 빨리 두 가지 일을 처리할 수 있습니다. 또한 4분 동안 다른 일을 할 수 있는 여력이 생깁니다. 따라서 모든 일은 동기적으로 처리가 가능하지만, 비동기적으로 처리할 수 있다면 더 빠르고 많은 일을 할 수 있습니다.

왜 비동기는 어려울까?

위의 사례를 생각해 봤을 때

'과연 우리가 컵라면을 처음 끓여보고, OTT를 처음 사용해 본다면 비동기식으로 일을 처리할 수 있었을까?'

라는 의심을 해볼 수 있습니다. 쉽지 않겠죠? 익숙하지 못한 일을 비동기로 처리하는것은 어렵습니다. 이와 마찬가지로 프로그래밍을 시작할 때는 보통 동기식으로 코드를 배우고 작성하기 때문에 비동기식 코드에 익숙하지 못합니다. 그래서 어렵습니다.

우리는 통신의 처리를 비동기로 해야 하는 이유는 너무 잘 알고 있습니다.

그리고 통신의 처리는 XHR 객체를 이용하거나 fetch 함수를 사용해야 한다는것도 잘 알고 있습니다.

중요한 것은 통신 결과를 처리하는 방법이라고 할 수 있습니다.

우리는 동기식 코드에 익숙하기 때문에
보통
1) 값을 도출하는 어떤 함수를 실행하고
2) 그 결과값을 변수에 저장하고
3) 변수에 접근해서 그 결과값을 이용합니다.

하지만 비동기식 코드는
1) 값을 도출하는 어떤 비동기식 함수를 실행하고
2) 그 결과값을 변수에 저장하고
3) 변수에 접근해서 그 결과값을 이용합니다.

2) 과 3) 사이에 문제가 발생합니다. 3)의 코드가 1) -> 2)의 과정을 기다려 주지 않기 때문입니다.

그렇다면 비동기 동작을 동기식처럼 코드를 작성하는게 쉽겠네요! 그래서 나온것이 Promise 객체입니다.

Promise

비동기 동작의 처리를 담당하는 객체.
대기(pending), 이행(fulfilled), 거부(rejected) 세 가지 상태를 통해 비동기 함수의 결과를 처리합니다.
프로미스 자체가 비동기는 아닙니다. 그냥 객체일 뿐입니다.

비동기 함수하는 별난 녀석의 상태를 관찰하기 위해 태어났습니다!

사용법은

비동기 함수를 포-옥 감싸서 함수의 처리 결과로 단순한 데이터가 아닌 프로미스 객체를 반환하게 합니다.

그래서 일단 프로미스를 사용하면 좋은 점은 비동기함수의 결과로 undefined를 반환할 일이 없다는 것 입니다.

그리고 프로미스 객체가 가지고 있는 then 메소드를 통해 비동기 함수의 결과를 then과 then이 이어서 동기적으로 처리하도록 만들어 줍니다.

결론은 비동기를 동기처럼 코드상에서 보여지게 할 수 있다!

한 가지 주의할 점은 위에서 말한것 처럼 프로미스 자체는 비동기가 아니라는것!

코드로 같이 살펴보겠습니다.

Thread

스레드(thread)는 어떠한 프로그램 내에서 실행되는 흐름의 단위를 말합니다.

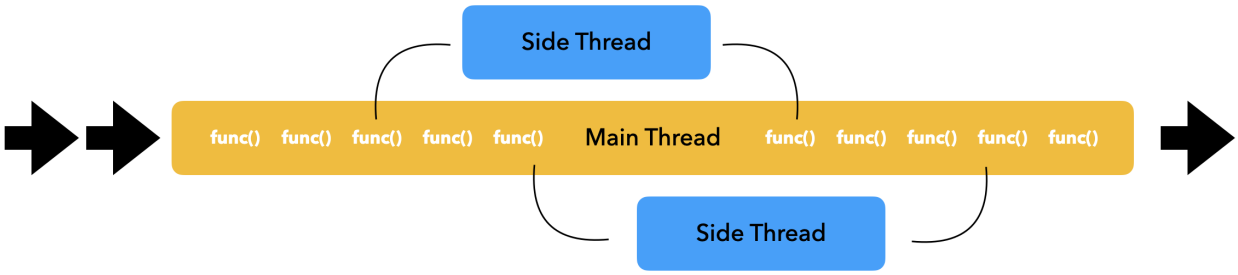
일종의 컨베이어벨트라고 생각하면 쉽습니다!

브라우저안에 있는 모든것은 메인쓰레드위에서 동작합니다.

자바스크립트 역시 브라우저가 해석하기 때문에 메인쓰레드 위에서 실행됩니다.

메엔스레드위에 실행이 오래 걸리는 코드가 작동한다면 그 뒤의 모든 코드가 처리가 지연되는 사태가 발생합니다.

때문에 브라우저는 사이드스레드를 생성해 실행이 오래걸릴것 같은 동작을 처리하고 실행이 종료되면 그결과를 다시 메인쓰레드로 가져와 브라우저에 반영하는 전략을 사용합니다.



