자바스크립트는 기본적으로 "동기 처리(synchronous)" 언어

기본 동작은 "동기" 순서대로 실행된다는 이야기이다.

자바스크립트는 한 줄씩 순서대로 실행하는 **싱글 스레드(single-thread)** 언어여서 다음과 같이 실행된다.

```
console.log("1");
console.log("2");
console.log("3");
```

결과는 항상순 순서대로 **동기 처리된 다음과 같은 결과이다.**

```
1
2
3
```

웹 프로그래밍에서 비동기가 필요할 때가 있다.

순서대로 진행하지 않는 예를 들어 보면

```
const data = fetchDataFromServer(); // 서버에서 데이터 가져오기
console.log(data); // 데이터가 오기 전까지 기다리면 전체가 멈춤
```

fetchDataFromServer() 불안한 서버 상태에서 문제가 발생하면 프로그램이 멈춰버림

이런 상황을 **동기 처리**로 하면 → 전체 앱이 **멈춘 것처럼 보이는 문제**발생
그래서 자바스크립트는 비동기 처리 방식(콜백, Promise, async/await)를 지원해서 이런 문제를 해결한다.

정리하면?

구분	설명
기본	자바스크립트는 원래 **동기적**으로 작동함
실무	네트워크 요청, 타이머 등은 **비동기 방식**으로 처리 필요
도구	콜백, Promise, async/await 등으로 비동기 구현 가능

자바스크립트에서 **비동기 처리**는 시간이 오래 걸리는 작업(예: 서버 요청, 파일 읽기 등)을 기다리지 않고 다른 작업을 먼저 수행할 수 있도록 도와주는 방식 크게 다음 3가지 방식이 있다.

1. 콜백 함수 (Callback)

가장 기본적인 비동기 처리 방식

아래 코드를 보면 모두 **동기식(synchronous)**으로 작성되어 있어, 코드가 위에서 아래로 **순차적으로 실행** 됩니다

```
function fetchDataSync(callback) {
  // 실제 네트워크 요청처럼 1초 대기하는 건 동기적으로 불가능하지만,
  // 여기서는 그냥 데이터 가져왔다고 가정
  console.log("데이터 가져오기 완료");
  callback();
}

//다음 코드들은 동기적으로 실행된다.
fetchDataSync(() => {
  console.log("콜백 실행됨");
});
console.log("프로그램 완료");
```

실행 결과:

데이터 가져오기 완료
콜백 실행됨
프로그램 완료

실행 순서 설명:

1. fetchDataSync() 호출됨.
2. console.log("데이터 가져오기 완료") 실행.
3. callback() 실행 → console.log("콜백 실행됨").
4. 마지막으로 console.log("프로그램 완료") 실행.

따라서 위 출력은 **정확히 그 순서대로** 화면에 표시됩니다.

이 코드는 **비동기 처리**를 사용하고 있어서, 실행 흐름에 따라 결과가 약간 예상과 다를 수 있습니다.

```
function fetchData(callback) {
  setTimeout(() => {
    console.log("데이터 가져오기 완료");
    callback(); // 작업이 끝난 뒤 실행할 함수
  }, 1000);
}

fetchData(() => {
  //이부분의 코드는 비동기 식으로 실행된다.
  console.log("콜백 실행됨");
});

console.log("프로그램 완료");
```

실행 결과:

프로그램 완료
(1초 후)
데이터 가져오기 완료
콜백 실행됨

실행 흐름 설명:

1. fetchData() 호출 → 내부적으로 setTimeout 등록 (1초 후 실행 예정)
2. 바로 다음 줄인 console.log("프로그램 완료")가 즉시 실행됨
3. 1초 뒤, setTimeout 안의 코드 실행:
    - console.log("데이터 가져오기 완료")
    - callback() 호출 → console.log("콜백 실행됨")

즉, **console.log("프로그램 완료")는 비동기 작업보다 먼저 실행**됩니다.
그래서 결과적으로 프로그램 완료가 제일 먼저 출력되고, **1초 후** 나머지 메시지가 출력됩니다.

