

# 13. 비동기 프로그래밍

---

# 비동기 처리 방식

---

# 동기 처리 방식과 비동기 처리 방식

예) 커피 전문점에서 커피 주문하고 마시기

## <동기 처리 방식>

- A라는 사람이 커피를 주문하면 그 주문을 받아서 커피를 만들고 A에게 넘겨준다.
- 뒤에 아무리 많은 손님이 있어도 한번에 하나의 손님만 처리한다.
- 주문을 받고 커피를 만드는 것이 한 과정이기 때문에 대기 줄이 점점 더 길어지고 주문을 처리하는데도 시간이 걸린다.

## <비동기 처리 방식>

- A라는 사람이 커피를 주문하면 그 주문을 주방으로 넘기고,
- A에게는 진동벨을 주면서 커피가 완성되면 알려 주겠다고 한다.
- 대기하고 있던 B의 주문을 받고 진동벨을 건네준다.
- 중간에 A의 커피가 완성되면 A에게 알려 준다.

# 싱글 스레드 vs 멀티 스레드

**스레드(thread)** : 프로세스에서 작업을 실행하는 단위

한번에 하나의 스레드만 처리하면 '싱글 스레드', 한번에 여러 스레드를 사용한다면 '멀티 스레드'라고 한다

자바스크립트는 싱글 스레드를 사용한다. → 한번에 하나의 작업만 처리할 수 있다.

시간이 오래 걸리는 작업이 앞에서 실행 중이라면 그 작업이 끝날 때까지 무작정 기다려야 한다.

시간이 짧은 함수를 먼저 처리해서 멀티 스레드처럼 동작하게 한다.

```
function displayA() {  
  console.log("A");  
}  
function displayB() {  
  setTimeout(() => console.log("B"), 2000);  
}  
function displayC() {  
  console.log("C");  
}  
displayA();  
displayB();  
displayC();
```

A, B, C 순서가 아니라 A, C, B 순서로 진행  
display()를 2초 후에 실행하도록 했기 때문

A	<a href="#">async.js:2</a>
C	<a href="#">async.js:8</a>
B	<a href="#">async.js:5</a>
>	

# 함수 이름을 콜백으로 사용하기

예) 사용자가 커피를 주문하면 3초 후에 커피가 완료되는 프로그램

1) 커피 주문 프로그램

```
function order(coffee) {  
    console.log(`${coffee} 주문 접수`);  
}
```



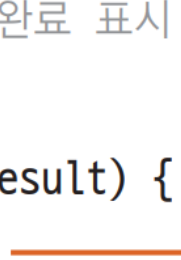
2) 완성됐다고 알려주는 프로그램

```
function display(result) {  
    console.log(`${result} 준비 완료`);  
}
```

# 함수 이름을 콜백으로 사용하기

예상하는 프로그램 흐름

```
function order(coffee) {  
  // coffee 주문  
  // 3초 기다린 후 완료 표시  
}  
function display(result) {  
  // 커피 완료 표시  
}
```

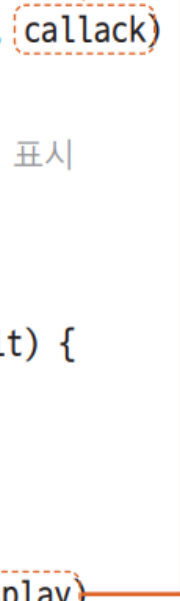


A solid orange line with an arrowhead starts from the right side of the comment '// 3초 기다린 후 완료 표시' in the `order` function and points to the `display` function.

콜백을 사용한 프로그램 흐름

```
function order(coffee, callback) {  
  // coffee 주문  
  // 3초 기다린 후 완료 표시  
}  
  
function display(result) {  
  // 커피 완료 표시  
}
```

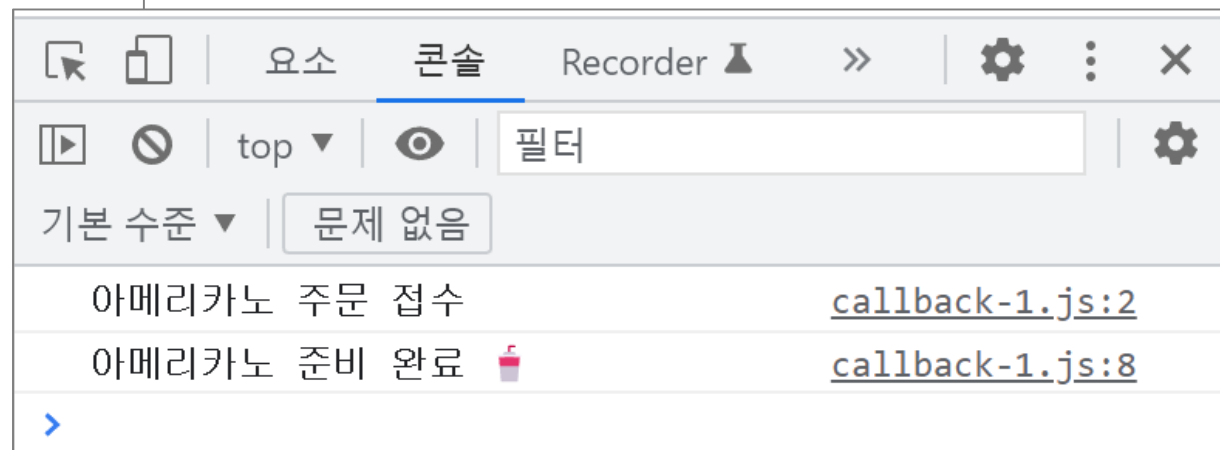
`order("아메리카노", display)`



A solid orange line with an arrowhead starts from the `callback` parameter in the `order` function signature, goes up and then right, and points to the `display` function. The `callback` parameter and the `display` function are enclosed in dashed orange boxes.

