비동기, 이벤트루프, Promise

7/ 비동기, 이벤트루프, Promise

목차

01

동기, 비동기

동기, 비동기

동기 처리 방식: 직렬적으로 일을 수행

© CanStockPhoto.com - csp89908013

동기, 비동기

비동기 처리 방식: 병렬적으로 일을 수행

동기, 비동기

비동기 처리 방식: 병렬적으로 일을 수행

동기, 비동기

- 동기: 작업을 동시에 수행하거나, 동시에 끝나거나, **끝나는 동시에 시작함을 의미**
- 비동기: 시작, 종료가 일치하지 않으며, **끝나는 동시에 시작을 하지 않음**을 의미

➔ 요청 작업을 **순차적으로 처리하느냐 아니냐**
➔ 요청한 작업에 대한 완료 알림을 반드시 받아야 다음 작업을 수행한다는 것은 작업을 순서대로 처리한다는 것

➔ 동기 작업은 요청한 작업에 대해 **순서가 지켜지는 것**을 말하는 것이고,
비동기 작업은 **순서가 지켜지지 않을 수 있다는 것**

자바스크립트 특징

- 자바스크립트는 싱글 쓰레드로 동작하는 언어이다.
  - 싱글 쓰레드가 뭐지?
    - 일할 수 있는 손이 한 쌍이라고 생각을 하자
    - 다른 언어들은 멀티 쓰레드를 제공함

js: 난 손이 한 쌍인걸?

java: 난 멀티쓰레드를 지원해서 손이 여러개임

자바스크립트 특징

- 자바스크립트는 오래 걸리거나 복잡한 작업을 수행할 때 어떻게 잘 처리할 수 있을까?

HTTP 요청 →

타이머 함수 →

← 이벤트 핸들러

← 그 외 작업

js: 이걸 어떻게 처리하지?..

03

이벤트 루프

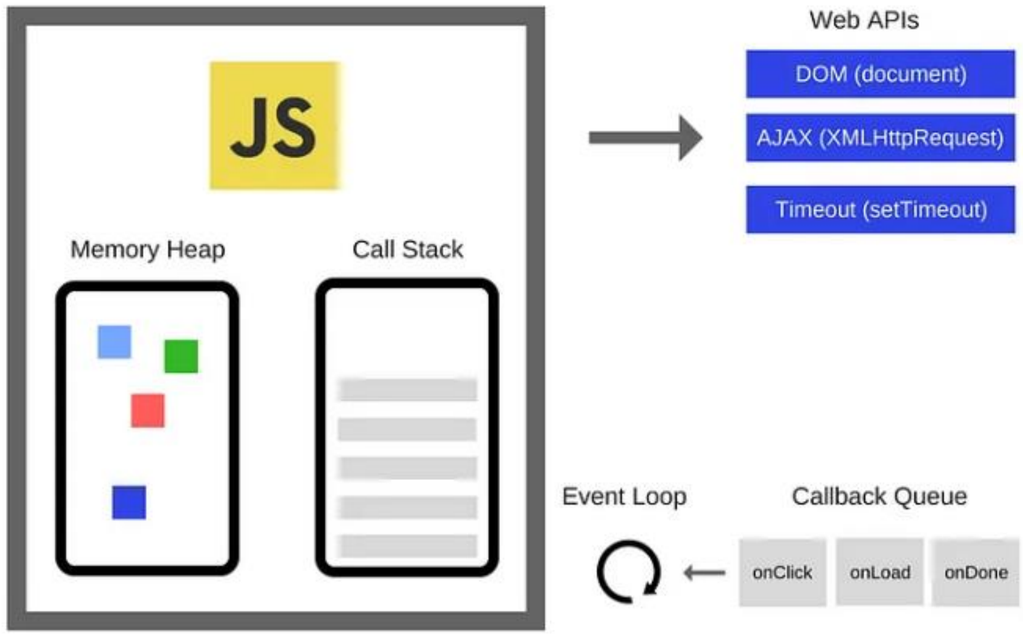
이벤트 루프

- 일반적으로 자바스크립트는 콜 스택에 쌓인 작업을 순차적으로 실행한다.
- 이때, 비동기 작업(HTTP 요청, 타이머 작업, 이벤트 핸들러)는 이벤트 루프로 처리가 된다.
  - 이벤트 루프는 비동기 작업을 처리하기 위한 메커니즘, 동작 방식
  - 이벤트 루프는 자바스크립트 엔진과 자바스크립트가 구동되는 환경이 서로 상호 연동하기 위한 장치

JS: 난 이벤트 루프를 통해서
비동기 작업을 처리하지!

동기, 비동기

이벤트 루프


JS
Memory Heap
Call Stack
Web APIs
DOM (document)
AJAX (XMLHttpRequest)
Timeout (setTimeout)
Event Loop
Callback Queue
onClick
onLoad
onDone

이벤트 루프

- 자바스크립트 코드는 브라우저 또는 Node.js에 의해 실행이 된다.

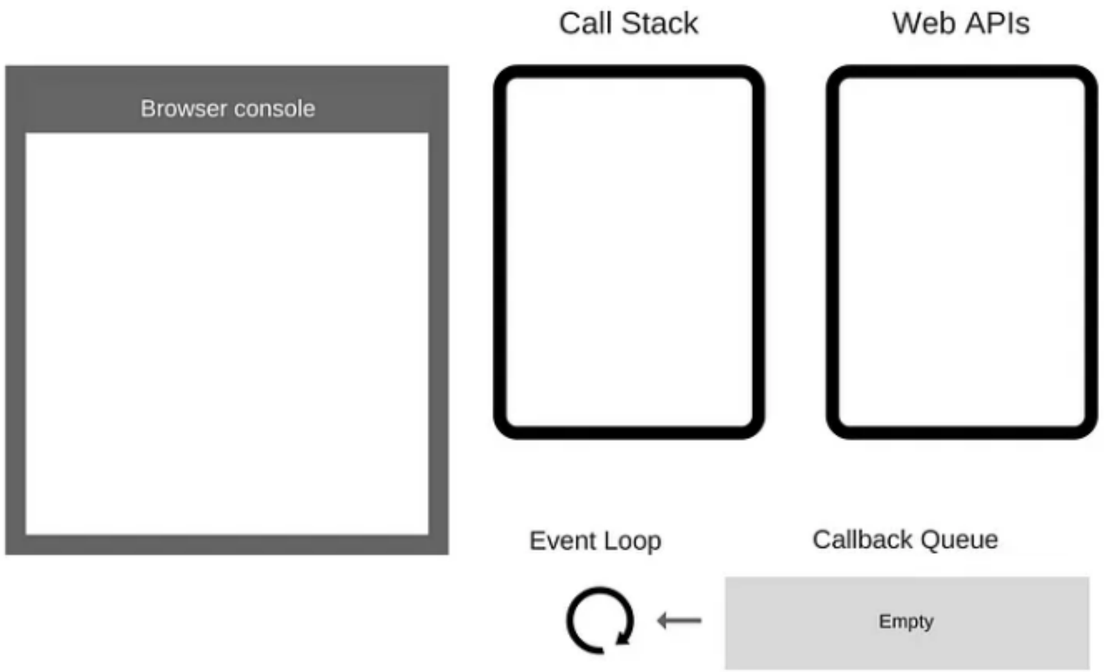
브라우저들: 한번 실행 시켜볼까?

```
> console.log(1+1);
  setTimeout(function cb1() { console.log(2+2) }, 1000);
  console.log(3+3);
```

```
2
6
< undefined
4
```

