

# 45장 프로미스



모던 자바스크립트 Deep Dive

# 1. 비동기 처리를 위한 콜백 패턴의 단점 - 콜백 헬

- 비동기 함수는 비동기 처리 결과를 외부에 반환할 수 없고, 상위 스코프의 변수에 할당할 수도 없음
  - > 비동기 함수의 처리 결과에 대한 후속 처리를 비동기 함수 내부에서 수행해야 함
- 콜백 헬이란?
- 콜백 함수를 통해 비동기 처리 결과에 대한 후속 처리를 수행하는 비동기 함수가 비동기 처리 결과를 가지고 또다시 비동기 함수를 호출할 때 콜백 함수 호출이 중첩되어 복잡도가 높아지는 현상

# 1. 비동기 처리를 위한 콜백 패턴의 단점 - 콜백 헬

```
get('/step1', (a) => {  
  get(`/step2/${a}`, (b) => {  
    get(`/step3/${b}`, (c) => {  
      get(`/step4/${c}`, (d) => {  
        console.Log(d);  
      });  
    });  
  });  
});  
});
```

# 1. 비동기 처리를 위한 콜백 패턴의 단점 - 에러 처리의 한계

```
try {  
    setTimeout(() => { throw new Error('Error!'); }, 1000);  
} catch (e) {  
    console.error('캐치한 에러', e);  
}
```

→ 에러를 캐치하지 못함

- setTimeout 함수의 콜백 함수를 호출한 것은 setTimeout 함수가 아님
  - 에러는 호출자 방향으로 전파됨
- > setTimeout 함수의 콜백 함수가 발생시킨 에러는 catch 블록에서 캐치되지 않음

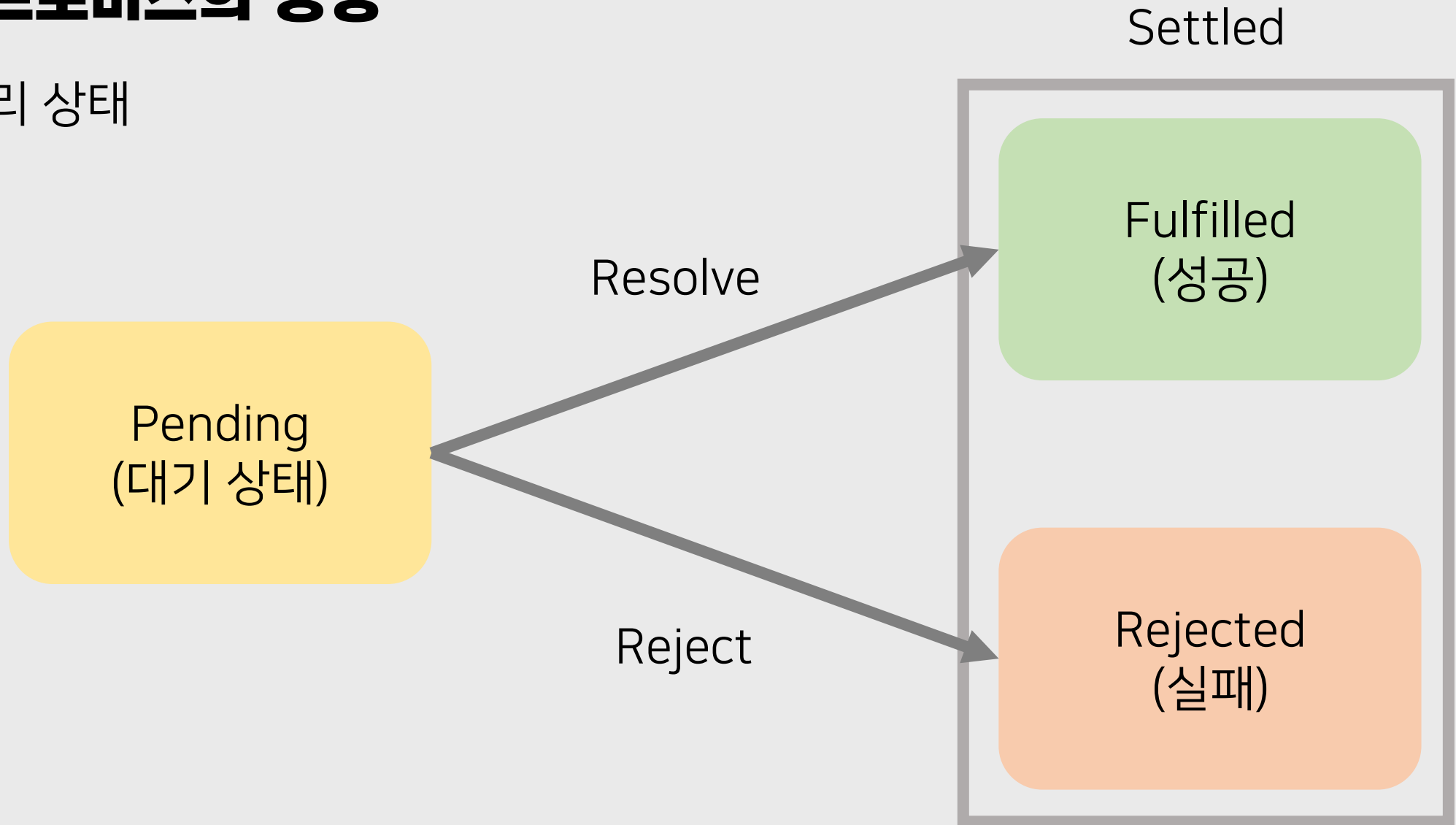
## 2. 프로미스의 생성

```
const promise = new Promise((resolve, reject) => {  
  if (/* 비동기 처리 성공 */) {  
    resolve('result');  
  } else { /* 비동기 처리 실패 */  
    reject('failure reason');  
  }  
});
```

