Javascript 동작 원리와 비동기

18번째 세션

NEXT X LIKELION 양효령

목차

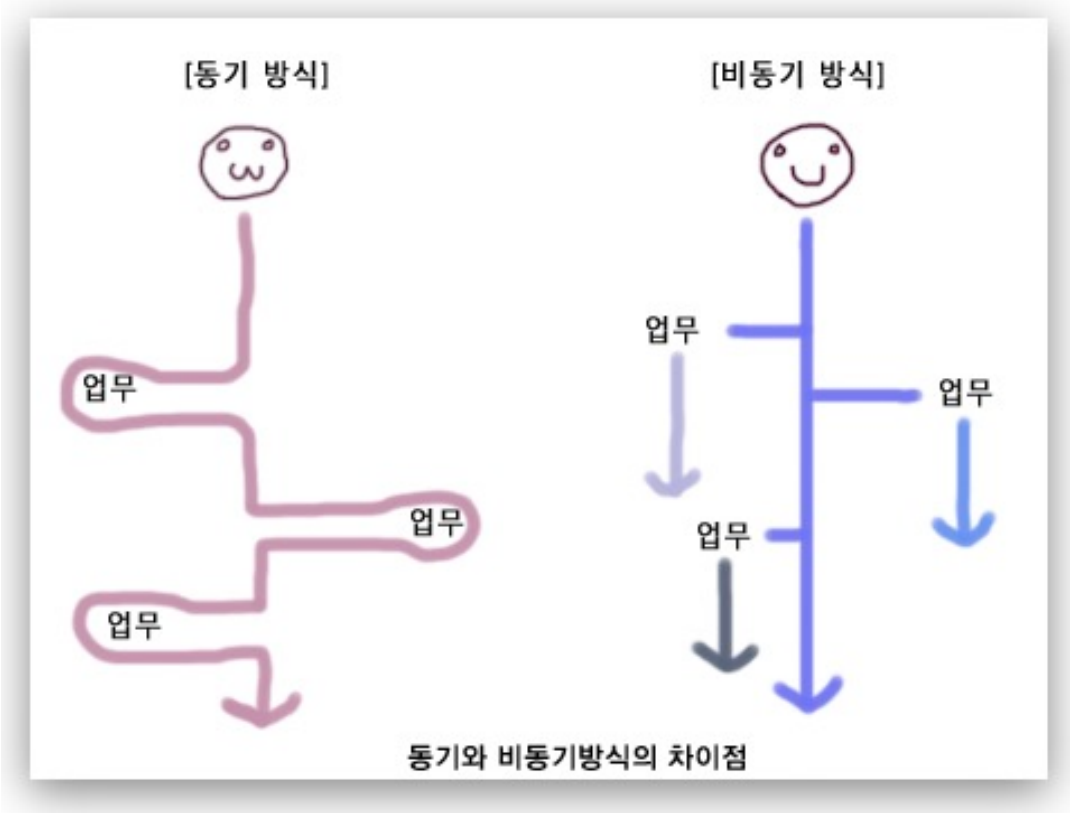
1. 자바스크립트 동작 원리
2. 자바스크립트 비동기
    - callback
    - promise
    - async / await

자바스크립트 동작 원리

NEXT X LIKELION

동기 vs 비동기

- **동기(synchronous)**
  - 요청과 결과가 한 자리에서 동시에 일어남
  - 요청을 하면 시간이 얼마나 걸려도 요청한 자리에서 결과가 주어짐
- **비동기(asynchronous)**
  - 요청과 결과가 동시에 일어나지 않음
  - 요청 내용에 대해서 바로 응답을 받지 않아도 됨