동기, 비동기

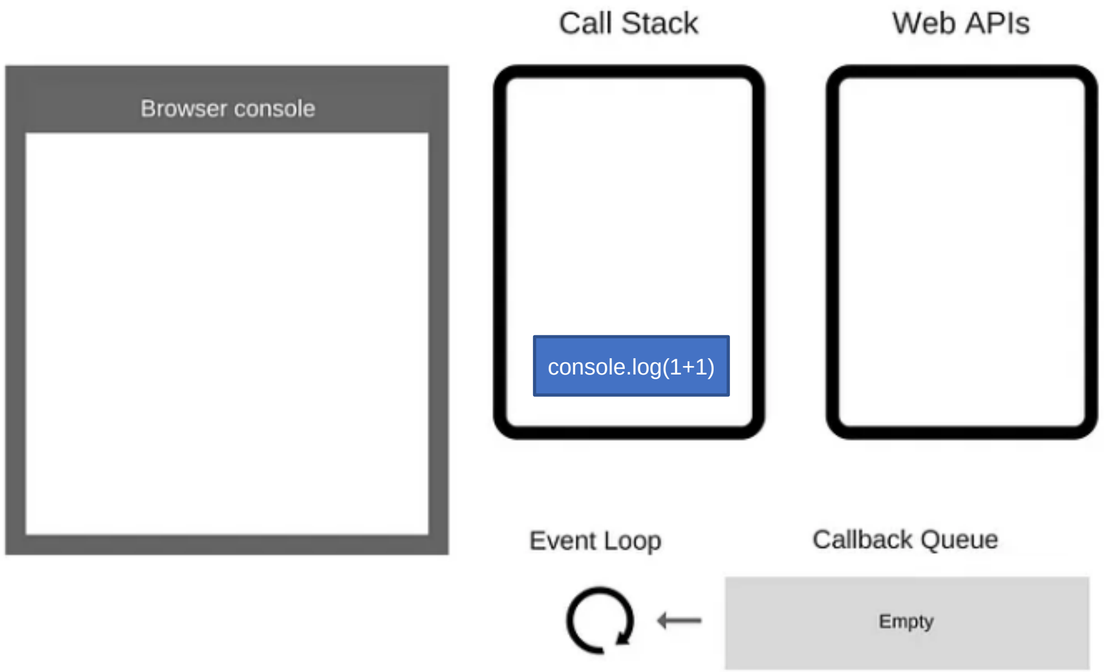
이벤트 루프




```
> console.log(1+1);
  setTimeout(function cb1() { console.log(2+2) }, 1000);
  console.log(3+3);
```

동기, 비동기

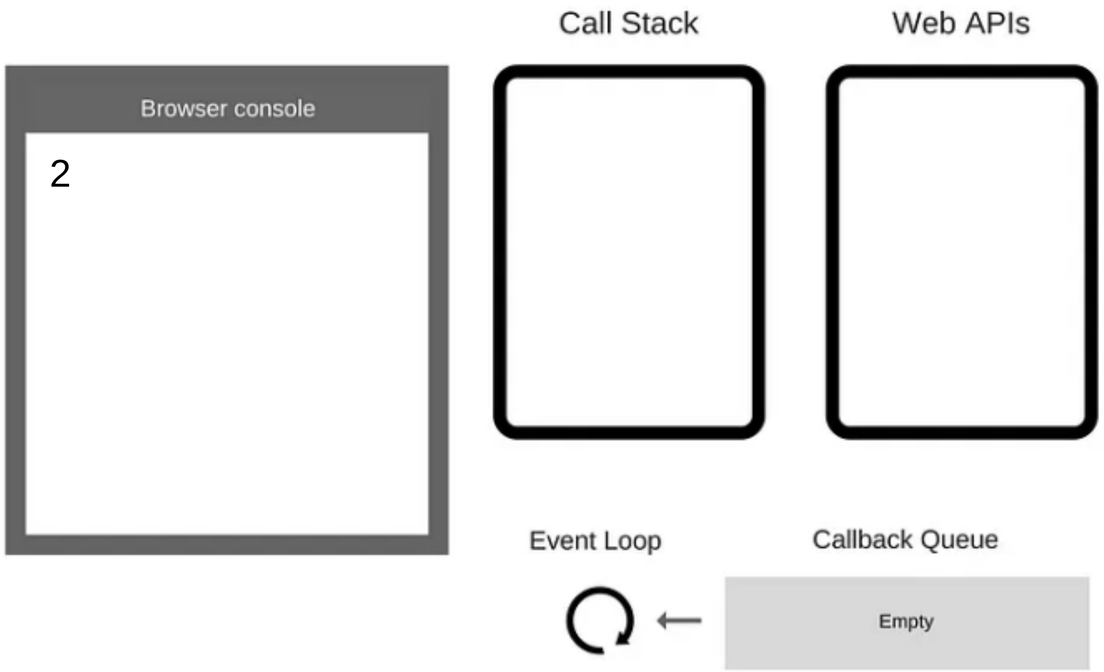
이벤트 루프




```
console.log(1+1);
setTimeout(function cb1() { console.log(2+2) }, 1000);
console.log(3+3);
```