그리고 이 모든 흐름을 관리하는 것은 이벤트루프 (event loop)!

만약 우리가 setTimeout 메소드를 실행한다고 가정했을때 이 함수가 메인쓰레드 위에서 실행되면 다른 모든 기능을 중단시키겠죠?

때문에 브라우저는 setTimeout 메소드를 메인이 아닌 사이드쓰레드에서 설정된 대기시간만큼 실행하도록 합니다.

JavaScript

```
function foo() {
  console.log('foo has been called');
}
setTimeout(foo, 0);

console.log('After setTimeout1');
console.log('After setTimeout2');
console.log('After setTimeout3');
console.log('After setTimeout4');
```

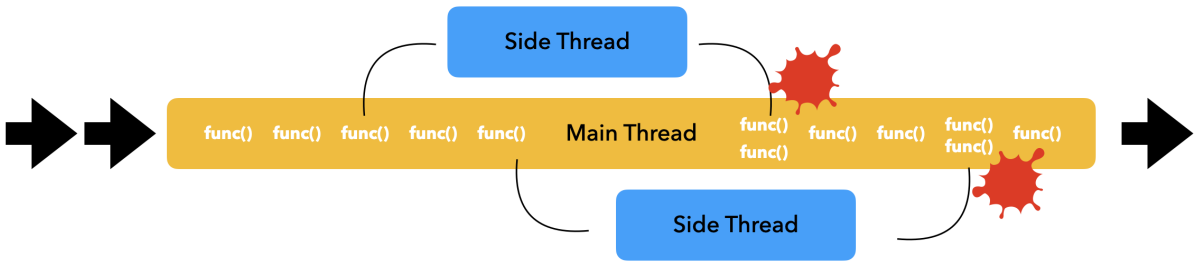
Console

```
"After setTimeout1"
"After setTimeout2"
"After setTimeout3"
"After setTimeout4"
"foo has been called"
```

그리고 약속된 시간이 되면 콜백함수를 메인쓰레드에서 실행합니다.

근데 만약 약속된 시간에 메인스레드에 와봤더니 거기에 이미 다른 메소드가 실행되고 있다면?

이때 발생하는 문제를 바로 ‘race condition problems’ 라고 합니다.



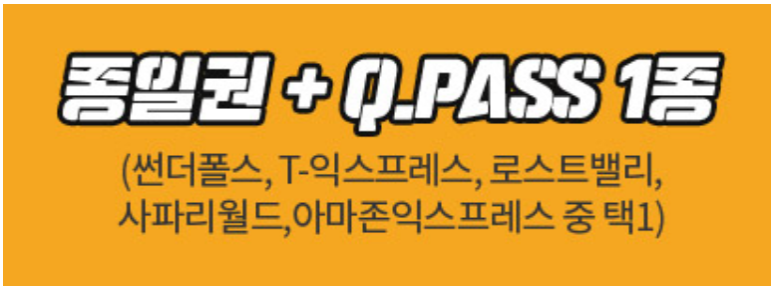
브라우저는 싱글쓰레드로 한번에 하나의 코드만 처리할 수 있기 때문에

이렇게 동시에 여러가지 코드가 실행되는 상황을 막아야한다. 누군가 교통정리를 해야합니다!

이때 등장하는것이 ‘task queues’. 이때 큐(queue)는 자료구조의 일종으로 First in First out 구조를 가집니다.

즉 어떤 처리를 순서대로 해야할때 사용하는 자료구조입니다.

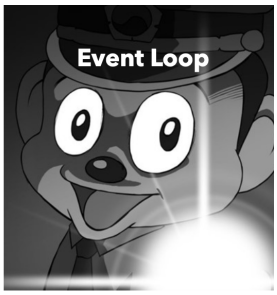
쉽게 말해 queue는 ‘대기줄’의 의미를 가지며 놀이공원의 대기줄을 생각하시면 되겠습니다.

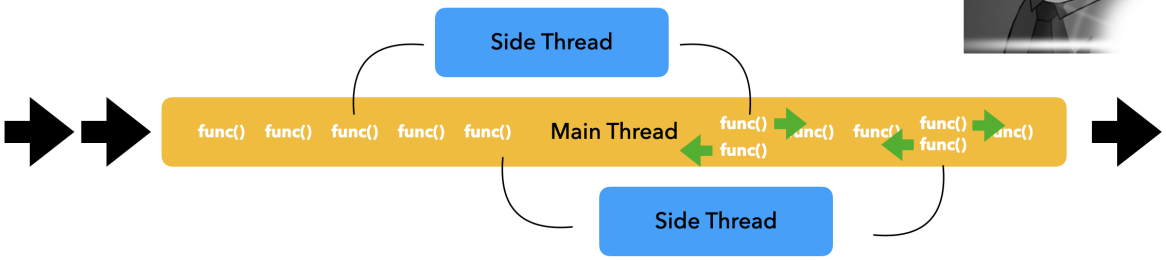


대기줄을 타지 않아도 되는 큐패스의 이름이 이런 의미가 있죠!