13Wcallback-1.html, 13WjsWcallback-1.js

```
function order(coffee, callback) {  
  console.log(`${coffee} 주문 접수`);  
  setTimeout(() => {  
    callback(coffee);  
  }, 3000);  
}  
  
function display(result) {  
  console.log(`${result} 준비 완료`);  
}  
  
order("아메리카노", display);
```





# 익명 콜백 사용

예) 1초마다 A -> B -> C -> D -> STOP! 순으로 표시하기

13WjsWcallback-2.js

```
function displayLetter() {  
  console.log("A");  
  setTimeout( () => {  
    console.log("B");  
    setTimeout( () => {  
      console.log("C");  
      setTimeout( () => {  
        console.log("D");  
        setTimeout( () => {  
          console.log("stop!");  
        }, 1000);  
      }, 1000);  
    }, 1000);  
  }, 1000);  
}  
displayLetter();
```

콜백이 계속 반복되는 상태를 '콜백 지옥' 이라고 한다.  
소스의 가독성이 떨어지고, 오류가 발생했을 때 디버깅  
하기 어렵다.

---

**프로미스**

---

# 프로미스

- 콜백 안에 콜백, 그 안에 또 콜백 .... ➔ 콜백 지옥
- 콜백을 사용했을 때의 복잡함을 피하기 위해, ES6부터 "프로미스(promise)" 등장.

처리에 성공했을 때 실행할 함수와 성공하지 못했을 때 실행할 함수를 미리 약속하자~  
프로미스를 사용하려면 Promise 객체를 먼저 만들어야 한다.

성공했을 때 실행할 함수 resolve()와 실패했을 때 실행할 함수 reject()도 함께 지정

```
new Promise(resolve, reject)
```

프로미스는 객체를 생성(제작)하는 부분과 소비하는 부분으로 나뉜다.  
프로미스 제작 코드에서 '성공' 과 '실패'를 확인한 후 소비 코드로 알려준다.

## 예) 피자 주문 흐름을 만들어 보자

likePizza가 true라면 → 성공했을 때 실행할 함수에 '피자를 주문합니다.' 넘긴다

likePizza가 false라면 → 실패했을 때 실행할 함수에 '피자를 주문하지 않습니다.' 넘긴다

13wjsWpromise-1.js

```
let likePizza = true;
const pizza = new Promise((resolve, reject) => {
  if (likePizza)
    resolve('피자를 주문합니다.');
```

```
    else
      reject('피자를 주문하지 않습니다.');
```

```
});
```

프로미스 제작 코드

프로미스를 실행할 때 사용하는 함수

- then() – 프로미스에서 성공했다는 결과를 보냈을 때 실행할 소스
- catch() – 프로미스에서 실패했다는 결과를 보냈을 때 실행할 소스
- finally() – 프로미스의 성공과 실패에 상관없이 실행할 소스

```
프로미스객체.then( ).catch().finally();
```

*프로미스객체*



```
.then( )
```

```
.catch()
```

```
.finally();
```

## 프로미스 제작 코드

```
let likePizza = true;
const pizza = new Promise((resolve, reject) => {
  if (likePizza)
    resolve('피자를 주문합니다.');
```



```
  else
    reject('피자를 주문하지 않습니다.');
```

```
});
```

## 프로미스 소비 코드

```
pizza
  .then(
    result => console.log(result)
  )
  .catch(
    err => console.log(err)
  );
```

likePizza 가 true일 때

피자를 주문합니다. [promise-1.js:12](#)



likePizza 가 false일 때

피자를 주문하지 않습니다. [promise-1.js:7](#)



```
pizza
  .then (
    result => console.log(result)
  )
  .catch (
    err => console.log(err)
  )
  .finally (
    () => console.log("완료")
  );
```

피자를 주문합니다.

[promise-1.js:9](#)

완료

[promise-1.js:15](#)