동기, 비동기

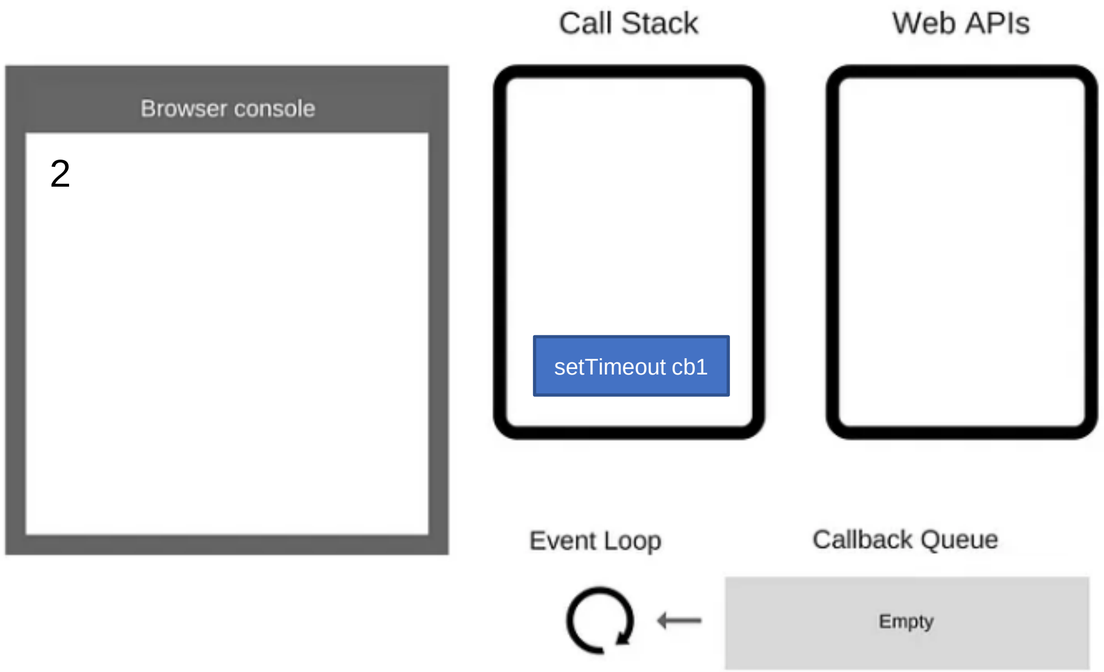
이벤트 루프




```
console.log(1+1);
setTimeout(function cb1() { console.log(2+2) }, 1000);
console.log(3+3);
```

동기, 비동기

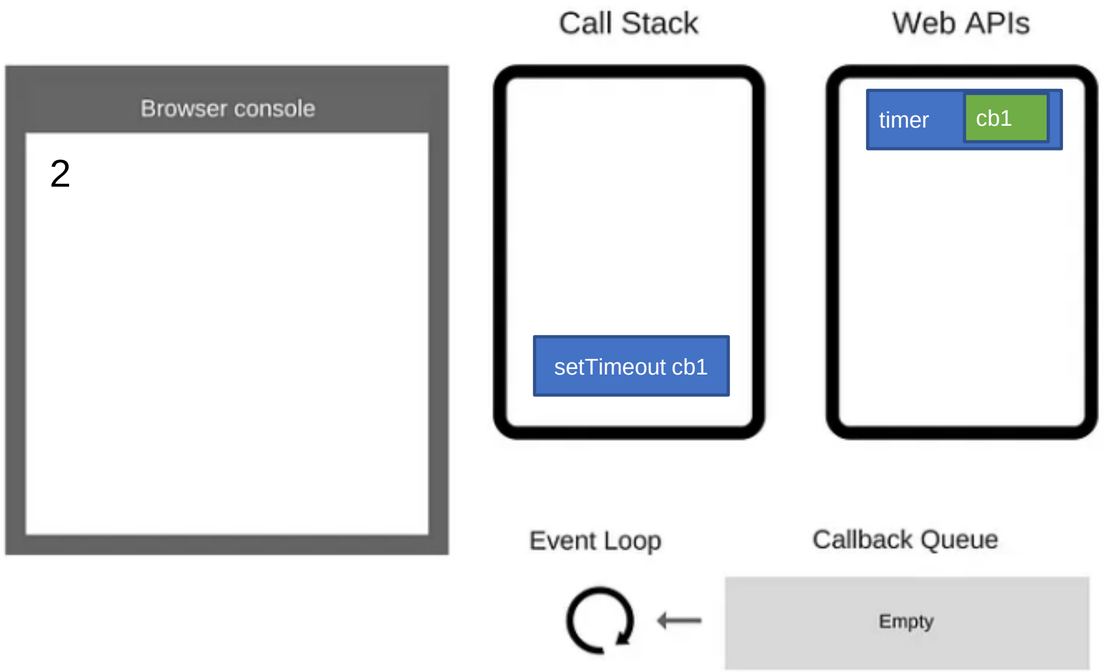
이벤트 루프




```
> console.log(1+1);
setTimeout(function cb1() { console.log(2+2) }, 1000);
console.log(3+3);
```

동기, 비동기

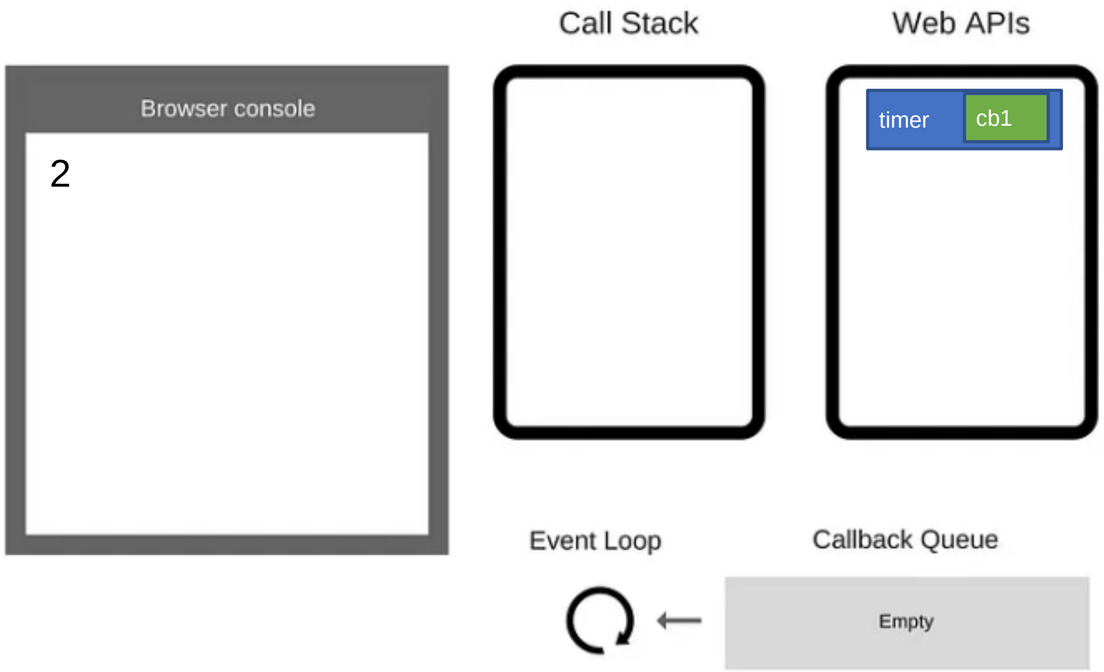
이벤트 루프




```
> console.log(1+1);
setTimeout(function cb1() { console.log(2+2) }, 1000);
console.log(3+3);
```

