Generator

여러개의 값을 필요에 따라 하나씩 반환(yield)할 수 있다.
generator를 만들려면 generator 함수인 **function***이 필요하다.
generator 함수는 일반 함수와 동작 방식이 다르다. 호출하면 코드가 실행되지 않고
대신 실행을 처리하는 generator객체가 반환된다.

.next()는 generator의 주요 메서드다.
.next()는 항상 두 프로퍼티를 가진 객체를 반환한다. value: 산출값 done : 함수 코드가 끝났으면 true 아니라면 false
이 메서드를 호출하면 가장 가까운 yield<value>문을 만날때까지 실행이 지속된다.
value가 없다면 생략할 수 있지만 이 경우 undefined가 된다.
yield<value>문을 만나면 실행을 멈추고 value값을 반환된다.

```
function* geTest() {
    yield "apple"
    yield "banana"
    yield "grape"
}
let fruit = geTest()
console.log(fruit.next())
console.log(fruit.next())
console.log(fruit.next())
console.log(fruit.next())
```

function* 로 generator 함수를 만든다.
fruit라는 함수는 geTest라는 generator객체를 생성한다.

fruit.next()로 geTest의 가장 가까운 yield문을 실행한다.
처음 만난건 yield<apple>이므로 실행을 잠깐 멈추고
value: apple done : 아직 돌수 있으므로 false 가 나온다.
다음으로 banana,grape까지 수행한다.
4번째 fruit.next()는 더이상 yield값이 없기 때문에
value: undefined done : 더이상 돌수없기때문에 true가 나온다.

Promise

자바스크립트 비동기 처리에 사용되는 객체다.
여기서 자바스크립트의 비동기 처리란 ,
특정 코드의 실행이 완료될 때까지 기다리지 않고 다음 코드를 먼저 수행하는 특성을 의미한다.

Promise의 3가지 상태(states)

여기서 말하는 상태란 promise 처리 과정을 의미한다. new Promise()로 생성하고 종료될 때까지 3가지 상태를 갖는다

[Pending(대기)]

-new Promise() 메서드를 호출하면 대기 상태가 된다.

이 메서드를 호출할 때 콜백 함수를 선언할 수 있고, 콜백 함수의 인자는 resolve,reject 이다.

[Fulfilled(이행)]

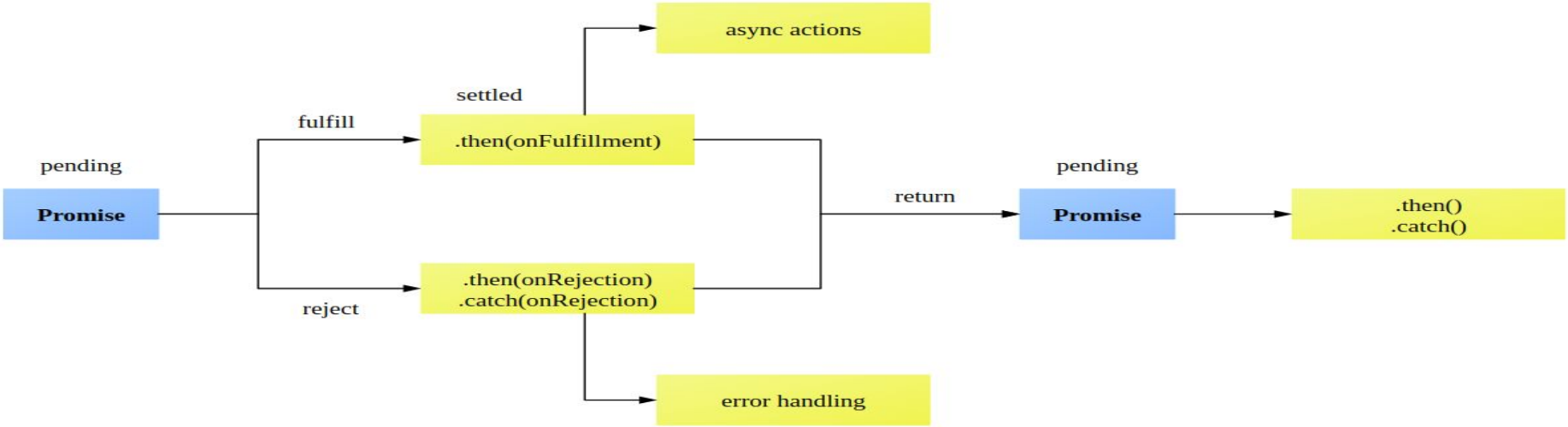
-resolve 인자를 Promise 메서드를 실행하면 이행 상태가 된다.