2. Promise

JavaScript에서 Promise는 비동기 작업의 완료 또는 실패를 나타내는 객체입니다. 이를 통해 비동기 코드를 더 명확하고 체계적으로 작성할 수 있습니다.

다음은 Promise의 주요 문법에 대한 자세한 설명이다.

1. new Promise((resolve, reject) => { ... })

Promise 객체는 new Promise() 생성자를 사용하여 생성됩니다. 이 생성자는 두 개의 콜백 함수 resolve와 reject를 인자로 받는 실행 함수(executor)를 필요로 합니다.

Promise의 생성자 매개변수는 함수를 받고 그함수의 매개변수 resuolve,reject 도 함수다.

```
const promise = new Promise((resolve, reject) => {
  // Promise 생성자에 함수 이부분은 동기적으로 실행 되지만
  // 그안에서 비동기 작업 수행을 진행해서 성공 여부에 따라 resolve나 reject 메소드를 호출한다.
```

```
// 이곳에 비동기 작업 코드 기술
  if (/* 작업 성공 */) {
    resolve('성공 결과');
  } else {
    reject('에러 메시지');
  }
});
```

- `resolve(value)`: 비동기 작업이 성공했을 때 호출하며, `value`는 이후 `.then()`에서 접근할 수 있습니다.
- `reject(reason)`: 비동기 작업이 실패했을 때 호출하며, `reason`은 이후 `.catch()`에서 접근할 수 있습니다.

이 구조는 비동기 작업의 결과를 명확하게 처리할 수 있도록 도와줍니다.

2. .then(onFulfilled, onRejected)

`Promise` 객체는 `.then()` 메서드를 사용하여 비동기 작업의 성공과 실패에 대한 처리를 정의할 수 있습니다.

```
promise.then(
  (value) => {
    // 작업 성공 시 실행할 작업 이곳에 기술
    console.log(value);
  },
  (error) => {
    // 작업 실패 시 실행할 작업 여기서 기술
    console.error(error);
  }
);
```

- `onFulfilled`: `resolve`가 호출되었을 때 실행되는 콜백 함수입니다.
- `onRejected`: `reject`가 호출되었을 때 실행되는 콜백 함수입니다.

또한, `.then()`은 새로운 `Promise`를 반환하므로, 체이닝을 통해 연속적인 비동기 작업을 처리할 수 있습니다.

3. .catch(onRejected)

`.catch()` 메서드는 `.then()`의 두 번째 인자와 동일한 역할을 하며, `Promise` 체인에서 발생한 에러를 처리하는 데 사용됩니다.

```
promise
  .then((value) => {
    // 작업 성공 시 실행
    console.log(value);
  })
  .catch((error) => {
    // 작업 실패 시 실행
    console.error(error);
  });
```

이 구조는 에러 처리를 한 곳에서 집중적으로 관리할 수 있게 해줍니다.

Promise 체이닝의 핵심 원리랑 연결되는 내용이야.
정확하게 말하면 이렇게 이해하면 돼:

then에서 리턴한 값은 자동으로 Promise로 감싸져서 다음 then으로 넘어감

```
return "다음 작업";
```

요렇게 **문자열**을 리턴했지만, 자바스크립트는 내부적으로:

```
return Promise.resolve("다음 작업");
```

처럼 **자동으로 Promise로 감싸서** 다음 then으로 넘겨줘.

🔄 흐름을 예제로 보면

```
fetchData()
  .then(result => {
    console.log(result);          // fetchData()의 결과 출력
    return "다음 작업";           // 일반 문자열 리턴
  })
  .then(msg => {
    console.log(msg);             // "다음 작업" 출력됨
  });
```

이 경우 "다음 작업"은 자동으로 Promise.resolve("다음 작업")처럼 되기 때문에,
다음 then의 콜백에서 msg로 바로 받아서 출력할 수 있어.