동기, 비동기

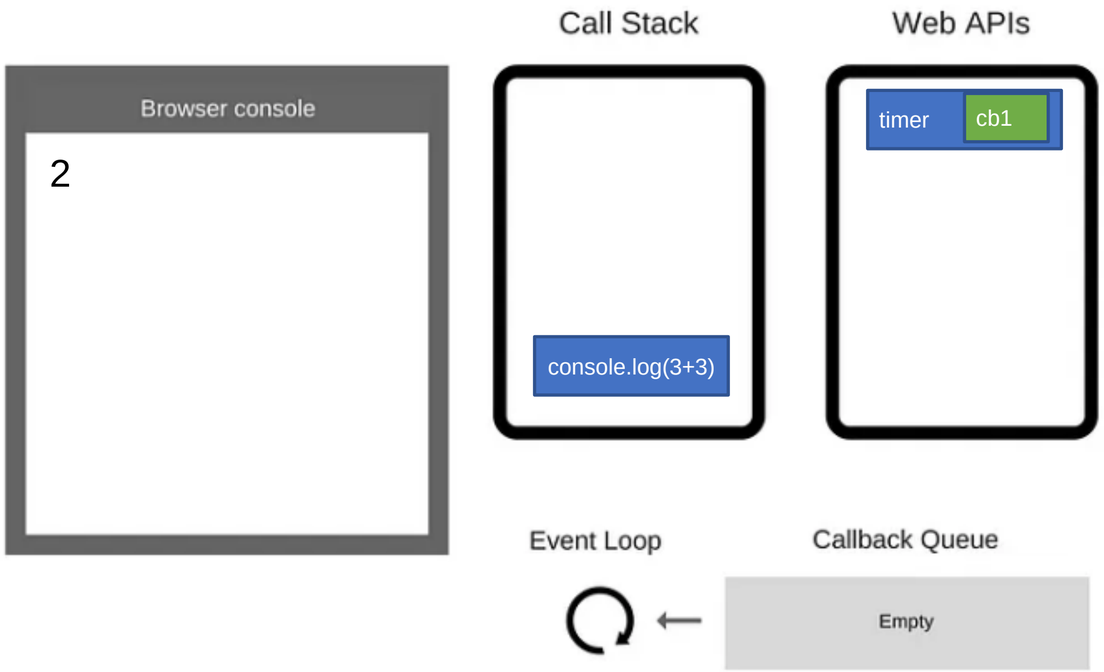
이벤트 루프




```
> console.log(1+1);
setTimeout(function cb1() { console.log(2+2) }, 1000);
console.log(3+3);
```

동기, 비동기

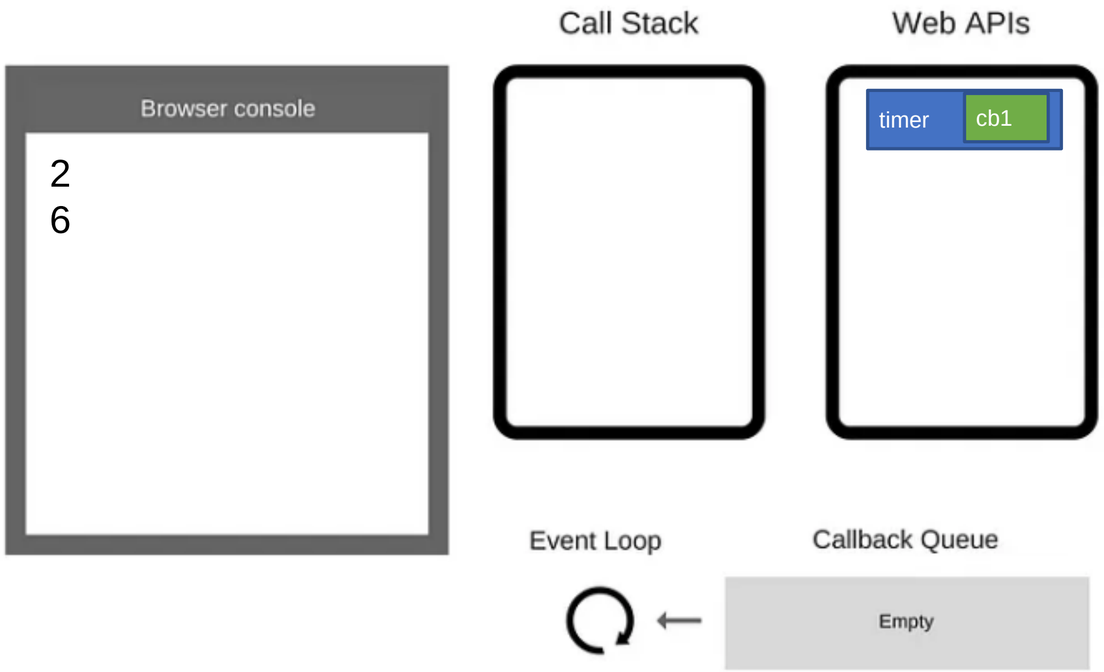
이벤트 루프




```
> console.log(1+1);
  setTimeout(function cb1() { console.log(2+2) }, 1000);
  console.log(3+3);
```