동기와 비동기방식의 차이점

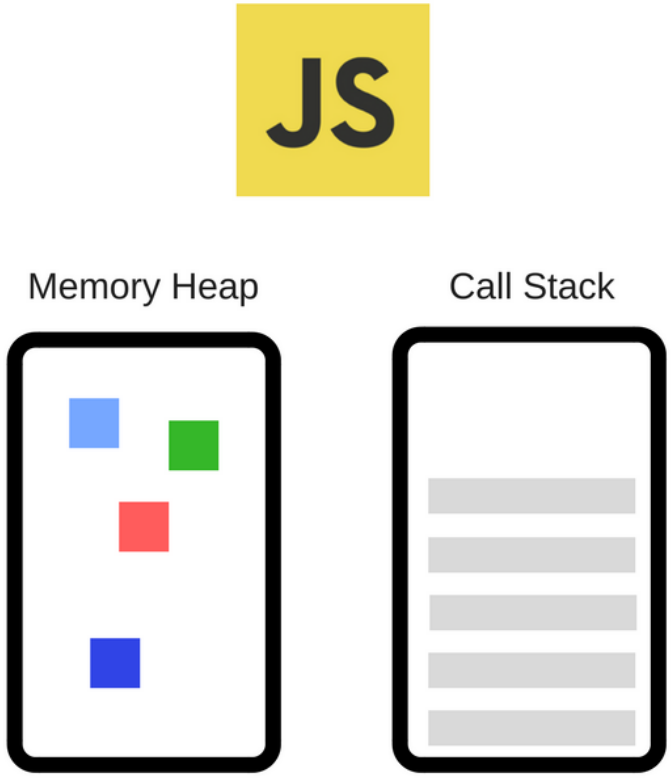
자바스크립트 동작 원리

싱글 스레드 기반 동기적 언어

: 한 번에 하나의 작업만 순차적으로 수행함

그럼 어떻게 비동기 작업이 가능하지?