- 비동기 처리가 성공하면 콜백 함수의 인수로 전달받은 **resolve** 함수를 호출
- 비동기 처리가 실패하면 **reject** 함수를 호출

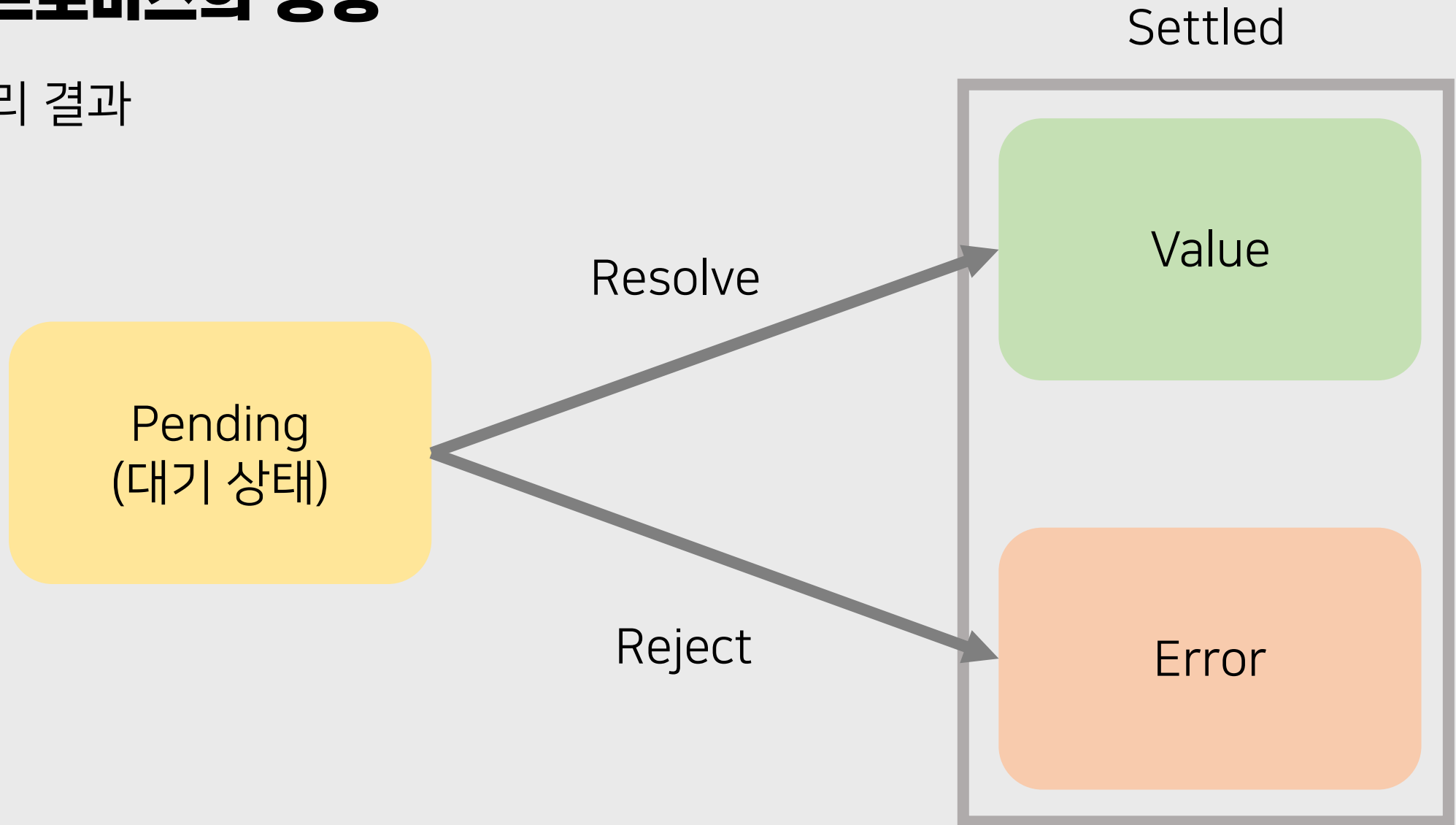
## 2. 프로미스의 생성

- 처리 상태



## 2. 프로미스의 생성

- 처리 결과



### 3. 프로미스의 후속 처리 메서드

`Promise.prototype.then`


`Promise.prototype.catch`


`Promise.prototype.finally`



### 3. 프로미스의 후속 처리 메서드 - then

```
new Promise((resolve) => resolve('fulfilled')).then(  
  (v) => console.log(v),  
  (e) => console.error(e)  
); // fulfilled
```

 → 비동기 처리 성공했을 때 호출

 → 비동기 처리 실패했을 때 호출

```
new Promise( (_, reject) => reject(new Error('rejected'))).then(  
  (v) => console.log(v),  
  (e) => console.error(e)  
); // Error: rejected
```

- 첫 번째 콜백 함수는 성공 처리 콜백 함수, 두 번째 콜백 함수는 실패 처리 콜백 함수

### 3. 프로미스의 후속 처리 메서드 - catch, finally

```
new Promise((_, reject) => reject(new Error('rejected')))  
  .catch((e) => console.log(e)); // Error: rejected
```

- catch 메서드의 콜백 함수는 프로미스가 rejected 상태인 경우만 호출됨

```
new Promise(() => {})  
  .finally(() => console.log('finally')); // finally
```

- finally 메서드의 콜백 함수는 프로미스의 성공 또는 실패와 상관없이 한 번 호출됨

## 4. 프로미스의 에러 처리

```
promiseGet(wrongUrl).then(  
  (res) => console.log(res),  
  (err) => console.error(err)  
);
```

- then 메서드의 두 번째 콜백 함수로 처리

```
promiseGet(wrongUrl)  
  .then((res) => console.log(res))  
  .catch((err) => console.error(err));
```

- catch를 사용해 처리

## 5. 프로미스 체이닝

```
const url = 'https://jsonplaceholder.typicode.com';

promiseGet(`${url}/posts/1`)
  .then(({ userId }) => promiseGet(`${url}/users/${userId}`))
  // promiseGet 함수가 반환한 프로미스가 resolve한 값
  .then((userInfo) => console.log(userInfo))
  // 첫 번째 then 메서드가 반환한 프로미스가 resolve한 값
  .catch((err) => console.error(err))
  // promiseGet 함수 또는 앞선 후속 처리 메서드가 반환한 프로미스가
  // reject한 값
);
```