동기, 비동기

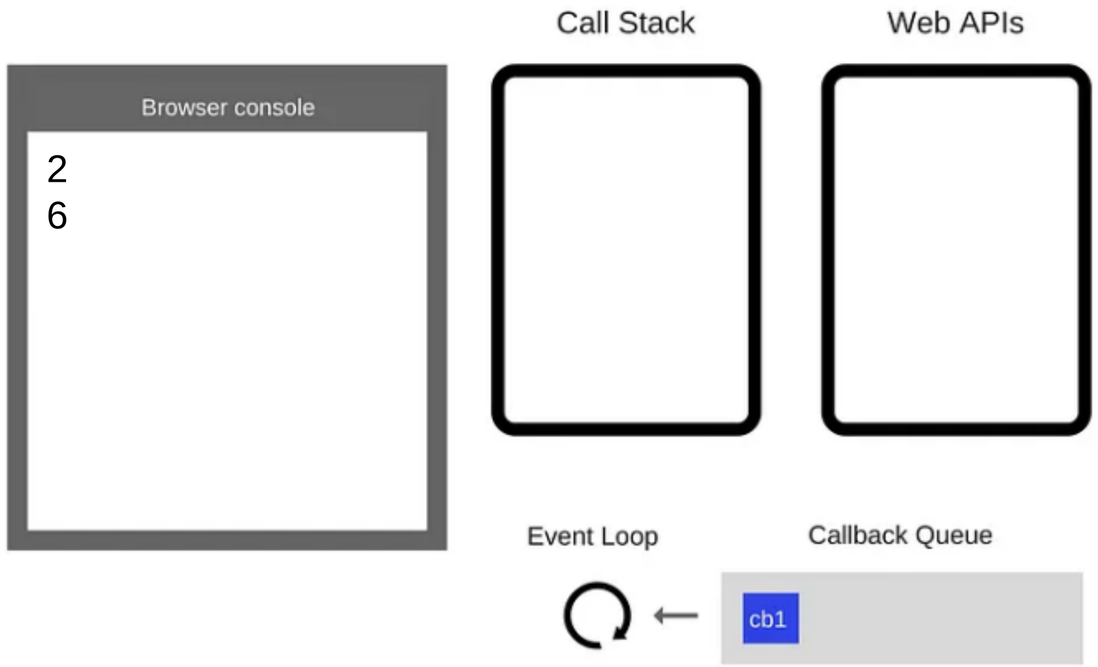
이벤트 루프




```
> console.log(1+1);
  setTimeout(function cb1() { console.log(2+2) }, 1000);
  console.log(3+3);
```

동기, 비동기

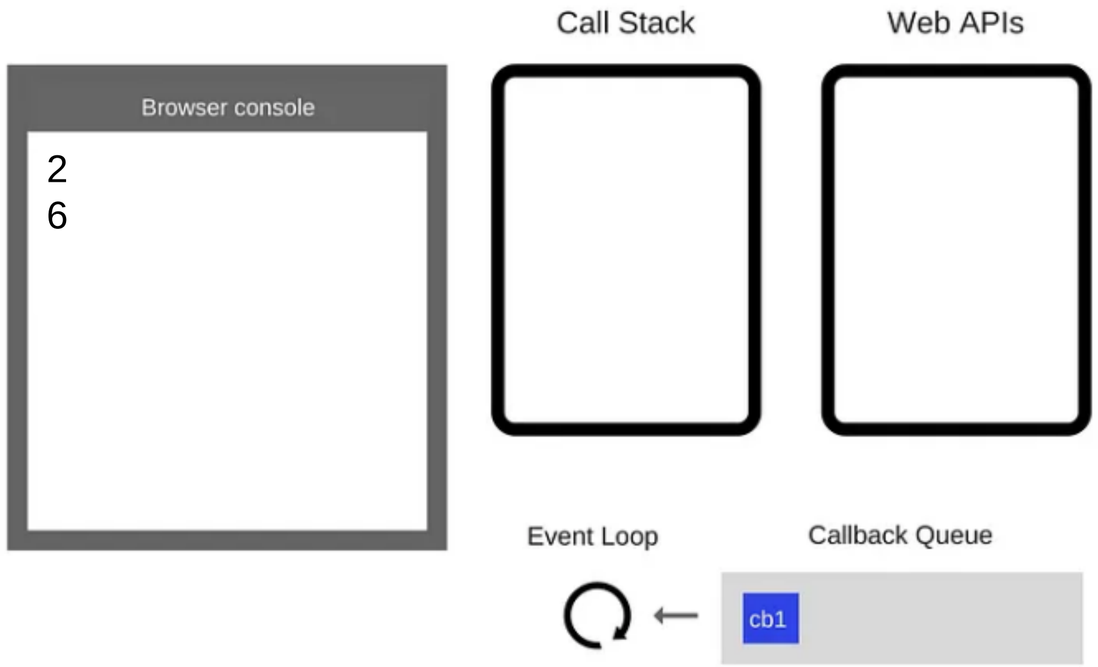
이벤트 루프




```
> console.log(1+1);
  setTimeout(function cb1() { console.log(2+2) }, 1000);
  console.log(3+3);
```

동기, 비동기

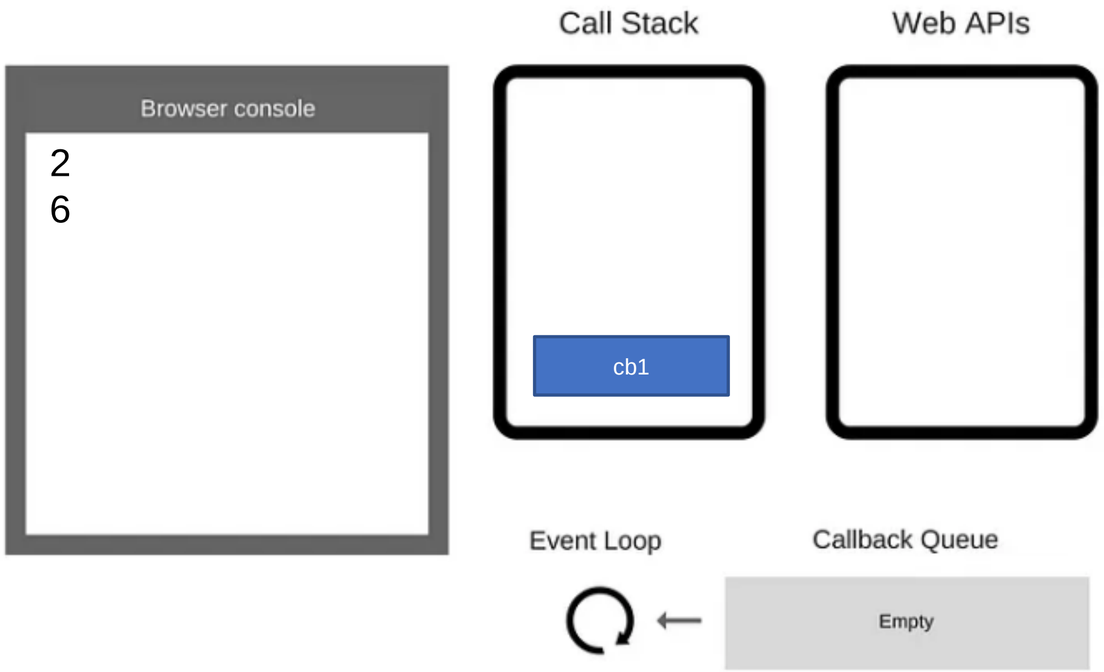
이벤트 루프




```
> console.log(1+1);
  setTimeout(function cb1() { console.log(2+2) }, 1000);
  console.log(3+3);
```