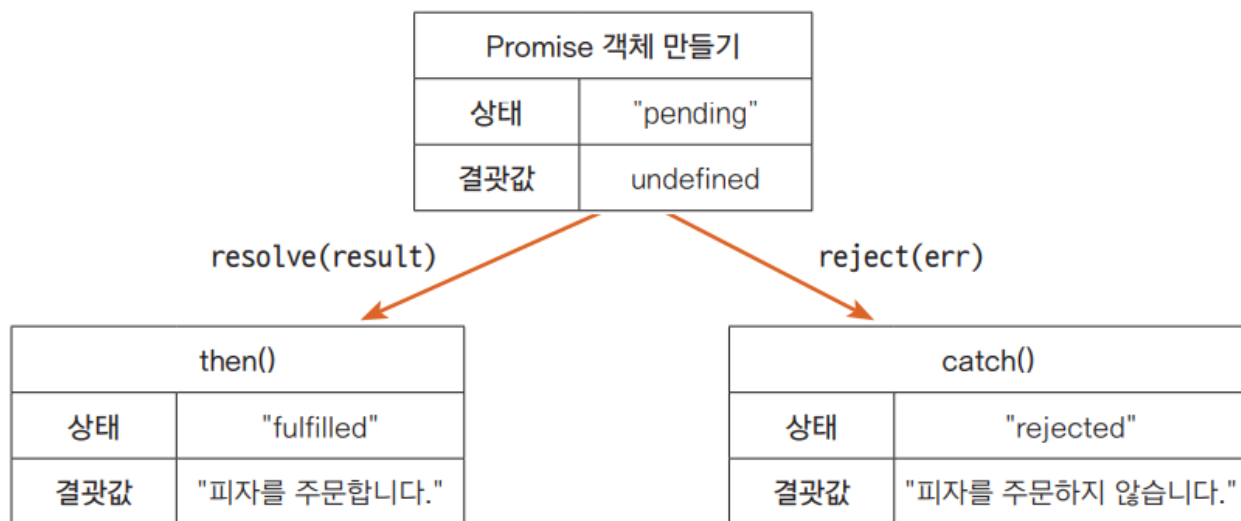
# 프로미스의 상태

프로미스는 `resolve()` 함수나 `reject()` 함수를 매개변수로 받아서 실행하는 객체  
프로미스 객체는 `resolve()` 함수나 `reject()` 함수를 실행하면서 상태가 바뀜

상태	설명
pending	처음 프로미스를 만들면 대기 상태 <sup>pending</sup> 가 됩니다.
fulfilled	처리에 성공하면 이행 상태 <sup>fulfilled</sup> 가 됩니다.
rejected	처리에 성공하지 못하면 거부 상태 <sup>rejected</sup> 가 됩니다.



# 프로미스의 상태



- 제작코드 : fulfilled 상태인지, reject 상태인지에 따라 '피자를 주문합니다.' 또는 '피자를 주문하지 않습니다.'라는 결괏값을 넘겨준다.
- 소비코드 : 결괏값을 result나 err 같은 변수 이름으로 받아서 사용한다.

# [실습] 커피 주문하고 완료하는 프로미스

13WjsWpromise-ex.js

## 프로미스 제작 코드

프롬프트 창을 통해 사용자에게 원하는 커피를 입력하게 한다.

커피 메뉴가 입력되었으면 화면에 주문이 접수되었다고 표시하고

3초 후에 resolve 함수에 커피 메뉴를 건네준다.

커피 메뉴가 입력되지 않았으면 주문이 없다는 오류 메시지를 표시한다.

## 프로미스 소비 코드

성공했으면 display 함수를, 실패하면 showErr 함수를 실행한다.

화면에 내용을 표시하는 display 함수와 오류를 표시하는 showErr 함수를 만든다.

1. order 라는 프로미스 객체를 만든다. (우선 프로미스만 만들기)

```
const order = new Promise((resolve, reject) => {  
  });
```

2. 라이브 서버를 사용해 콘솔 창에서 order 객체의 상태를 확인한다. (웹 브라우저 창은 계속 열어두기)

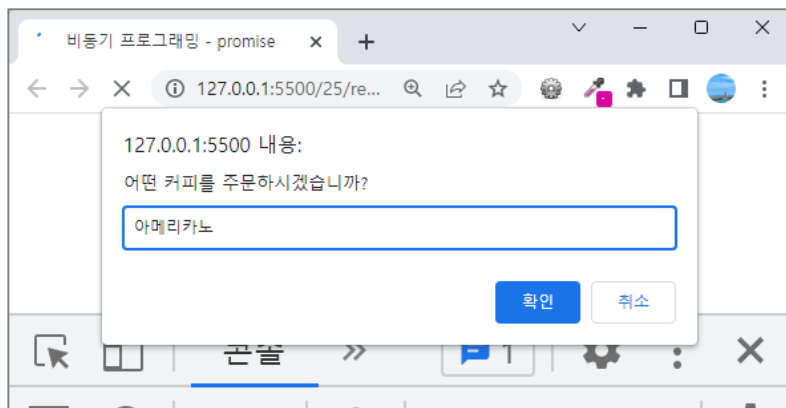
```
> order  
< ▶ Promise {<pending>}  
> |
```

프로미스의 상태는 pending

3. order 객체 완성하기

```
const order = new Promise((resolve, reject) => {  
  let coffee = prompt("어떤 커피를 주문하시겠습니까?", "아메리카노");  
  if(coffee != null && coffee != "") {  
    document.querySelector(".start").innerText = `${coffee} 주문 접수`;  
    setTimeout(() => {  
      resolve(coffee);  
    }, 3000);  
  } else {  
    reject("커피를 주문하지 않았습니다.");  
  }  
});
```

4. 웹 브라우저 창으로 돌아와 원하는 메뉴를 입력하고 [Enter]를 누른다



5. 웹 브라우저 창에는 '아메리카노 접수'가 나타난다.

