

JavaScript 스레딩 및 비동기

-JavaScript

싱글스레드의 특징

한번의 하나의 일만 수행

프로그래밍 난이도가 쉽다.

CPU 및 메모리 사용이 적으며 단순 CPU이용
은 멀티스레드 보다 뛰어나다.

연산량이 많을경우 효율이 떨어진다.

에러 처리가 안되면 멈춘다.

JavaScript

왜 싱글 스레드인가?

- 자바 스크립트는 홈페이지 보조를 위해 사용되었으며 싱글스레드이나 프레임워크나 환경이 멀티스레드이기 때문에 싱글스레드를 여러개 돌림

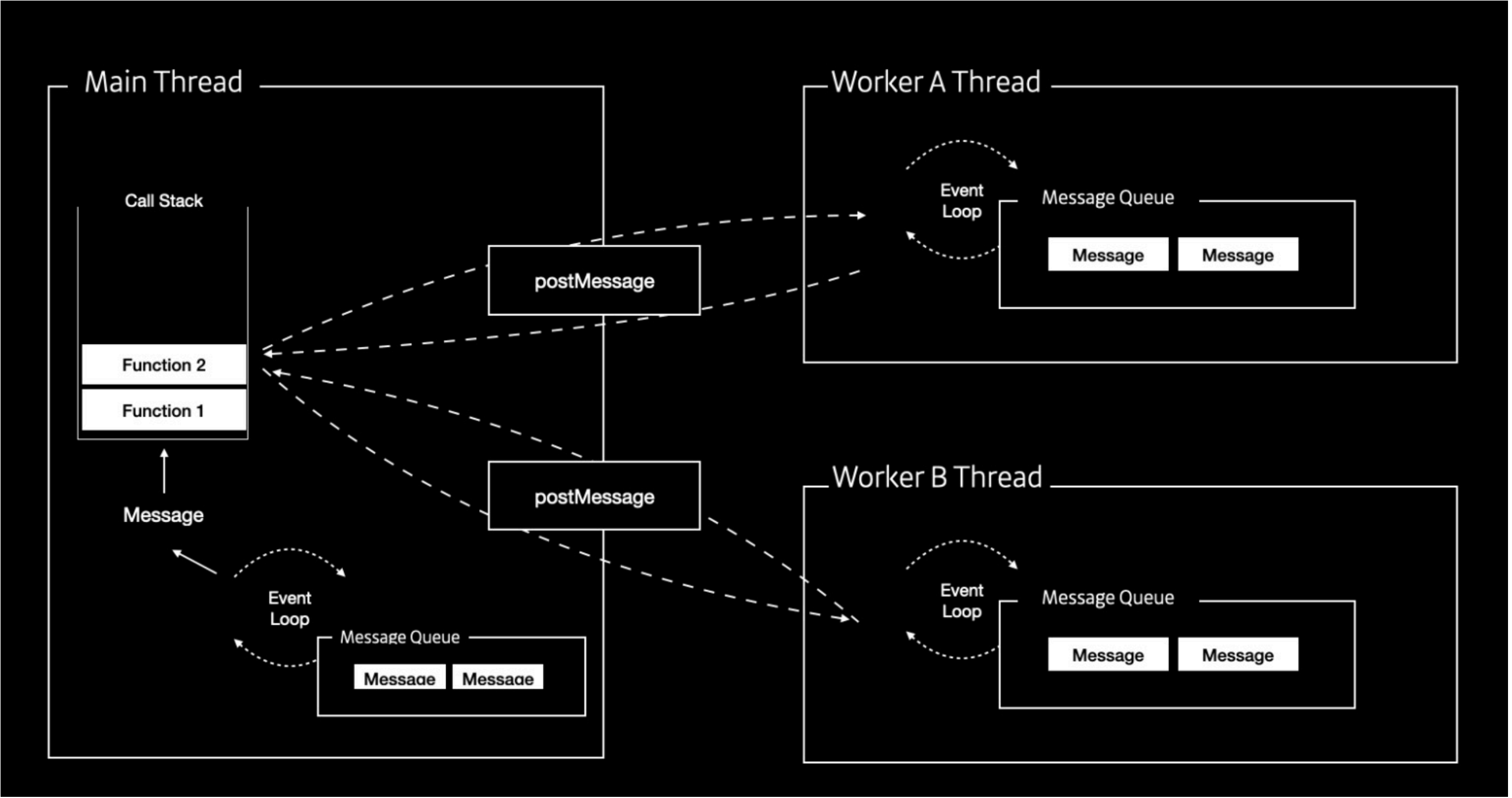
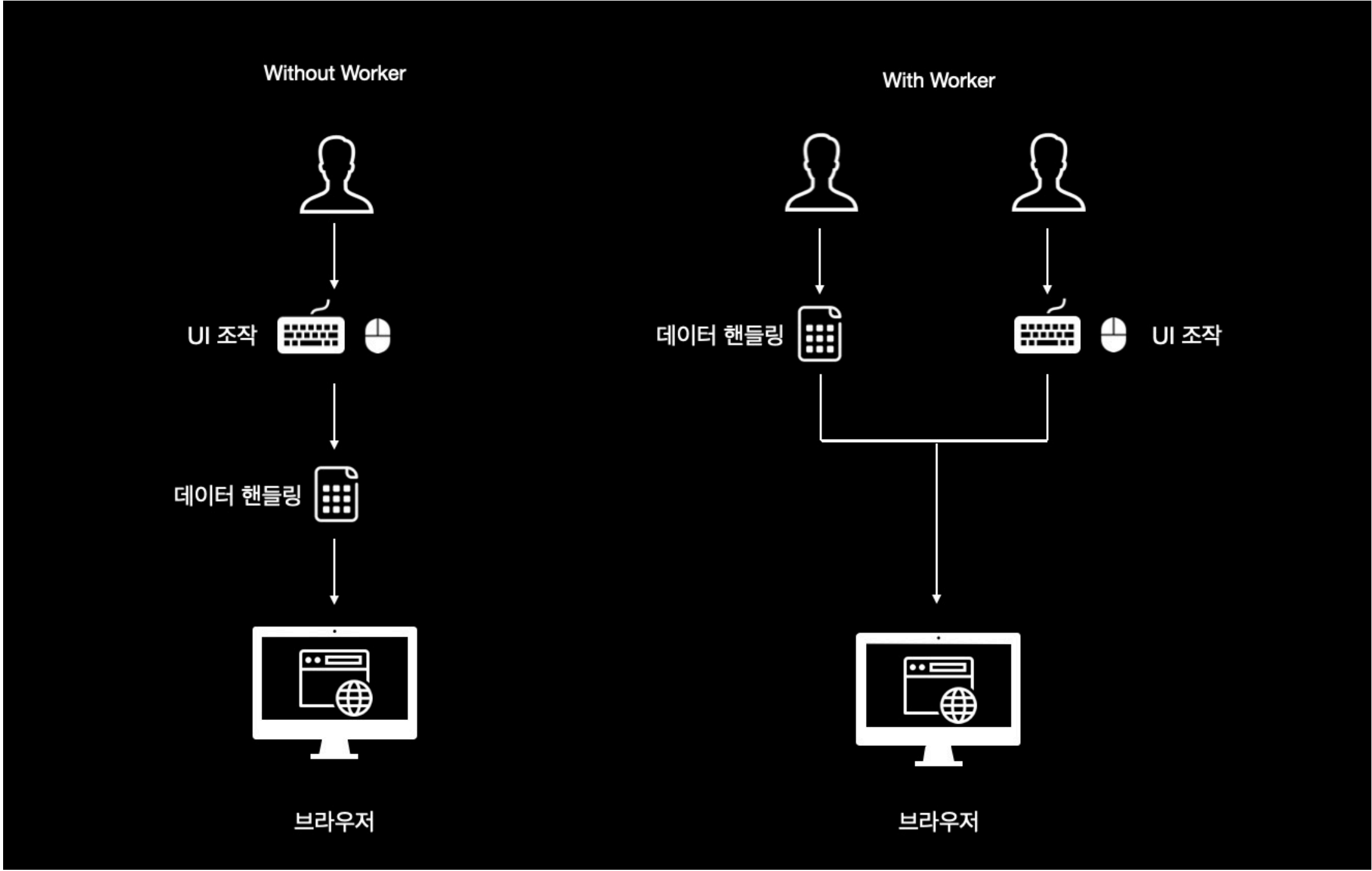
즉 스크립트 언어 답게 자바스크립트는 순차적으로 실행됨

(자바스크립트 ⇒ 싱글 스레드)

(프레임워크 ⇒ 멀티스레드)

JavaScript 멀티 스레딩

예외는 없는가?