✅ 정리하면

리턴값 유형	다음 `.then(...)`으로 전달되는 값
일반 값 (`"문자열"`, 숫자, 객체)	`Promise.resolve(값)`으로 감싸짐
`Promise` 객체	그대로 다음 `then`이 처리함
아무것도 없음 (`undefined`)	다음 `then`은 `undefined` 받음

이 예제는 **`Promise` 생성자 안의 코드는 동기,**
그 뒤에 이어지는 `.then()`은 **비동기(마이크로태스크)** 로 실행돼.

🔍 코드 설명:

js
복사편집

```
const promise = new Promise((resolve, reject) => {
  console.log("나는 즉시 실행된다!");
  resolve("완료");
});

promise.then(result => {
  console.log("then 실행:", result);
});

console.log("끝");
```

✅ 실행 순서:

1. `new Promise(...)` 호출됨 → 안에 있는 함수 **즉시 실행 (동기)**
   👉 `console.log("나는 즉시 실행된다!");` 실행됨

2. `resolve("완료")` → 성공 상태로 바뀜, **then 예약**

3. `console.log("끝");` → **동기 코드니까 바로 실행**

4. 마이크로태스크 큐에 쌓여 있던 `then(...)` 실행

📜 실제 출력 결과:

복사편집

```
나는 즉시 실행된다!
끝
then 실행: 완료
```

💡 핵심 정리:

코드 위치	실행 시점
Promise 생성자 내부 코드	🕛 즉시 실행 (동기)
`.then(...)`, `.catch(...)`	⏳ 나중에 실행 (비동기, 마이크로태스크)

이 예제는 **Promise 생성자 안의 코드는 동기,**
그 뒤에 이어지는 `.then()`은 **비동기(마이크로태스크)** 로 실행돼.

🔍 코드 설명:

js
복사편집

```
const promise = new Promise((resolve, reject) => {
  console.log("나는 즉시 실행된다!");
  resolve("완료");
});

promise.then(result => {
  console.log("then 실행:", result);
});

console.log("끝");
```

✅ 실행 순서:

1. `new Promise(...)` 호출됨 → 안에 있는 함수 **즉시 실행 (동기)**
   👉 `console.log("나는 즉시 실행된다!");` 실행됨

2. `resolve("완료")` → 성공 상태로 바뀜, **then 예약**

3. `console.log("끝");` → **동기 코드니까 바로 실행**

4. 마이크로태스크 큐에 쌓여 있던 `then(...)` 실행

📜 실제 출력 결과:

복사편집

나는 즉시 실행된다!
끝
then 실행: 완료

💡 핵심 정리:

코드 위치	실행 시점
Promise 생성자 내부 코드	⏱️ 즉시 실행 (동기)
.then(...), .catch(...)	⏳ 나중에 실행 (비동기, 마이크로태스크)

4. .finally(onFinally)

.finally() 메서드는 Promise가 이행되든 거부되든 상관없이 항상 실행되는 콜백 함수를 정의합니다.

```
promise
  .then((value) => {
    console.log(value);
  })
  .catch((error) => {
    console.error(error);
  })
  .finally(() => {
    // 항상 실행
    console.log('작업 완료');
  });
```

이 메서드는 정리 작업이나 로딩 상태 해제 등 공통적으로 수행해야 하는 작업을 처리하는 데 유용합니다.

---

5. Promise의 상태

Promise는 다음 세 가지 상태를 가질 수 있습니다:

- pending: 초기 상태로, 이행되거나 거부되지 않은 상태입니다.
- fulfilled: resolve가 호출되어 작업이 성공적으로 완료된 상태입니다.
- rejected: reject가 호출되어 작업이 실패한 상태입니다.