동기, 비동기

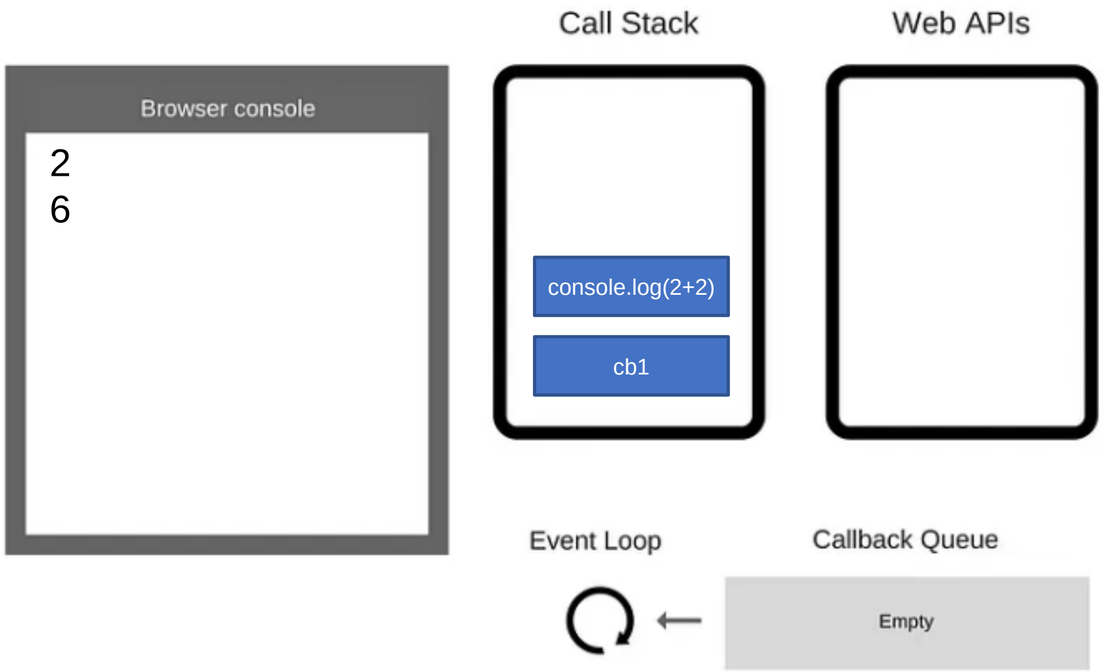
이벤트 루프




```
> console.log(1+1);
setTimeout(function cb1() { console.log(2+2) }, 1000);
console.log(3+3);
```

동기, 비동기

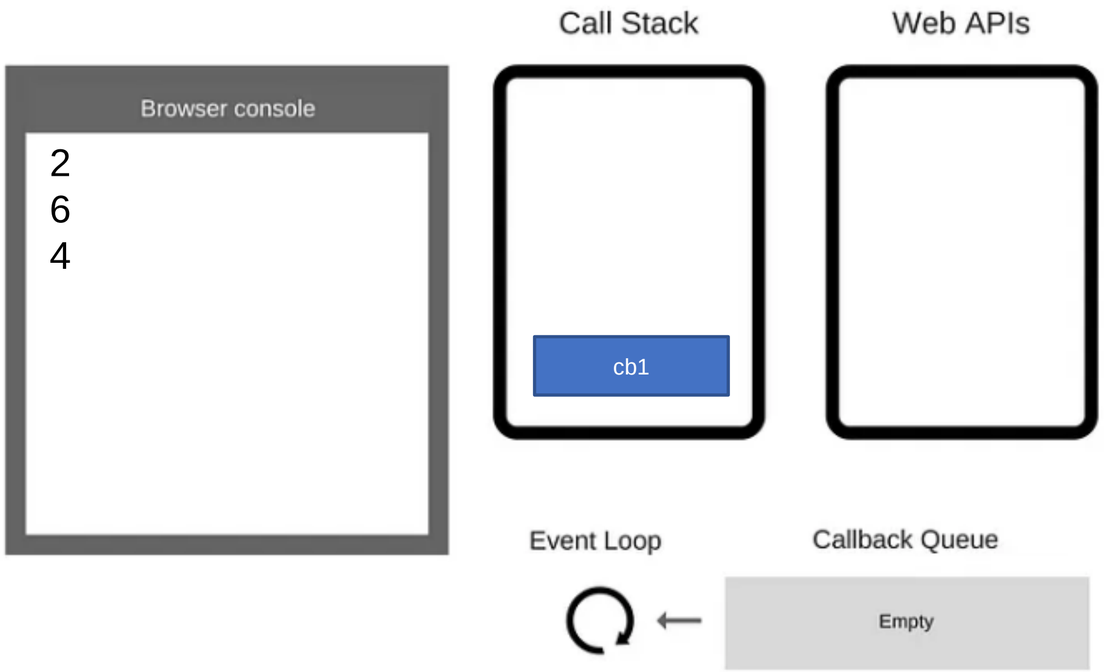
이벤트 루프




```
> console.log(1+1);
setTimeout(function cb1() { console.log(2+2) }, 1000);
console.log(3+3);
```