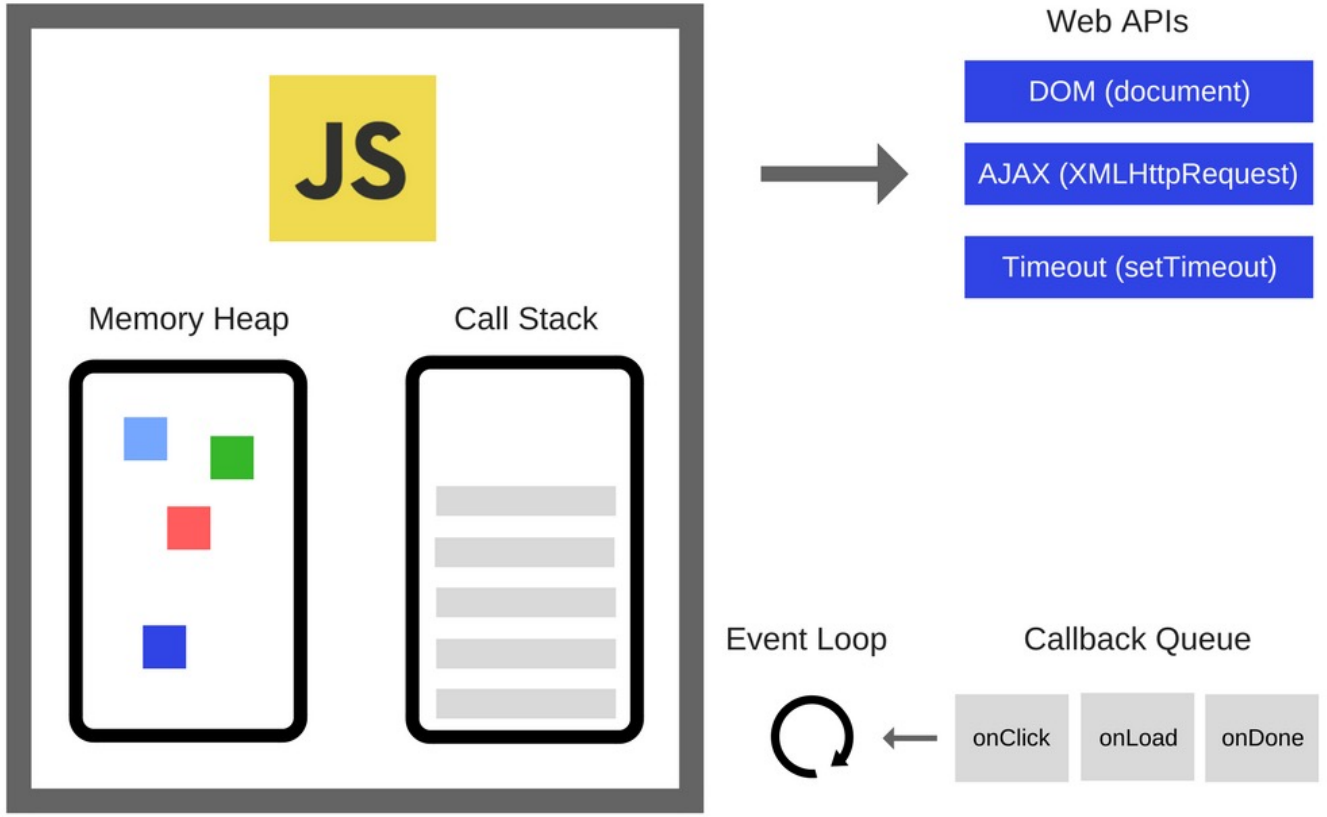
자바스크립트 엔진



- 자바스크립트 코드를 이해하고 실행을 도와줌
- 대표적으로 V8
- 구성
  - **Memory Heap**: 메모리 할당이 일어나는 곳, 데이터를 임시 저장하는 곳으로, 함수나 변수, 함수를 실행할 때 사용하는 값들을 저장
  - **Call Stack**: 코드 실행에 따라 호출 스택이 쌓이는 곳, 코드가 실행되면 코드의 내부의 실행 순서를 기록해 놓고, 하나씩 순차적으로 진행할 수 있도록 도와주는 곳

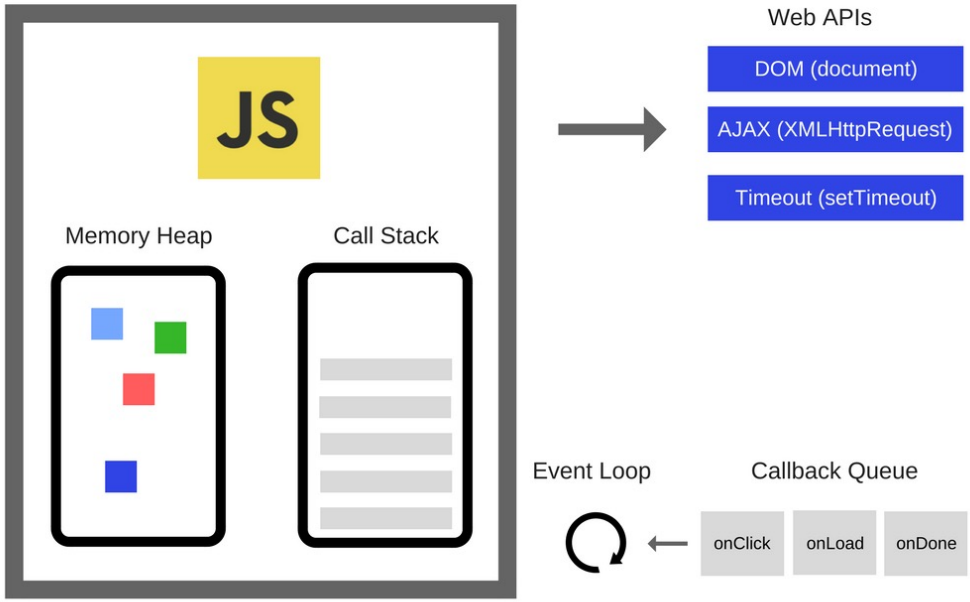
자바스크립트 런타임

- 자바스크립트가 구동되는 환경
- 크롬 등의 브라우저, Node.js 등
- 자바스크립트 엔진 밖에서 자바스크립트에 관여하는 요소
  - Web API
  - Task Queue
  - Event Loop



자바스크립트 런타임

- **Web API**
  - Web API는 브라우저에서 제공되는 API
  - 자바스크립트 엔진에서 정의되지 않았던 setTimeout이나 HTTP 요청(ajax) 메소드, DOM 이벤트 등의 메소드를 지원
- **Task Queue**
  - 이벤트 발생 후 호출되어야 할 콜백 함수들이 기다리는 공간
  - 이벤트 루프가 정한 순서대로 줄을 서 있으므로 콜백 큐(Callback Queue) 라고도 함
- **Event Loop**
  - 이벤트 발생 시 호출할 콜백 함수들을 관리하고, 호출된 콜백 함수의 실행 순서를 결정



자바스크립트 동작 원리

자바스크립트는 동기적으로 동작하는데 어떻게 비동기 작업이 가능하지?