브라우저는 메인쓰레드에서 실행해아하는 어떤 함수가 나타나면 이벤트 루프에게 얘기해 이 함수를 테스크 큐에 등록합니다.

그리고 실행해야하는 함수의 차례가 오면 이벤트 루프는 함수를 실행합니다.





다시 정리하면 setTimeout 함수가 실행되면

대기시간동안에는 사이드쓰레드에 있다가

시간이 다 되면 이벤트 루프가 콜백 함수를 테스크 큐에 등록

테스크 큐에서 콜백함수의 차례가 되면 메인스레드에서 함수를 실행합니다!

테스크 큐는 크게 두 종류가 있습니다.

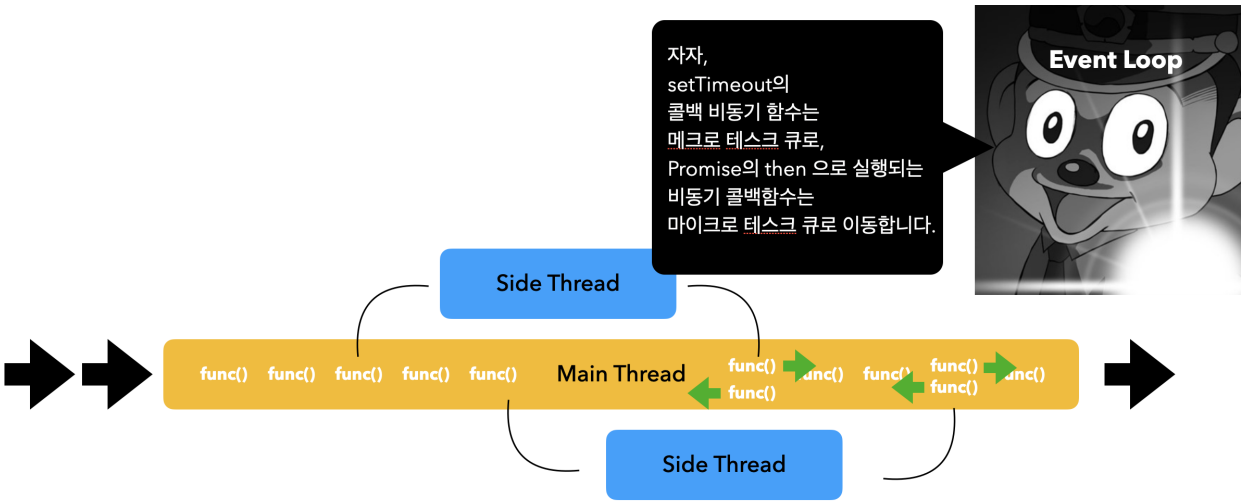
1. 메크로 테스크 (Macro Tasks)
2. 마이크로 테스크 (Mirco Tasks)

setTimeout 같은 메소드는 메크로 테스크큐에 등록되며 실행되기 전에 다른 코드가 실행되고 있다면 끝날 때까지 기다린 다음 실행됩니다. 질서 정연하게 차례를 지킨다고 볼 수 있겠죠?

하지만 마이크로 테스크 큐에 등록된 코드들은 차례가 되면 바로 메인스레드에서 실행됩니다.

이 말은 즉 마이크로 테스크큐에 등록된 코드들은 메크로 테스크 큐에 등록된 함수보다 더 높은 실행 우선순위를 가지며, 이 코드들이 실행될 때는 메크로 테스크 큐의 코드들의 실행을 배제한다고 볼 수 있습니다.

그리고 우리가 사용하는 프라미스 객체의 콜백 함수들은 모두 마이크로 테스크 큐에 등록됩니다.



async & await

async 키워드가 함수 앞에 붙으면 함수가 Promise를 반환하게 합니다.

그리고 await 는 async 함수가 Promise를 반환할 때, 상태가 결정될때까지 다음 코드의 실행을 멈추게 합니다.

그리고 상태가 결정되면 Promise 객체의 fulfilled 값을 반환합니다.

async, await 는 promise 의 설탕 문법입니다. 우리가 수업시간에 살펴봤던 코드를 보고 이 코드를 async, await 없이 구현해 보는건 어떨까요?

```
async function message() {
    let hello = await new Promise((resolve) => {
        setTimeout(() => {
            resolve('hello');
        }, 100)
    })

    let world = await new Promise((resolve) => {
        setTimeout(() => {
            resolve('world');
        }, 100)
    })

    console.log(`${hello} ${world}`);
}
```

```
message();
```

그리고 await의 중요한 특징은 async 함수 안에서 코드의 실행 순서를 확정지을 수 있다는것 입니다.

이때 주의 할 점은 자바스크립트 엔진은 await 를 만나는 순간 이를 비동기로 처리합니다.