백그라운드 처리를 위한 Web Worker 존재

JavaScript 멀티 스레드

왜 필요한가?

```
// main.js
const worker = new Worker("worker.js");

worker.postMessage("데이터 전송");

worker.onmessage = function(event) {
  console.log("Worker로부터의 응답:", event.data);
};

// worker.js
self.onmessage = function(event) {
  // 복잡한 연산 수행
  let result = "연산 결과";
  self.postMessage(result);
};
```

사용목적

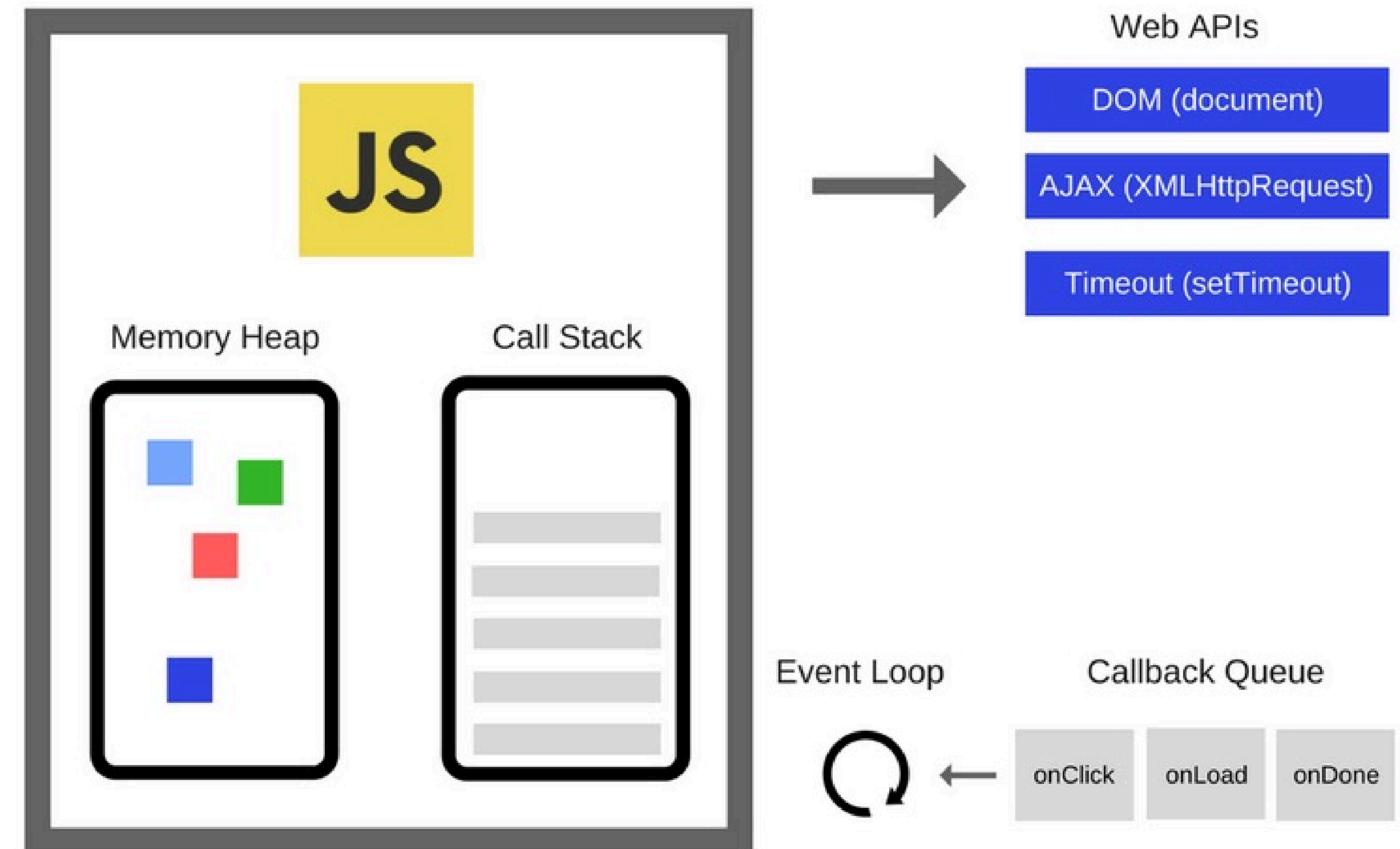
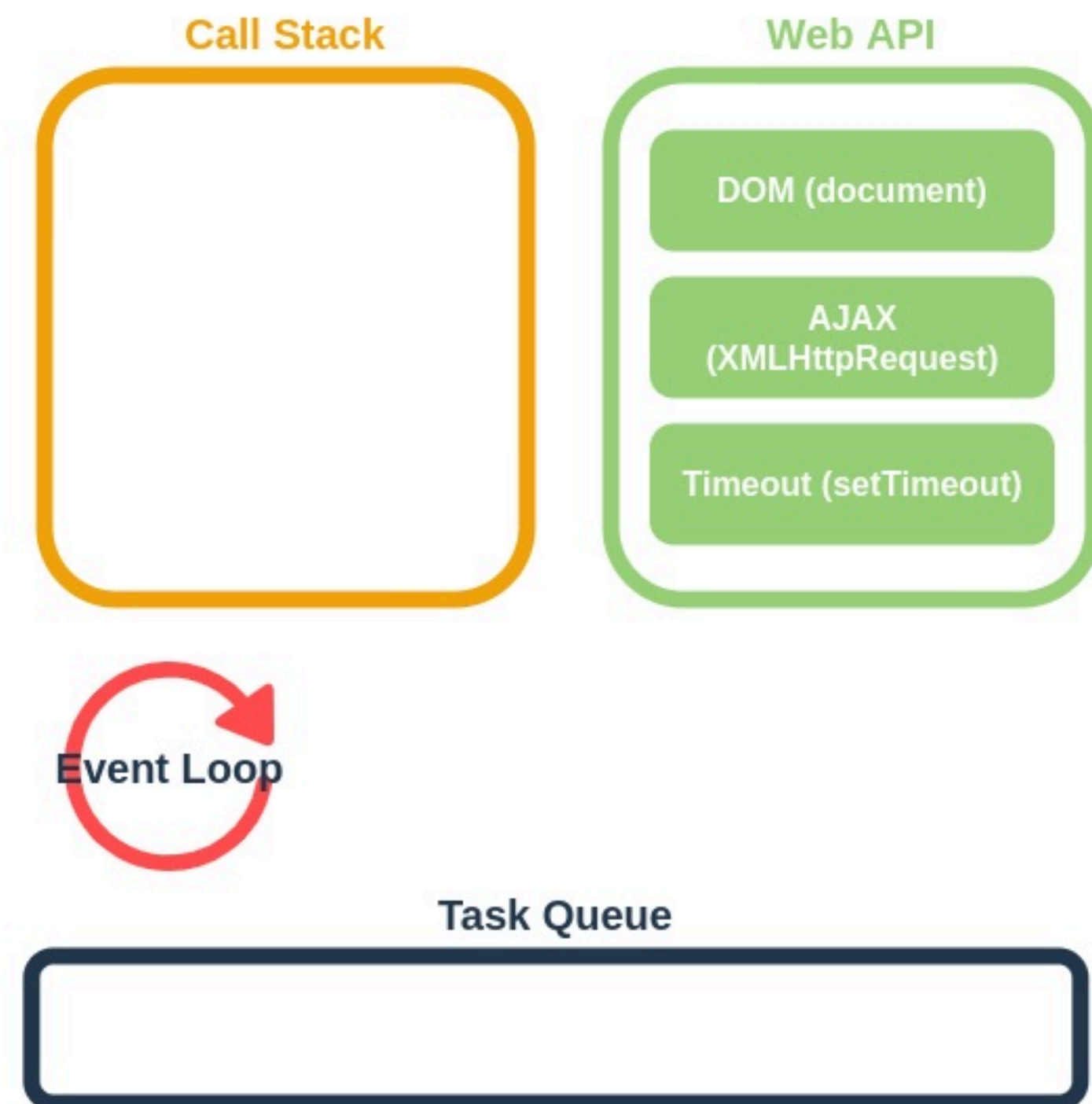
1. 복잡한 계산이 필요한 경우에 유용
2. 메인 스레드에 영향을 주지 않고 작업하는 경우