동기, 비동기

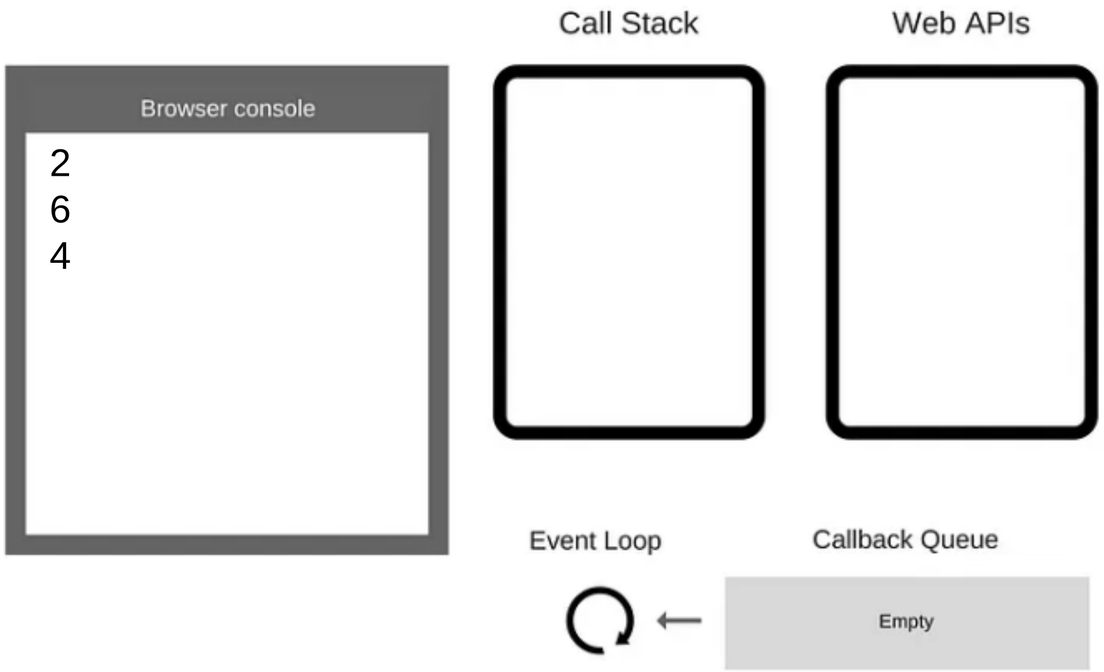
이벤트 루프




```
> console.log(1+1);
setTimeout(function cb1() { console.log(2+2) }, 1000);
console.log(3+3);
```

동기, 비동기

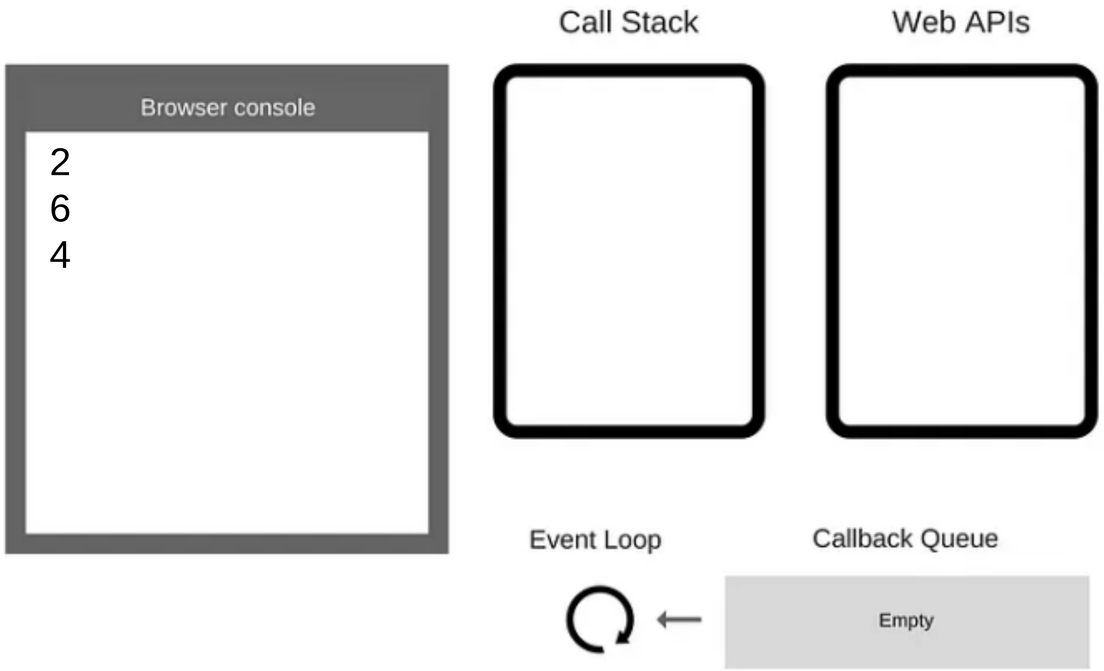
이벤트 루프




```
> console.log(1+1);
  setTimeout(function cb1() { console.log(2+2) }, 1000);
  console.log(3+3);
```

동기, 비동기

이벤트 루프




```
console.log(1+1);
setTimeout(function cb1() { console.log(2+2) }, 0);
console.log(3+3);
```

04

Promise (feat. Callback)

콜백지옥, promise 맛보기

콜백 지옥

```
function hell(win) {
 // for listener purpose
 return function() {
   loadLink(win, REMOTE_SRC+'/assets/css/style.css', function() {
     loadLink(win, REMOTE_SRC+'/lib/async.js', function() {
       loadLink(win, REMOTE_SRC+'/lib/easyXDM.js', function() {
         loadLink(win, REMOTE_SRC+'/lib/json2.js', function() {
           loadLink(win, REMOTE_SRC+'/lib/underscode.min.js', function() {
             loadLink(win, REMOTE_SRC+'/lib/backbone.min.js', function() {
               loadLink(win, REMOTE_SRC+'/dev/base_dev.js', function() {
                 loadLink(win, REMOTE_SRC+'/assets/js/deps.js', function() {
                   loadLink(win, REMOTE_SRC+'/src/' + win.loader_path + '/loader.js', function() {
                     async.eachSeries(SCRIPTS, function(src, callback) {
                       loadScript(win, BASE_URL+src, callback);
                     });
                   });
                 });
               });
             });
           });
         });
       });
     });
   });
 };
}
```

