Q1. 밑의 사진의 답과 뭔지 가 궁금합니다.

그리고, yield를 통해 비동기 코에서 함수끼리 통신이 가능하다는게 어떤 뜻인가요?

```
function* foo() {
    console.log(yield);
    console.log(yield);
    console.log(yield);
}
let g = foo();
g.next();
g.next(1);
g.next(2);
g.next(3);
```

Generator

Generator function 으로부터 반환된 객체야 반복자 반복자 프로토콜을 준수

문법: function* gen() {

yeild 1

yeild 2

yeild 3

}

var g = gen(); // "Generator { }

Yield : 다른 generator 또는 이터러블 객체에 yield를 위임할때 사용.

구문 : yield* [[expression]]

Generator.methods

1) Generator.prototype.next() : yield 표현을 통해 yield된 값을 리턴

2) Generator.prototype.return() : 주어진 값을 리턴하고 Generator를 종료.

3) Generator.prototype.throw() : Generator로 에러를 throw 합니다.

* Yield 는 Context Switching 을 한다.

yield는 return처럼 함수를 종료 But! 함수를 재호출(next()) 할경우 해당 지점에서 시작!

- Call back 이라기 보다는 함수를 실행하기위한 Stand by 상태

즉, 위의 var g = gen() 은 함수를 실행 한 것이 아니라 함수를 사용 가능한 준비상태 가 되는것

Generator는 Object로 출력된다.

Value : yield가 종료된 표현의 값

done : generator가 마지막 구문까지 실행 했냐 라는 의미. (마지막구문 → true)

* for of 구문 사용가능 → iterate 즉 반복가능. → Value만 출력 (Done은 X)

* 반복자가 반복자를 호출하는 방식 : Yield * 사용

* Yield로 반환 하는 게 없어도 next(임의값) 이 넣으면 출력이 가능하다.

⇒ 비동기 코에서 함수 꺼져 돌려줌 가능하다

Callback

1) 동기 : Hoisting 이후 코드들이 차례 대로 실행 되는것. (Synchronous)

2) 비동기 : 언제 코드가 실행 될지 알수없는 상태 (asychronous) 병렬 처리

* Call back function

: 전달된 함수를 나중에 호출을 통해 불러 오는것. ⇒ 비동기 실행방법 1.

- SetTimeout (Call back function() { }, ms)
  - 브라우저 API
  - ↳ Arrow function 가능 (() => ___ , ms)

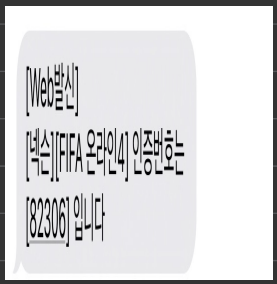
- 항수 Call back 에서도 SetTimeout 을 통해 Sychronous function 과 Asychronous function 으로 나눔.

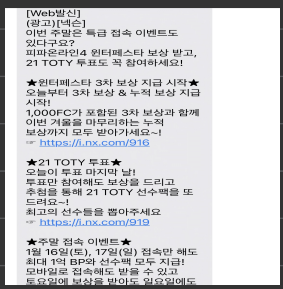
Promise

: 비동기를 간편하게 처리가능해도록해주는 Object. / task의 Critical Section을 Control / 병렬처리를 가능하게 해결.

: 비동기 작업이 맞이할 미래의 완료 또는 실패 와 그 결과 값을 나타낸다.

프로미스 생성 (사전등록) ⟶ 프로미스 등록 (사전예약하기) ⟶ response 값 전달 (안내문자)





활용방법

1) 상태를 확인해줌.

- 프로미스가 heavy한 operation을 수행하는지 (Network 사용? / file을 read 하는지?)
- 성공 vs 실패 했는지 상태확인.

Promise

-생성하기.

Promise는 Class → new Promise() 해야함.

executor 함수를 Call back → (Executor 함수는 (resolve :(Value?:any)⇒Void, reject:(reson?:any)⇒Void)⇒Void) ⇒ Promise<any>

-resolve : 기능을 정상적으로 수행해서 결과를 리턴

- reject : 기능을 수행하다가 뭔가 생기면 호출 하는 것

Const promise = new promise(resolve, reject) ⇒ { 수행할 작업 }

* new Promise() 하는 순간 바로 executor 함수 바로 실행

SetTimout 시간딜레이 후 resolve를 전달

```
const promise = new Promise(
    function(resolve, reject){
        setTimeout(function(){
            console.log("Hello")
            reject("Error!")
        }, 2000)
    }
)
promise.then(function(response){
    console.log("Success")
}). catch(function(error){
    console.log(error)
}). finally(() => {
    console.log("나는 무조건 실행된다.!!!")
})

const PromiseTest4 =()=>{
    console.log("PromiseTest Start")
    console.log("PromiseTest Fin")
```

-사용하기 (then.catch.finally) → Promise.prototype.

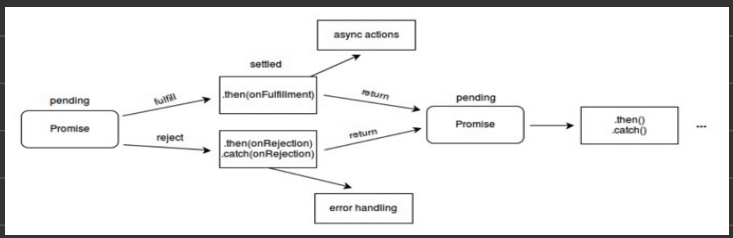
then: 값이 정상적으로 수행이 된다면 (then), 원하는 Call back 함수 수행.

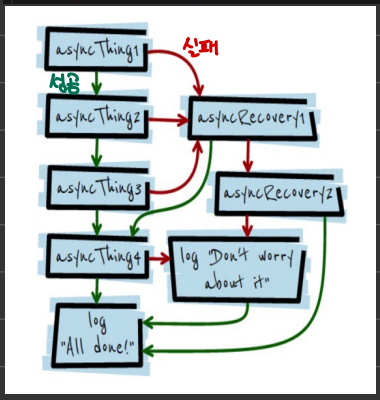
Promise.then(Callback 함수) → then 에 의해 대기열에 오름.

Catch: 에러 뱉을시 Callback 함수 수행.

.Catch(Callback 함수)

finally : 무조건 실행됨. (실패/성공 상관X)





Promise

- Promise parallel

① .then().then().then() ····

여러개의 Promise 들은 then 을 이어서 병렬처리. → 그럼 다음 catch 도 계속 이어서? X 한번의 사용으로 모든 then 처리 가능.

② Promise.all([Promise1( ), Promise2( ), ····])

주어진 모든 Promise 들은 실행한 후 전행되는 Promise 를 반환

③ Promise.race([Promise1( ), Promise2( ), ····])

주어진 Promise 들중 가장 먼저 완료된 것의 결과 값을 이행 or 거부

④ 변수선언 = await + expression
↳ 기다릴 어떤값

Promise 를 기다리기 위해 사용. → 비동기로 실행되는 것들은 끝날때까지 기다리는것.