[Rejected(실패)]

-reject 인자를 호출하면 Rejected 상태가 된다. 또한 실패 상태가 되면 실패한 이유(실패 처리의 결과 값)를 catch()로 받을수 있다.

resolve,reject 결정

```
const promise = new Promise(
    executor: function (resolve, reject){
        setTimeout( handler: function (){
            console.log("Hello")
            resolve()
        }, timeout: 2000)
    }
)
```



Promise

promise.then.catch.final

.then promise가 이행되었을 때 실행되는 함수로 여기서 실행결과를 받는다.

.catch reject가 되었을때 여기로 reject값을 받는다.

.final solve,reject 여부와 상관없이 promise가 진행되면 실행한다.

```
// resolve()는 promise.then
// reject() 는 promise.catch
// finally() 는 무조건
promise.then(response =>
    console.log("Success")
).catch(function (error){
    console.log(error)
}).finally( onFinally: ()=>{
    console.log("PromiseTest5: 나는 무조건 실행된다!!!!")
})
```

Promise

promise.all 주어진 이터러블 객체의 프라미스가 모두 이뤄질 때 promise를 반환하는 메서드다.
어떤 비동기 작업들을 다 끝내고 나서 다음 작업으로 넘어갈 때 유용하다.

```
Promise.all( values: [
    plus( num1: 100, num2: 200),
    minus( num1: 100, num2: 200),
    mult( num1: 100, num2: 200),
    divide( num1: 100, num2: 200)
]).then(response => console.log(response))
```

promise.all을 통해
[plus,minus,mult,divide]를 동시에 작업한다.
모든 promise가 다 끝나면 출력한다.

Promise

promise.race promise.all과 비슷하지만 가장 먼저 처리되는 promise의 결과(혹은 에러)를 반환한다.

```
Promise.race( values: [
    plus( num1: 100, num2: 200),
    minus( num1: 100, num2: 200),
    mult( num1: 100, num2: 200),
    divide( num1: 100, num2: 200)
]).then(response => console.log(response))
```

promise.race을 통해
[plus,minus,mult,divide]를 동시에 작업한다.
promise의 setTimeOut에 의해 가장 빠르게 값이 나온
divide가 출력된다.

Async & Await

async와 await는 자바스크립트의 비동기 처리패턴 중 가장 최근에 나온 문법이다.
콜백 함수와 프로미스의 단점을 보완하고 가독성을 높여준다.

async는 function앞에 붙으며 해당 함수는 항상 promise를 반환한다.
promise가 아닌 값을 반환하더라도 이행 상태의 promise(resolved promise)로 값을 감싸 이행된 promise가 반환되도록 한다.

await는 async 함수 안에서만 동작한다.
이 키워드를 만나면 해당 promise가 처리될때까지 기다린다 결과는 그 이후에 반환한다.

```
async function asyncProcess() {
    const res = await Promise.all(
        values: [
            plus( num1: 100,  num2: 200),
            minus( num1: 100,  num2: 200),
            mult( num1: 100,  num2: 200),
            divide( num1: 100,  num2: 200)
        ]
    )
    console.log(res)

}
```

asyncProcess의 promise는 항상 반환되며
'res'라는 Promise.all을 통해 [plus,minus,mult,divide] promise 전체를 수행하며
동시에 await 키워드로 4개의 promise가 전부 수행된뒤 console.log(res)가
수행된다.