사용하지않는경우

1. 큰 작업은 대부분 서버에서 처리
2. DOM이나 UI등 제한이 제한적
3. 단순한 계산은 스레드간 통신비용이 더 큼

- callback : 특정함수 완료시 실행되는 함수

JavaScript 비동기 구조



Call Stack : 자바스크립트에서 수행해야 할 함수들을 순차적으로 스택에 담아 처리
Web API : 웹 브라우저에서 제공하는 API로 AJAX나 Timeout등의 비동기 작업을 실행
Task Queue : Callback Queue 라고도 하며 Web API에서 넘겨받은 Callback 함수를 저장
Event Loop : Call Stack이 비어있다면 Task Queue의 작업을 Call Stack으로 옮김

즉 실행함수 저장(초기) → web api 실행 → callback을 Task Queue에 저장
 Event loop에서 call stack이 비었을때 Task Queue의 callback을 call stack

JavaScript 비동기 함수의 종류들 (1)

1. `setTimeout` 과 `setInterval`

- `setTimeout` 과 `setInterval` 은 특정 시간이 지나면 코드가 실행되도록 예약하는 방법입니다.
- `setTimeout`: 일정 시간이 지난 후에 한 번만 실행됩니다.
- `setInterval`: 일정 시간 간격으로 반복해서 실행됩니다.

javascript

Copy code

```
setTimeout(() => console.log("1초 후 실행"), 1000);
setInterval(() => console.log("1초마다 실행"), 1000);
```

2. `Promise`

- `Promise` 는 비동기 작업이 완료된 이후에 결과값을 제공하는 객체입니다. `then`, `catch`, `finally` 메서드를 사용해 처리할 수 있으며, 연속적인 비동기 처리를 할 때 유용합니다.

javascript

Copy code

```
new Promise((resolve, reject) => {
  setTimeout(() => resolve("완료!"), 1000);
})
.then(result => console.log(result))
.catch(error => console.error(error));
```

3. `async` / `await`

- `async` / `await` 는 `Promise` 기반 비동기 코드를 더욱 간결하게 작성할 수 있게 해 줍니다. `await` 키워드는 비동기 작업이 완료될 때까지 기다리며, 해당 작업이 완료되면 다음 코드를 실행합니다.

javascript

Copy code

```
async function fetchData() {
  const data = await fetch('https://api.example.com/data');
  return await data.json();
}
```

4. AJAX (`XMLHttpRequest`) 및 `fetch` API

- 비동기적으로 서버에 HTTP 요청을 보내기 위해 AJAX와 `fetch` API를 사용할 수 있습니다. 이 방법으로 데이터를 비동기적으로 가져와 화면에 렌더링할 수 있습니다.
- **AJAX**: 과거에는 `XMLHttpRequest` 객체를 사용해 비동기 요청을 보냈지만, 현대에서는 `fetch` 가 더 많이 사용됩니다.

javascript

Copy code

```
fetch('https://api.example.com/data')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error(error));
```

JavaScript 비동기 함수의 종류들 (2)

5. 이벤트 리스너

- `addEventListener` 메서드를 통해 특정 이벤트가 발생했을 때 비동기적으로 코드를 실행할 수 있습니다. 예를 들어 클릭 이벤트나 네트워크 상태 변화 등을 감지해 이벤트가 발생할 때마다 처리를 할 수 있습니다.

javascript

Copy code

```
document.getElementById("button").addEventListener("click", () => {
  console.log("버튼 클릭!");
});
```

6. Web Worker

- Web Worker를 사용하면 메인 스레드와 별개로 백그라운드에서 작업을 수행할 수 있어 UI가 멈추지 않도록 할 수 있습니다.

javascript

Copy code

```
const worker = new Worker("worker.js");
worker.postMessage("작업 시작");
worker.onmessage = (event) => {
  console.log("작업 결과:", event.data);
};
```

7. 요청 애니메이션 프레임 (requestAnimationFrame)

- `requestAnimationFrame`은 브라우저의 다음 리페인트 시점에 함수를 실행하도록 예약하는 함수입니다. 주로 애니메이션을 구현할 때 프레임 단위로 코드를 실행하는 데 사용됩니다.

javascript

Copy code

```
function animate() {
  // 애니메이션 로직
  requestAnimationFrame(animate);
}
animate();
```

8. WebSocket

- WebSocket은 서버와 클라이언트 간의 실시간 양방향 통신을 지원하는 프로토콜로, 데이터를 실시간으로 비동기적으로 주고받을 수 있습니다.

javascript

Copy code

```
const socket = new WebSocket('ws://example.com');
socket.onmessage = (event) => {
  console.log("서버로부터 메시지:", event.data);
};
```


JavaScript 비동기 함수의 종류들 (3) - 번외

9. IndexedDB와 Service Worker

- **IndexedDB**: 대용량 데이터를 비동기적으로 브라우저에 저장하고 관리할 수 있습니다.
- **Service Worker**: 웹 애플리케이션의 백그라운드에서 캐싱이나 푸시 알림 등을 비동기적으로 처리할 수 있는 기능입니다.