## 6. 프로미스의 정적 메서드

`Promise.resolve`

`Promise.reject`

`Promise.all`

`Promise.race`

`Promise.allSettled`

## 6. 프로미스의 정적 메서드 - resolve, reject

- resolve와 reject 메서드는 이미 존재하는 값을 래핑하여 프로미스를 생성하기 위해 사용

```
const resolvedPromise = Promise.resolve([1, 2, 3]);  
resolvedPromise.then(console.log); // [1, 2, 3]
```

- 아래 코드처럼 동작함

```
const resolvedPromise = new Promise((resolve) =>  
  resolve([1, 2, 3]));  
resolvedPromise.then(console.log); // [1, 2, 3]
```

## 6. 프로미스의 정적 메서드 - resolve, reject

- resolve와 reject 메서드는 이미 존재하는 값을 래핑하여 프로미스를 생성하기 위해 사용

```
const rejectedPromise = Promise.reject(new Error('Error!'));  
rejectedPromise.catch(console.log);
```

- 아래 코드처럼 동작함

```
const rejectedPromise = new Promise( (_, reject) =>  
    reject(new Error('Error!')) );  
rejectedPromise.catch(console.log);
```

## 6. 프로미스의 정적 메서드 - all

- all 메서드는 여러 개의 비동기 처리를 모두 병렬 처리할 때 사용

```
const requestData1 = () =>
  new Promise((resolve) => setTimeout(() => resolve(1), 3000));
const requestData2 = () =>
  new Promise((resolve) => setTimeout(() => resolve(2), 2000));
const requestData3 = () =>
  new Promise((resolve) => setTimeout(() => resolve(3), 1000));
```

```
Promise.all([requestData1(), requestData2(), requestData3()])
  .then(console.log)
  .catch(console.error);
```



## 6. 프로미스의 정적 메서드 - race

- race 메서드는 all 메서드처럼 모든 프로미스가 fulfilled 상태가 되는 것을 기다리지 않고, 가장 먼저 fulfilled 상태가 된 프로미스의 처리 결과를 resolve하는 새로운 프로미스 반환

```
Promise.race([
  new Promise((resolve) => setTimeout(() => resolve(1), 3000)), // 1
  new Promise((resolve) => setTimeout(() => resolve(2), 2000)), // 2
  new Promise((resolve) => setTimeout(() => resolve(3), 1000)), // 3
])
  .then(console.log) // 3
  .catch(console.log);
```

## 6. 프로미스의 정적 메서드 - allSettled

- allSettled 메서드는 전달받은 프로미스가 모두 settled 상태가 되면 처리 결과를 배열로 반환함

```
Promise.allSettled([
  new Promise((resolve) => setTimeout(() => resolve(1), 2000)),
  new Promise((_, reject) =>
    setTimeout(() => reject(new Error('Error!')), 1000))
]).then(console.log);
```

```
{status: "fulfilled", value: 1},
{status: "rejected", reason: Error: Error! At <anonymous>:3:54}
```

## 7. 마이크로태스크 큐

- 프로미스의 후속 처리 메서드의 콜백 함수는 마이크로태스크 큐에 저장됨
- 마이크로태스크 큐는 태스크 큐보다 우선순위가 높음

```
setTimeout(() => console.log(1), 0);
```

```
Promise.resolve()
```

```
.then(() => console.log(2))
```

```
.then(() => console.log(3));
```

## 8. fetch

- fetch 함수는 HTTP 요청 전송 기능을 제공하는 클라이언트 사이드 Web API
- HTTP 응답을 나타내는 Response 객체를 래핑한 Promise 객체를 반환

```
const promise = fetch(url [, options])
```

```
post(url, payload) {  
  return fetch(url, {  
    method: 'POST',  
    headers: {'content-Type': 'application/json'},  
    body: JSON.stringify(payload)  
  });  
}
```