자바스크립트 자체는 싱글 스레드로 동기적으로 동작하는 언어이지만,

자바스크립트 런타임에서 비동기 작업이 가능함

자바스크립트 비동기 처리

NEXT X LIKELION

자바스크립트 비동기

- **자바스크립트의 비동기 처리**
  - 특정 코드가 종료되지 않았어도 대기하지 않고 다음 코드를 실행
- **자바스크립트에서 비동기 처리가 필요한 이유**
  - 화면에서 서버로 데이터를 요청했을 때 서버가 언제 그 요청에 대한 응답을 할지도 모르는 상태에서 다른 코드를 실행 안하고 기다릴 수는 없기 때문에

하지만 실행 순서가 중요한 상황이 많음

-> **callback, promise, async/await**

Callback

- 다른 함수에 파라미터로 넘겨지는 함수
- 함수를 호출할 때 새로운 일이 생기거나 그 함수의 실행이 끝나면 지정한 콜백 함수를 실행해주도록 함수에 요청할 때 사용
- Ex) addEventListener, setTimeout, setInterval 등

```
function f(callback, ...){
  ...
  callback();
  ... }

f(a, ...);

// 이 코드에서 함수 f의 인자로 넘겨진 함수 a가 콜백함수이다.
```

Callback

- 예시: 3초 뒤에 '첫번째', '두번째' 가 순차적으로 콘솔에 찍힘

```
function first(callback){
  setTimeout(() => {
    console.log("첫번째");
    callback();
  }, 3000); }

function second(){
  console.log("두번째");
}

first(second);
// first(() => second()); 와 같음
// first(second()); 는 다름. Second함수를 호출한 결과를 인자로 넣는 것
```

Callback

- 콜백 지옥

```
const printAfterSecond = (text, callback) => {
  setTimeout(() => {
    console.log(text);
      callback && callback();
    }, 1000);
};

printAfterSecond("1초 경과", () =>
  printAfterSecond("2초 경과", () =>
    printAfterSecond("3초 경과", () =>
      printAfterSecond("4초 경과", () =>
        printAfterSecond("5초 경과", () =>
          console.log("완료")
        )
      )
    )
  )
);
```

Promise

- 어떤 함수가 내부에서 바로 실행되지 않고 어떤 약속을 맺어서 그 약속을 추후에 실행

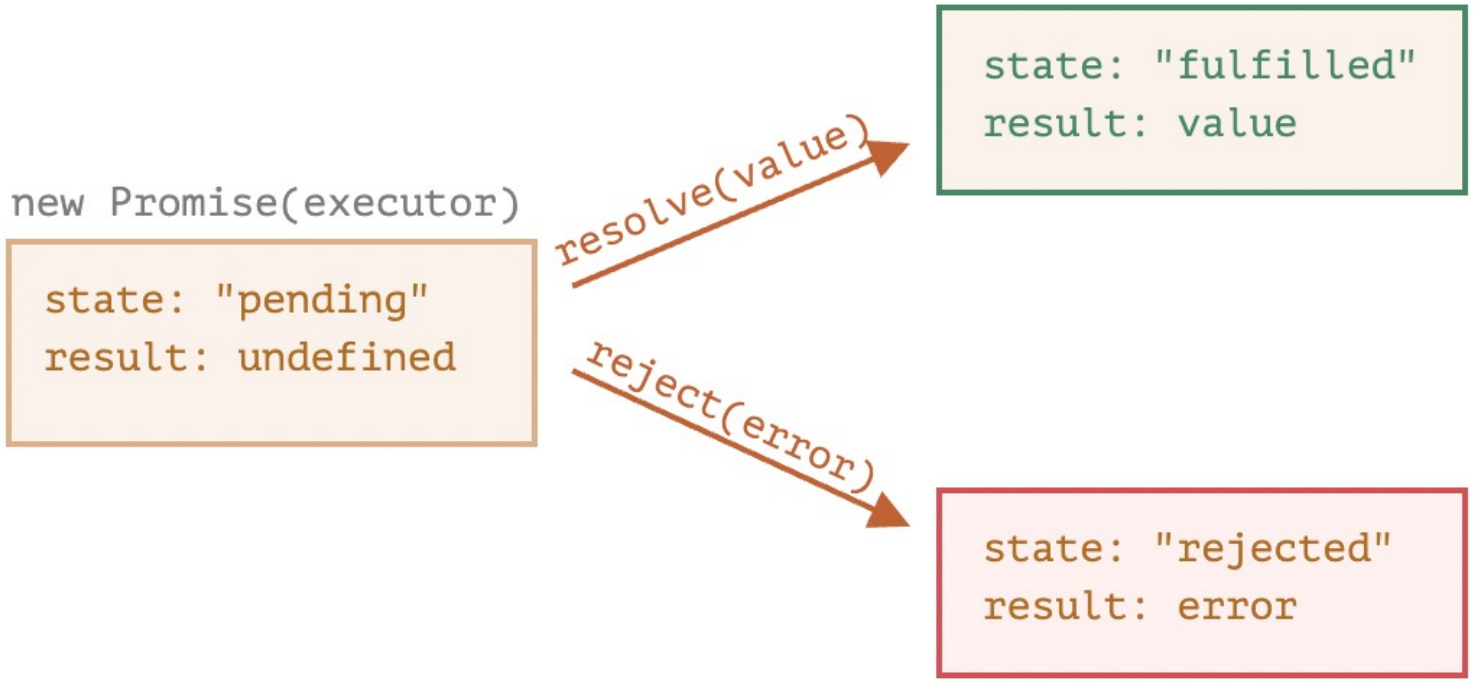
```
const promise = new Promise(function(resolve, reject){ ... });

// Arrow Function
const promise = new Promise((resolve, reject) => { ... });
```