6. 콘솔 창에서 order 객체의 상태 확인하기



현재 상태는 fulfilled이고

결괏값은 프롬프트 창에서 받은 입력 내용

## 7. then()을 실행했을 때 어떻게 실행되는지 확인하기

`order.then()`

```
> order
< ▶ Promise {<fulfilled>: '아메리카노'}
> order.then()
< ▶ Promise {<fulfilled>: '아메리카노'}
>
```

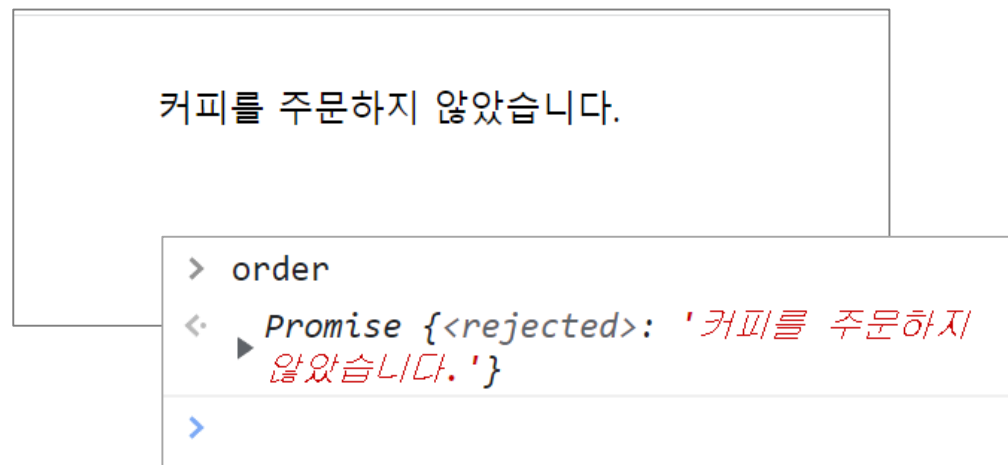
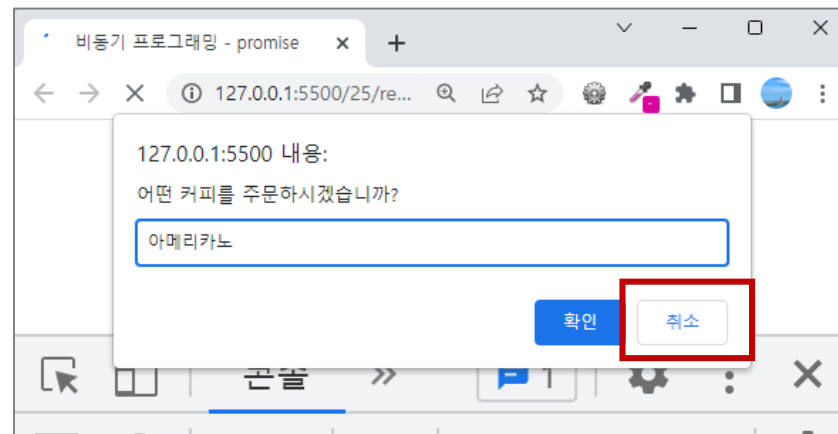
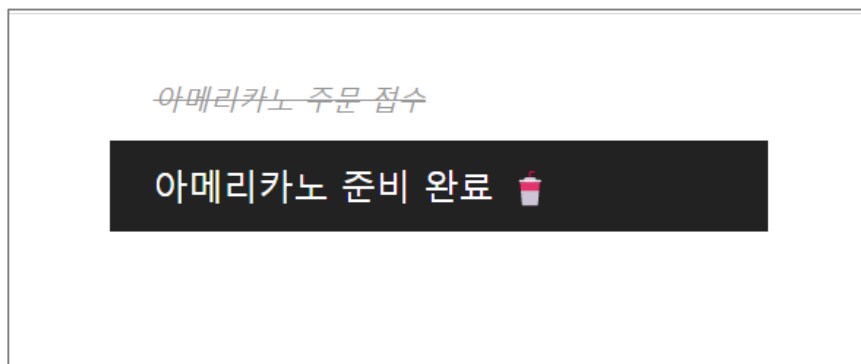
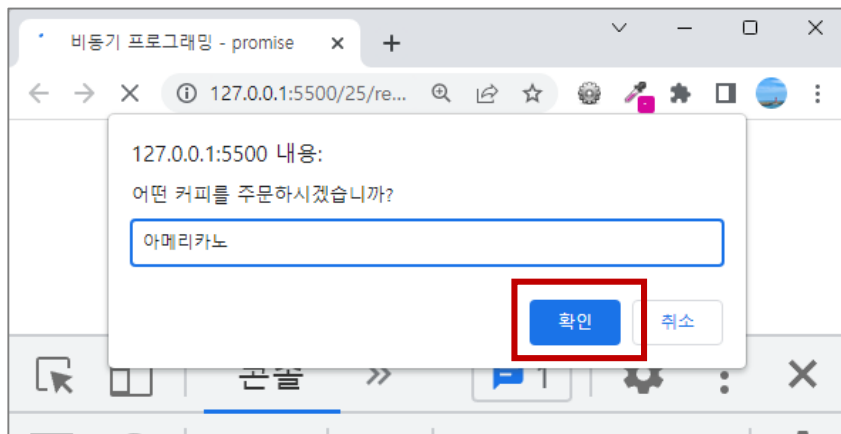
기억하자!!

then()을 실행하면 프로미스를 반환한다

## 8. 프로미스 소비 코드를 작성하기

```
function display(result) {  
    document.querySelector(".end").innerText = `${result} 준비 완료 🍽️ `;  
    document.querySelector(".end").classList.add("active");  
    document.querySelector(".start").classList.add("done");  
}  
  
function showErr(err) {  
    document.querySelector(".start").innerText = err;  
}  
  
order  
    .then(display)  
    .catch(showErr);
```

## 8. 웹 브라우저로 돌아가, 다시 커피 메뉴 주문하기



# 여러 단계 연결해서 프로그램 만들기

프로그램은 여러 단계를 연결해서 사용할 경우가 많다.

