


# Promise & Async, Await

## [ Promise ]

### 1. 용도

- 실행은 바로하고, 결과값 (resolve, reject)은 나중에 Return 받는 비동기 방식 수행 객체
- 결과값은 실행 완료 후, then/catch로 받는다
- 기본적으로 싱글스레드로 실행되는 JavaScript언어에서, 비동기 방식으로 수행되는 코드 작성을 위해 사용함.  
(setTimeout 함수를 이용한 구현시 발생하는 Callback-Hell 문제를 해결하기 위해 나온 신규 문법)
- [\(MDN\)Promise설명](#)
- 3개의 상태가 존재
  - 대기(pending): 이행하지도, 거부하지도 않은 초기 상태.
  - 이행(fulfilled): 연산이 성공적으로 완료됨.
  - 거부(rejected): 연산이 실패함. 

### 2. 문법구조

- resolve, reject 매개변수를 갖는 콜백함수를 구현하며 생성
- 객체 생성 시점에 내부 프로세스는 수행되며, 결과값 Return만 .then / .catch 메소드 호출 시점에 수행됨
- .catch 는 .then( undefined, failureCallback) 와 동일

```
const promise1 = new Promise((resolve, reject) => {
  if (false) {
    console.log('Promise True Logic');
    resolve('Pass');
  } else {
    console.log('Promise Fail Logic');
    reject('Fail');
  }
});

console.log('sync Process 1');
console.log('sync Process 2');

promise1.then((msg) => {
  console.log(msg);
}, (err) => {
  console.log(err);
});
```

▶ ===== [Result] =====

```
Promise Fail Logic
sync Process 1
```

```
sync Process 2
Fail
```

- `.then()` 여러번 사용하여 여러개의 콜백을 추가 할 수 있으며, 각각의 콜백은 주어진 순서대로 하나 하나 실행  
※ 성능상 좋지 않으며, 가능한 하나의 핸들러 내에서 수행토록 작성

```
myPromise
  .then((value) => `${value} and bar 1 `)
  .then((value) => `${value} and bar again 2`)
  .then((value) => `${value} and again 3`)
  .then((value) => `${value} and again 4`)
  .then((value) => {
    console.log(value);
  })
  .catch((err) => {
    console.error(err);
  });
```

▶ ===== [Result] =====

```
foo and bar 1  and bar again 2 and again 3 and again 4
```

### 3. 활용방법

- [\(MDN\) Using promises](#)
- [자바스크립트 프로미스 이해하기: 작동원리 완전분석](#)
- **Chaining**
  - Promise의 가장 뛰어난 장점 중 하나!
  - **Chaining after a catch**  
chain에서 작업이 실패한 후에도 새로운 작업을 수행하는 것이 가능하며 매우 유용

```
new Promise((resolve, reject) => {
  console.log("Initial");

  resolve();
})
  .then(() => {
    throw new Error("Something failed");
    console.log("Do this");
  })
  .catch(() => {
    console.log("Do that");
  })
```

```
.then(() => {  
  console.log("Do this, whatever happened before");  
});
```

► ===== [Result] =====

```
Initial  
Do that  
Do this, whatever happened before
```

참고: "Do this" 텍스트가 출력되지 않은 것에 주의!  
"Something failed" 에러가 rejection을 발생시켰기 때문임.

## [ Async, Await ]

### 1. 용도

- Promise를 활용하여 비동기 방식 구현 시 Callback-Hell 구조를 개선 했지만, 여전히 코드가 길어지는 문제가 존재.  
이를 해결하기 위해 새롭게 등장한 문법
- 함수 선언 시 async 사용
- async 함수의 반환값은 항상 Promise로 감싸져서 반환됨.

### 2. 문법구조

- [\(MDN\) async function](#)

### 3. 활용방법

```
const promise1 = Promise.resolve('Success1');  
const promise2 = Promise.resolve('Success2');  
  
(async () => {  
  for await (promise of [promise1, promise2]) {  
    console.log(promise);  
  }  
})();  
  
console.log('sync1');  
console.log('sync2');  
console.log('sync3');
```

```

console.log('sync4');
console.log('sync5');

(async () => {
  for (let i = 0; i < 10; i++) {
    console.log(`for loop sync : ${i}`);
  }
})();

console.log('sync6');
console.log('sync7');
console.log('sync8');
console.log('sync9');
console.log('sync10');

const fn = () => {
  for (let i = 0; i < 10; i++) {
    console.log(`for loop sync@fnPromise : ${i}`);
  }
};

const fnPromise = async () => {
  const p = new Promise((resolve, reject) => {
    resolve(fn);
  })
  return p;
};

fnPromise().then(f => { f(); });

console.log('sync11');
console.log('sync12');
console.log('sync13');
console.log('sync14');
console.log('sync15');

```

► ===== [Result] =====

```

sync1
sync2
sync3
sync4
sync5
for loop sync : 0
for loop sync : 1
for loop sync : 2
for loop sync : 3
for loop sync : 4
for loop sync : 5
for loop sync : 6
for loop sync : 7
for loop sync : 8

```

```
for loop sync : 9
sync6
sync7
sync8
sync9
sync10
sync11
sync12
sync13
sync14
sync15
Success1
for loop sync@fnPromise : 0
for loop sync@fnPromise : 1
for loop sync@fnPromise : 2
for loop sync@fnPromise : 3
for loop sync@fnPromise : 4
for loop sync@fnPromise : 5
for loop sync@fnPromise : 6
for loop sync@fnPromise : 7
for loop sync@fnPromise : 8
for loop sync@fnPromise : 9
Success2
```