- new Promise에 전달되는 함수는 executor(실행자, 실행 함수)
  - new Promise가 만들어질 때 자동으로 실행

- resolve(value) : 일이 성공적으로 끝난 경우 그 결과를 나타내는 value와 함께 호출
- reject(error) : 에러 발생 시 에러 객체를 나타내는 error와 함께 호출

Promise



상태	의미	구현
pending	비동기 처리가 아직 수행되지 않은 상태	resolve 또는 reject 함수가 아직 호출되지 않은 상태
fulfilled	비동기 처리가 수행된 상태 (성공)	resolve 함수가 호출된 상태
rejected	비동기 처리가 수행된 상태 (실패)	reject 함수가 호출된 상태
settled	비동기 처리가 수행된 상태 (성공 또는 실패)	resolve 또는 reject 함수가 호출된 상태

Promise - producer

- resolve

```
let promise = new Promise(function (resolve, reject) {
  // 프라미스가 만들어지면 executor 함수는 자동으로 실행됩니다.

  // 1초 뒤에 일이 성공적으로 끝났다는 신호가 전달되면서 result는 '완료'가 됩니다.
  setTimeout(() => resolve("완료"), 1000)
});
```

Promise - producer

- reject

```
let promise = new Promise(function (resolve, reject) {
  // 1초 뒤에 에러와 함께 실행이 종료되었다는 신호를 보냅니다.
  setTimeout(() => reject(new Error("에러 발생!")), 1000);
});
```

Promise - consumer

- Promise로 구현된 비동기 함수를 호출하는 측
- Promise 객체의 후속 처리 메소드(then, catch)를 통해 비동기 처리 결과 또는 에러 메시지를 전달받아 처리
- then

```
promise.then(
  function(result) { /* 결과(result)를 다룹니다 */ },
  function(error) { /* 에러(error)를 다룹니다 */ }
);
```

- catch

```
let promise = new Promise((resolve, reject) => {
  setTimeout(() => reject(new Error("에러 발생!")), 1000);
});

// .catch(f)는 promise.then(null, f)과 동일하게 작동합니다
promise.catch(alert); // 1초 뒤 "Error: 에러 발생!" 출력
```

Promise Chaining

- then 이나 catch 에서 다시 다른 then 이나 catch 를 붙일 수 있음
- 이전 then 의 return값을 다음 then 의 매개변수로 넘김
- 프로미스를 return한 경우 프로미스가 수행된 후 다음 then 또는 catch 가 호출됨

```
promise
.then((res) => res.json())
.then((res) => console.log(res))
.catch((err) => console.log(err));
```

(이미 다들 많이 해보셨습니다!)

Promise 실습1

금액을 입력받고, 잔액을 console로 찍어보자!(아래 함수 활용)

```
function buySomething(nowMoney) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const pay = parseInt(prompt("금액 입력"));
      const remain = nowMoney - pay;
      if (remain >= 0) {
        console.log(`${pay}원 지불`);
        resolve(remain);
      } else {
        reject(`잔액부족: 현재 잔액${nowMoney}원`);
      }
    }, 2000);
  });
}
```

콘솔창 예시(처음 금액이 1000원일 때)

- 입력한 금액은 200원일 때
  - 200원 지불
  - 잔액: 800원
- 입력한 금액은 1200원일 때
  - 1200원 지불
  - 잔액부족: 현재 잔액 1000원

Promise 실습2

잔액을 3번 이상 받도록 코드를 수정해보자!