(예: 서버에서 학생 자료를 가져온다 → 성공하면? 가져온 자료를 객체로 만든다 → 성공하면? 객체에서 필요한 정보만 꺼낸다 → 성공하면? 화면에 표시한다 .... )

여러 단계를 연결할 때 콜백 함수를 사용할 수도 있고, 프로미스를 사용할 수도 있다.

우선, 콜백 함수부터 살펴보자!



# 콜백 함수로 여러 단계 연결하기

예) 피자 만들기: 피자 도우 준비하기 → 토핑 올리기 → 굽기

13WjsWpizza-1.js

step1이 끝나면 실행할 함수

```
const step1 = (callback) => {  
  setTimeout(() => {  
    console.log("피자 도우 준비");  
    callback();  
  }, 2000);  
}
```

step2가 끝나면 실행할 함수

```
const step2 = (callback) => {  
  setTimeout(() => {  
    console.log("토핑 완료");  
    callback();  
  }, 1000);  
}
```

# 콜백 함수로 여러 단계 연결하기

step3가 완료되면 실행할 함수

```
const step3 = (callback) => {  
  setTimeout(() => {  
    console.log("굽기 완료");  
    callback();  
  }, 2000);  
}
```

```
console.log("피자를 주문합니다.");  
step1(function() {  
  step2(function() {  
    step3(function() {  
      console.log("피자가 준비되었습니다. 🍕");  
    });  
  });  
});
```

피자를 주문합니다.	<a href="#">pizza-1.js:22</a>
피자 도우 준비	<a href="#">pizza-1.js:3</a>
토핑 완료	<a href="#">pizza-1.js:10</a>
굽기 완료	<a href="#">pizza-1.js:17</a>
피자가 준비되었습니다. 🍕	<a href="#">pizza-1.js:26</a>



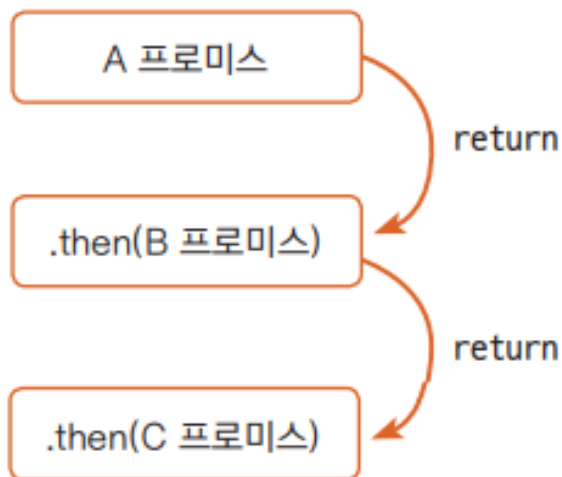
# 프로미스 체이닝으로 여러 단계 연결하기

프로미스는 resolve와 reject를 사용해서 성공과 실패에 대한 동작을 명확하게 구별할 수 있고, 함수에 계속해서 함수를 포함시키지 않기 때문에 콜백 지옥을 벗어날 수 있는 좋은 방법이다.

(예) A, B, C 각각 실행 시간이 다르지만 A 작업이 끝날 때까지 기다렸다가 B 작업을 하고, 다시 B 작업이 끝날 때 까지 기다렸다가 C 작업을 해야 한다면?

```
A.then(B).then(C)
```

then()을 사용해 여러 개의 프로미스를 연결하는 것을 '**프로미스 체이닝**'이라고 한다.



# 프로미스 반환 연습해 보기

13\js\pizza-2.js

- 1) 프로미스 제작 코드 - 다음 프로미스로 연결하기 위해 return문을 사용해 프로미스를 반환한다.

```
const pizza = () => {  
  return new Promise((resolve, reject) => {  
    resolve("피자를 주문합니다.");  
  });  
};
```

(여기에서는 resolve만 실행해 봄)

- 2) 웹 브라우저 콘솔 창에서 pizza() 함수를 실행했을 때 어떤 값이 반환되는지 확인한다.

```
pizza()
```

```
> pizza()  
◀ ▶ Promise {<fulfilled>: '피자를 주문합니다.'}  
>
```

← pizza() 함수를 실행하면 프로미스가 반환된다.