10. 기타 API

- **Notification API**: 알림을 비동기적으로 보내고 이벤트를 처리할 수 있습니다.
- **File API**: 파일을 읽고 쓰는 작업을 비동기적으로 처리합니다.
- **Geolocation API**: 위치 정보를 비동기적으로 요청할 수 있습니다.

Promised 보강설명

1. Promise 객체 생성

Promise 객체는 `new Promise()` 구문으로 생성되며, 비동기 작업을 수행할 함수를 인수로 받습니다. 이 함수는 `resolve` 와 `reject` 두 개의 인수를 받고, 작업이 성공하면 `resolve`, 실패하면 `reject` 를 호출하여 완료 상태를 전달합니다.

```
javascript
Copy code

const myPromise = new Promise((resolve, reject) => {
  // 비동기 작업 수행
  if (작업성공) {
    resolve("작업 성공 결과");
  } else {
    reject("작업 실패 결과");
  }
});
```

2. then, catch, finally 메서드

Promise 객체에서 비동기 작업이 완료된 후의 결과를 처리하기 위한 메서드들입니다.

- `then(onFulfilled, onRejected)`

작업이 성공했을 때 `onFulfilled` 콜백이 실행되고, 실패했을 때 `onRejected` 콜백이 실행됩니다.

```
javascript
Copy code

myPromise.then(result => {
  console.log("성공:", result);
}).catch(error => {
  console.log("실패:", error);
});
```

- `catch(onRejected)`

작업이 실패했을 때 `onRejected` 콜백이 실행됩니다. `then` 에 오류 핸들러를 붙이는 대신, `catch` 로 처리할 수 있습니다.

```
javascript
Copy code

myPromise.catch(error => {
  console.log("실패:", error);
});
```

- `finally(onFinally)`

작업의 성공 여부와 관계없이 항상 실행되는 콜백입니다. 리소스를 정리하거나 로딩 상태를 종료할 때 유용합니다.

```
javascript
Copy code


myPromise.finally(() => {
  console.log("작업 완료");
});
```


Promised 보강설명

- `Promise.resolve(value)`

이미 완료된 `Promise` 를 즉시 반환합니다. 인수로 전달된 `value` 를 사용해 `resolve` 된 `Promise` 를 반환합니다.

javascript

 Copy code

```
Promise.resolve("완료된 값").then(console.log); // 출력: 완료된 값
```

- `Promise.reject(reason)`

실패한 `Promise` 를 즉시 반환합니다. 인수로 전달된 `reason` 을 사용해 `reject` 된 `Promise` 를 반환합니다.

javascript


 Copy code

```
Promise.reject("에러 발생").catch(console.error); // 출력: 에러 발생
```

- `Promise.all(iterable)`

여러 `Promise` 가 모두 완료될 때까지 기다렸다가 결과를 배열로 반환합니다. 하나라도 실패하면 전체가 실패로 처리됩니다.

javascript

 Copy code

```
Promise.all([promise1, promise2, promise3])
  .then(results => console.log(results))
  .catch(error => console.error("하나 이상의 작업 실패:", error));
```

- `Promise.allSettled(iterable)`

모든 `Promise` 가 완료될 때까지 기다렸다가, 각 작업의 성공/실패 여부와 결과를 포함한 객체 배열을 반환합니다.

javascript


 Copy code

```
Promise.allSettled([promise1, promise2])
  .then(results => console.log(results));
```

- `Promise.race(iterable)`

가장 먼저 완료되는 `Promise` 의 결과 또는 오류를 반환합니다.

javascript


 Copy code

```
Promise.race([promise1, promise2])
  .then(result => console.log("가장 빠른 결과:", result))
  .catch(error => console.error("가장 빠른 실패:", error));
```

- `Promise.any(iterable)`

가장 먼저 성공하는 `Promise` 의 결과를 반환합니다. 모든 `Promise` 가 실패할 경우에만 `AggregateError` 를 반환합니다.

javascript

 Copy code

```
Promise.any([promise1, promise2, promise3])
  .then(result => console.log("첫 번째 성공:", result))
  .catch(error => console.error("모든 작업 실패:", error));
```

참고

devmag coding

ChatGPT