콘솔창 예시(처음 금액이 1000원일 때)

- 200원 지불
- 잔액: 800원
- 400원 지불
- 잔액: 400원
- 500원 지불
- 잔액 부족: 현재 잔액 400원

async / await

- Promise로 콜백 지옥은 면했지만… 여전히 복잡하다!
- async와 await는 자바스크립트의 비동기 처리 패턴 중 가장 최근에 나온 문법
- 기존의 비동기 처리 방식인 콜백 함수와 프로미스의 단점을 보완하고 개발자가 읽기 좋은 코드를 작성할 수 있음

async / await

- **async**
  - function 앞에 async를 붙이면 해당 함수는 항상 프라미스를 반환
- **await**
  - 자바스크립트는 await 키워드를 만나면 프라미스가 처리될 때까지 기다림
  - 결과는 그 이후 반환
  - await는 async 함수 안에서만 동작

```
async function f() {
  let promise = new Promise((resolve, reject)
    => {
      setTimeout(() => resolve("완료!"), 1000);
  });

  // 프라미스가 이행될 때까지 기다림 (*)
  let result = await promise;
  alert(result); // "완료!"
}

f();
```

async / await 실습3

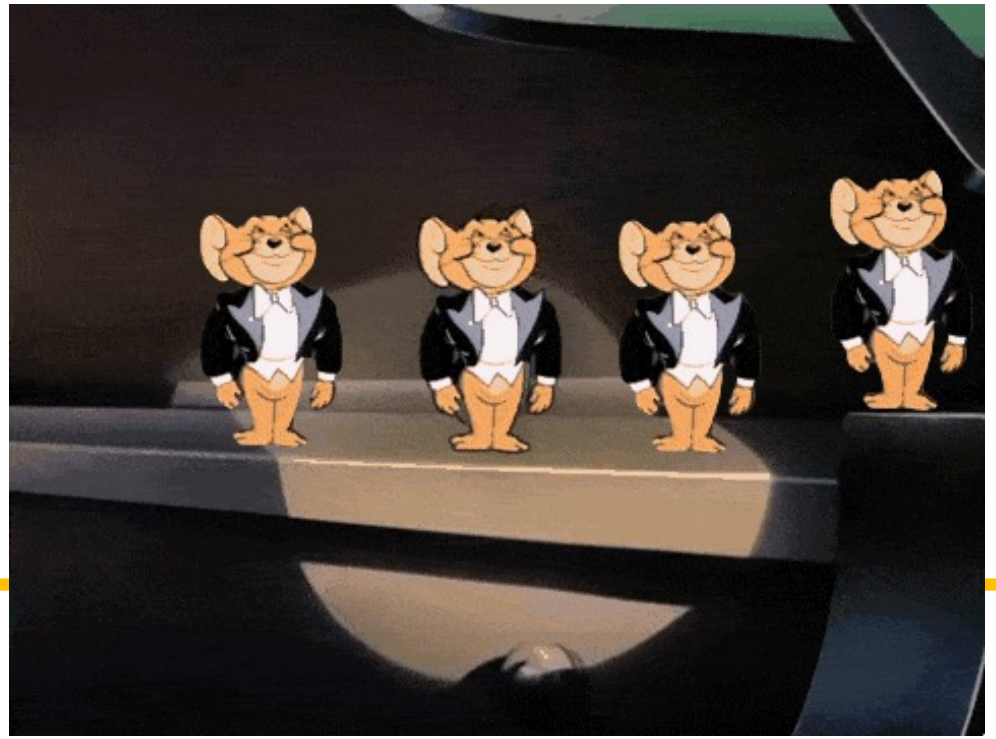
실습 1 코드를 async await으로 바꿔보자

```
function buySomething(nowMoney) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const pay = parseInt(prompt("금액 입력"));
      const remain = nowMoney - pay;
      if (remain >= 0) {
        console.log(`${pay}원 지불`);
        resolve(remain);
      } else {
        reject(`잔액부족: 현재 잔액${nowMoney}원`);
      }
    }, 2000);
  });
}
```

```
buySomething(1000)
  .then((remain) => {
    console.log(`잔액: ${remain}원`);
})
.catch((error) => {
    console.log(error);
  });
```

Frontend
Backend
기획 & 디자인

NEXT X LIKELION



NEXT X LIKELION 양효령