### 3) 프로미스 소비 코드 작성

pizza() 함수를 통해 프로미스를 만들고 resolve를 처리하기 위해 then()을 연결한다.

```
const pizza = () => {  
  return new Promise((resolve, reject)  
=> {  
    resolve("피자를 주문합니다.");  
  });  
};
```

```
const step1 = (message) => {  
  console.log(message);  
};
```

```
pizza().then(result => step1(result));
```

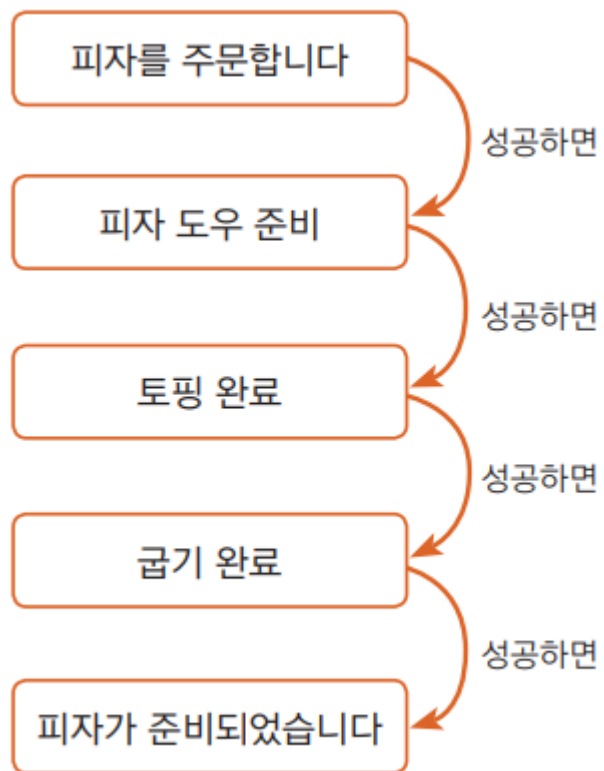
무슨 값이 담겼을까?

넘겨받은 결과값을 처리하는 step1() 함수

피자를 주문합니다.

pizza-2.js:8





각 단계마다 이전 단계가 완료되어야 함  
→ 프로미스를 사용해 연결하자

```
const pizza = () => {  
  return new Promise((resolve, reject)  
=> {  
    resolve("피자를 주문합니다.");  
  });  
};
```

```
const step1 = (message) => {  
  console.log(message);  
  return new Promise((resolve, reject)  
=> {  
    setTimeout(() => {  
      resolve('피자 도우 준비');  
    }, 3000);  
  });  
};
```

```
const step2 = (message) => {  
  console.log(message);  
  return new Promise((resolve, reject) => {  
    setTimeout(() => {  
      resolve('토핑 완료');  
    }, 1000);  
  });  
};
```

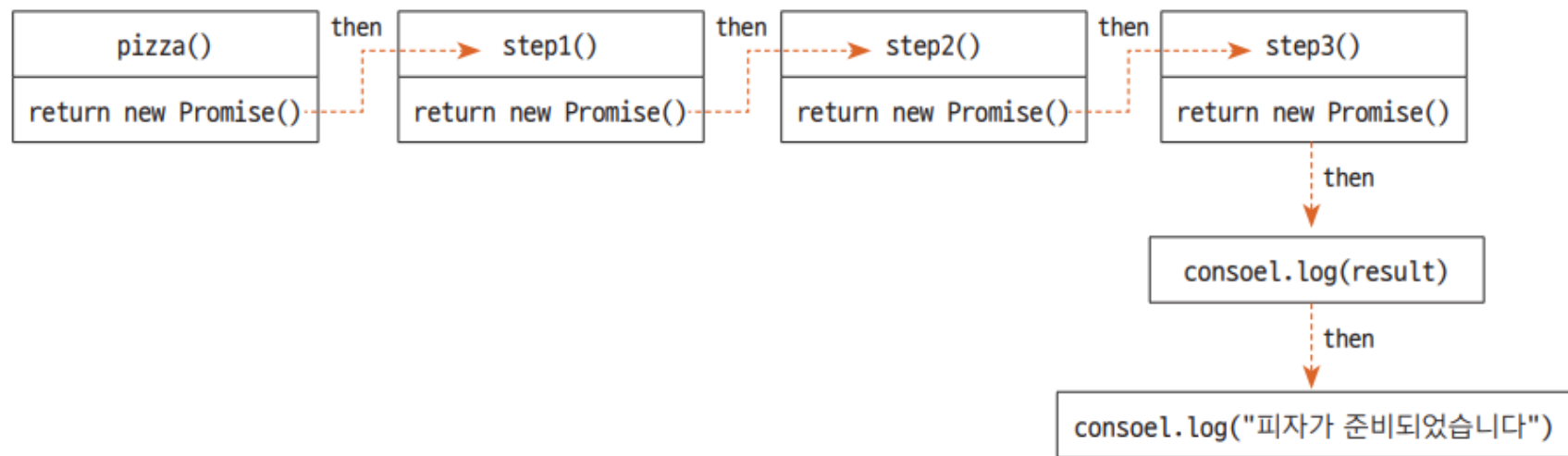
↑ 1초 후에 다음 resolve 처리

```
const step3 = (message) => {  
  console.log(message);  
  return new Promise((resolve, reject) => {  
    setTimeout(() => {  
      resolve('굽기 완료');  
    }, 2000);  
  });  
};
```