Promise

Promise는 콜백 패턴이 가진 단점을 보완한 또 다른 비동기 처리 패턴이자 객체

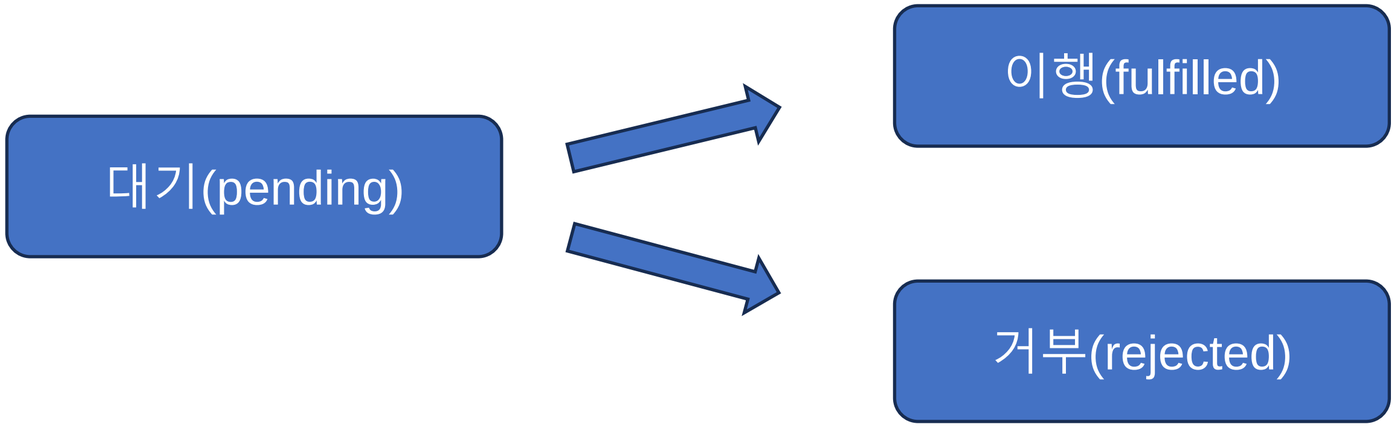
- 자바스크립트는 비동기 처리를 위한 하나의 패턴으로 콜백 함수를 사용함
- 콜백 함수는 가독성이 나쁘고 에러 처리가 번거로움

```
1  const myPromise = new Promise();
```

```
1  const myPromise = new Promise(() => {
2
3  });
```

Promise

Promise 상태



Promise

- Promise 문법

```
- new Promise((resolve, reject) => {
    if(/* 비동기 처리 성공 */) {
        resolve(…)
    } else { /* 비동기 처리 실패 */
        reject(…)
    }
  })
```

Promise

- Promise 상태
  - 프로미스의 상태는 기본적으로 pending 상태이다. 이후 비동기 처리가 수행되면 결과에 따라 상태가 변경된다.

프로미스 상태	의미	해당 상태로 변경하기 위한 작업
pending(대기)	비동기 처리가 아직 수행되지 않은 상태	없음 (프로미스가 생성된 직후 기본 상태)
fulfilled(이행)	비동기 처리가 수행된 상태(성공)	resolve 함수 호출
rejected(거부)	비동기 처리가 수행된 상태(실패)	reject 함수 호출

Promise

- 프로미스 체이닝
  - 프로미스 객체로 메소드 체이닝을 하는 것

    (메소드 체이닝: 같은 객체에
    메소드를 연속적으로 호출 하는 것)

```
myPromise
    .then((result) => {
        console.log(result);
    })
    .catch((err) => {
        console.log(err);
    })
    .finally(() => {
        console.log("finally");
    });
```

Promise

- 프로미스 체이닝
  - then, catch, finally는 언제나 프로미스를 반환하므로 연속적으로 호출할 수 있음

```
new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(1);
    }, 0000);
})
.then(result => {
  console.log(result); // 1
  return result + 10;
})
.then(result => {
  console.log(result); // 11
  return result + 20;
})
.then(result => {
  console.log(result); // 31
});
```

Promise

- then
  - 프로미스의 후속 처리 메서드
  - then을 호출하면 Promise 객체를 반환하지만 사용자가 지정한 값만 반환할 수도 있음
  - 프로미스의 상태가 변화하면 then 에 인수로 전달한 콜백 함수가 호출된다.

Promise

- then
  - new Promise((resolve, reject) => {
    …
    }).then((**첫번째 콜백 함수**, **두번째 콜백 함수**))

  - then은 두 개의 콜백 함수를 인자로 받는다
    - fulfilled 상태가 됐을 때 호출되는 콜백 함수 (resolve)
    - rejected 상태가 됐을 때 호출되는 콜백 함수 (reject)

Promise

- catch
  - 프로미스의 후속 처리 메서드
  - catch 를 호출하면 Promise 객체를 반환한다.
  - 프로미스가 rejected 상태인 경우에 호출됨
  - catch 메서드를 사용하면 then 메서드를 호출한 이후에 발생한 모든 에러를 캐치함

Promise

- catch
  - new Promise((resolve, reject) => {
    …
    }).catch((**콜백 함수**))

  - catch는 한 개의 콜백 함수를 인자로 받는다

Promise

- finally
  - 프로미스의 후속 처리 메서드
  - 비동기 작업이 resolve되거나 reject 되더라도 항상 호출됨

Promise

- finally
  - new Promise((resolve, reject) => {
    …
    }).finally((**콜백 함수**))

  - finally는 한 개의 콜백 함수를 인자로 받는다
    - 대신 콜백 함수에선 어떠한 인자도 받지 않는다

Promise

프로미스의 정적 메서드 종류

- Promise.resolve() : 주어진 값을 가지고 이행된 상태의 프로미스 객체 생성
- Promise.reject(): 주어진 값을 가지고 거부된 상태의 프로미스 객체 생성
- Promise.all(): 여러 개의 프로미스를 동시에 실행할 때 사용하는 메서드, 하나라도 거부되면 전체 프로미스가 거부됨
- Promise.allSettled(): 여러 개의 프로미스를 동시에 실행하고 모든 프로미스가 이행되거나 거부될 때까지 기다리는 메서드
- Promise.any(): 여러 개의 프로미스를 동시에 실행하고 그 중에 하나라도 이행이 되면 해당 프로미스의 값을 반환. 모든 프로미스가 거부될 때만 전체 프로미스가 거부됨
- Promise.race(): 가장 빨리 처리되는 프로미스의 값을 반환하는 메서드

참고

- 모던 자바스크립트 Deep Dive
- 짐코딩 CODING GYM
- 우아한 부트캠프
- https://medium.com/sessionstack-blog/how-javascript-works-event-loop-and-the-rise-of-async-programming-5-ways-to-better-coding-with-2f077c4438b5
- https://0taeng.tistory.com/17
- https://inpa.tistory.com/
- https://blog.toktokhan.dev/t-767eb0fa38f3
- https://www.korecmblog.com/blog/node-js-event-loop
- https://baeharam.netlify.app/posts/javascript/event-loop
- 나무위키/포켓몬스터