↑ 2초 후에 다음 resolve 처리

`pizza()`

```
.then((result) => step1(result))    // pizza()가 성공하면 step1() 실행
.then((result) => step2(result))    // step1()이 성공하면 step2() 실행
.then((result) => step3(result))    // step2()이 성공하면 step3() 실행
.then((result) => console.log(result)) // step3()이 성공하면 "굽기 완료" 표시
.then(() => {
  console.log('피자가 준비되었습니다. 🍕');
});
```





# 프로미스 소비 코드 줄여쓰기

앞의 소스에서, `then()` 함수는 이전 프로미스의 결과값을 받아서 다른 함수로 연결한다.


```
pizza()  
  .then(result => step1(result))
```

위 소스를 다음과 같이 줄여서 쓸 수 있다.

```
pizza()  
  .then(step1)
```

앞의 피자 만들기 소스에서 프로미스 소비 코드를 줄이면.

```
pizza()  
  .then((result) => step1(result))  
  .then((result) => step2(result))  
  .then((result) => step3(result))  
  .then((result) => console.log(result))  
  .then(() => {  
    console.log('피자가 준비되었습니다. 🍕');  
  });
```



```
pizza()  
  .then(step1)  
  .then(step2)  
  .then(step3)  
  .then(console.log)  
  .then(() => {  
    console.log("피자가 준비되었습니다. 🍕");  
  });
```

---

# fetch API

---

# fetch API란

- 서버에 자료를 요청하거나 자료를 받아올 때 사용하는 API
- XMLHttpRequest를 대신한다.
- fetch는 프로미스를 반환한다!

`fetch(위치, 옵션)`

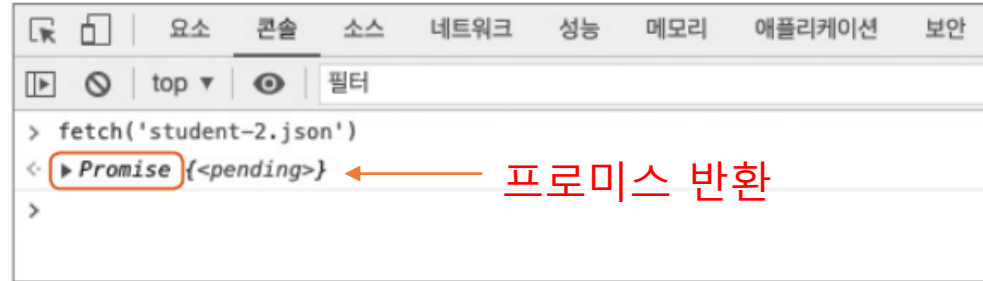
- 위치: 자료가 있는 URL이나 파일 이름
- 옵션: GET이나 POST 같은 요청 방식 지정 (따로 지정하지 않으면 GET 메서드 사용)

1. VS Code에서 13wfetch.html을 열고 라이브 서버를 사용해서 웹 브라우저에 문서를 표시한다.

(직접 탐색기에서 13wfetch.html을 열면 안 된다!!)

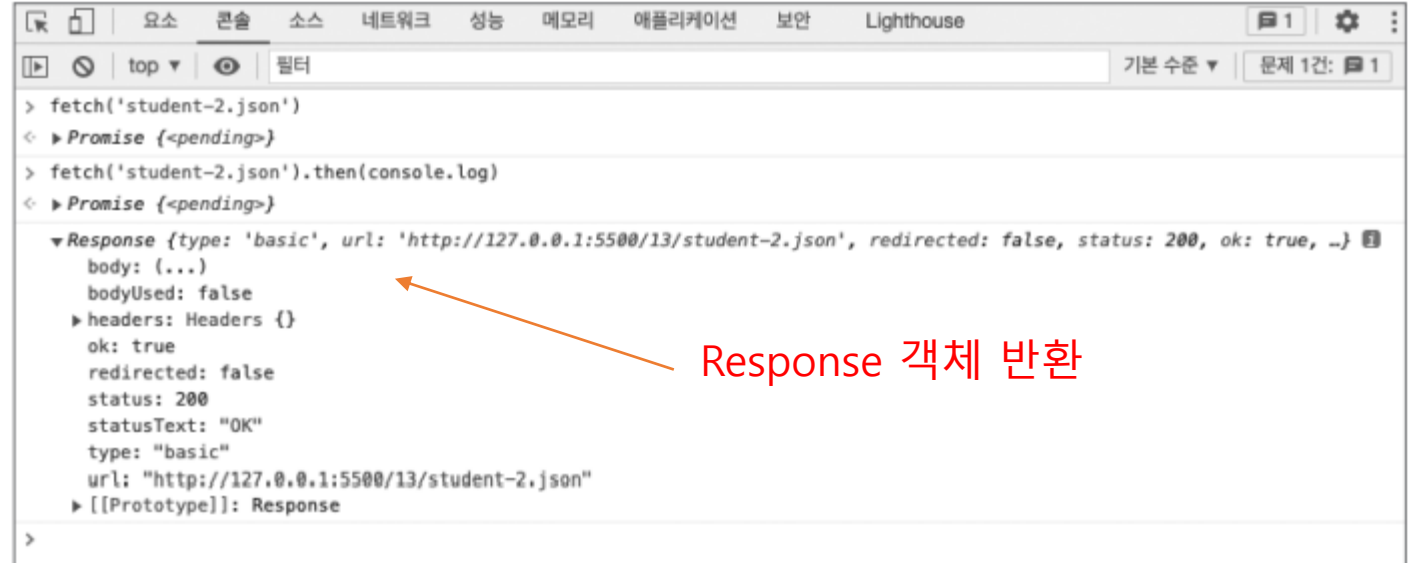
2. 콘솔 창 을 열고 다음과 같이 입력

```
fetch(student-2.json)
```



3. then()을 사용해 결과 표시

```
fetch(student-  
2.json).then(console.log)
```



XMLHttpRequest 객체 대신 fetch  
API를 사용해 보자

13WjsWfetch.js

수강생 명단	
도레미	<ul style="list-style-type: none"><li>전공: 컴퓨터 공학</li><li>학년: 2</li></ul>
백두산	<ul style="list-style-type: none"><li>전공: 철학과</li><li>학년: 1</li></ul>
홍길동	<ul style="list-style-type: none"><li>전공: 국문학과</li><li>학년: 3</li></ul>

```
fetch('student-2.json')
  .then(response => response.json())
  .then(json => {
    let output = '';
    json.forEach(student => {
      output += `
        <h2>${student.name}</h2>
        <ul>
          <li>전공 : ${student.major}</li>
          <li>학년 : ${student.grade}</li>
        </ul>
        <hr>
      `;
    });
    document.querySelector('#result').innerHTML =
output;
  })
  .catch(error => console.log(error));
```

---

# **async와 await**

---

# async 함수

- 프로미스는 콜백 지옥이 생기지 않도록 소스를 읽기 쉽게 바꾼 것
- 프로미스 체이닝은 프로미스를 계속 연결해서 사용하기 때문에 콜백 지옥처럼 소스가 복잡해질 수도 있다  
→ 이런 문제를 줄이기 위해 `async` 함수와 `await` 예약어 등장

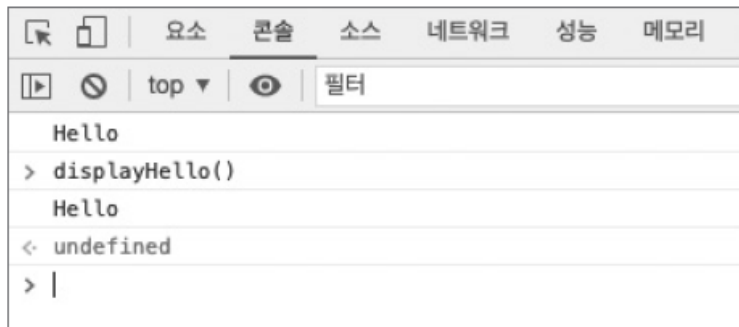
`async`라는 예약어를 함께 사용하면 그 함수 안에 있는 명령을 비동기적으로 실행할 수 있다.

기본형 `async function() { ... }`



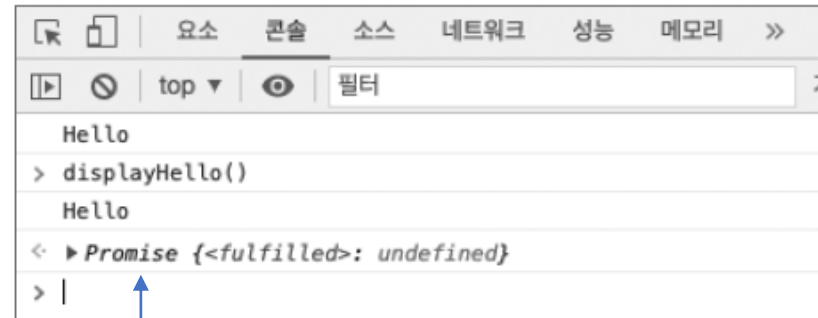
## 일반 함수

```
function displayHello() {  
  console.log("Hello");  
}  
displayHello();
```



## async 함수

```
async function displayHello()  
{  
  console.log("Hello");  
}  
displayHello();
```



async() 함수는 프로미스 반환

(예) 사용자가 좋아하는 주제를 지정하면 Hello와 함께 주제를 화면에 표시하는 프로그램

## 일반 함수

```
<script>
function whatsYourFavorite() {
  let fav = "Javascript";
  return new Promise((resolve, reject) =>
    resolve(fav));
}
function displaySubject(subject) {
  return new Promise((resolve, reject) =>
    resolve(`Hello, ${subject}`));
}
whatsYourFavorite()
  .then(displaySubject)
  .then(console.log);
</script>
```

## async 함수

```
async function whatsYourFavorite() {
  let fav = "Javascript";
  return fav;
}
async function displaySubject(subject)
{
  return `Hello, ${subject}`;
}
whatsYourFavorite()
  .then(displaySubject)
  .then(console.log);
```

# await

- 프로미스 체이닝을 좀더 쉽게 작성
- async 함수에서만 사용할 수 있다.

whatsYourFavorite() 함수 처리에 시간이 얼마나  
걸리든 기다렸다가 결과값을 response에 저장

```
async function whatsYourFavorite() {  
  let fav = "Javascript";  
  return fav;  
}  
  
async function displaySubject(subject) {  
  return `Hello, ${subject}`;  
}  
  
async function init() {  
  const response = await whatsYourFavorite();  
  const result = await displaySubject(response);  
  
  console.log(result);  
}  
  
init();
```