이러한 상태 관리를 통해 비동기 작업의 흐름을 명확하게 파악하고 제어할 수 있습니다.

이러한 Promise 문법을 활용하면 복잡한 비동기 작업을 더 구조적이고 가독성 있게 관리할 수 있습니다.

```
function fetchData() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve("데이터 가져오기 완료");
    }, 1000);
  });
}

fetchData()
  .then(result => {
    console.log(result);
    return "다음 작업";
  })
```

```
.then(msg => {
  console.log(msg);
})
.catch(error => {
  console.error(error);
});
```

🔍 실행 순서

1. `fetchData()` 호출

2. 1초 뒤에 `resolve("데이터 가져오기 완료")` 실행

3. 첫 번째 `.then()`의 `result`는 `"데이터 가져오기 완료"` → 이 값을 출력하고
   → `"다음 작업"`을 다음 then으로 전달

4. 두 번째 `.then()`의 `msg`는 "다음 작업"
5. 에러가 나면 catch로 전달

✅ 예제: 데이터를 1초 후 받아오는 함수

javascript
복사편집

```
function fetchData() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve("📦 데이터 도착!");
    }, 1000);
  });
}
```

✅ 이걸 async/await로 쓰면?

javascript
복사편집

```
async function getData() {
  console.log("1. 데이터 요청 중...");

  const result = await fetchData(); // fetchData가 끝날 때까지 기다림

  console.log("2.", result);        // "📦 데이터 도착!" 출력
  console.log("3. 다음 작업 실행");
}

getData();
```

🔍 출력 결과 (1초 후):

markdown

복사편집

1. 데이터 요청 중...
2. 📦 데이터 도착!
3. 다음 작업 실행

🧠 핵심 포인트

- `async` 함수 안에서만 `await` 사용 가능
- `await`는 Promise가 처리될 때까지 기다림
- 그래서 코드가 동기적으로 보이지만, 실제론 비동기

지금 주신 `fetchData()` 함수를 똑같이 쓰면서,
`async/await` 방식으로 바꾼 예제를 아래처럼 만들 수 있어요:

✅ async/await로 변환한 예제

javascript
복사편집

```
function fetchData() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const success = Math.random() > 0.5;

      if (success) {
        resolve("데이터 가져오기 완료");
      } else {
        reject("❌ 데이터 가져오기 실패");
      }
    }, 1000);
  });
}

async function run() {
  try {
    console.log("🚀 데이터 요청 중...");

    const result = await fetchData(); // 성공하면 resolve값을 받음
    console.log("✅", result);

    const next = "다음 작업";
    console.log("➡️", next);

  } catch (error) {
```

```
    console.error("🚨", error); // 실패하면 reject값을 catch에서 처리
  }
}

run();
```

🧐 실행 흐름

- `await fetchData()`
  → `resolve()`면 `result`에 값이 들어오고 다음 코드 실행
  → `reject()`면 바로 `catch`로 이동

🔁 실행 결과 (확률에 따라 다름):

성공 시:

복사편집
🚀 데이터 요청 중...
✅ 데이터 가져오기 완료
➡️ 다음 작업

실패 시:

복사편집
🚀 데이터 요청 중...
🚨 ❌ 데이터 가져오기 실패

정리된 설명:

```
// 1. Waiting function (returns a message after waiting)
const wait = (seconds) => new Promise(resolve =>
  setTimeout(() => resolve(`Done waiting ${seconds} seconds! ⏱️`), seconds
* 1000)
);

// 2. Bathroom function
async function goToBathroom() {
  console.log("Going to the bathroom~ 🚽");

  const result = await wait(3);  // ✅ 3초 기다리고 결과 받기
  console.log(result);           // => Done waiting 3 seconds! ⏱️

  console.log("I'm here! 😊");
}

// 3. Execution part
goToBathroom();
console.log("Program ends");
```

🟢 실행 결과:

```
vbnet
복사
편집
Going to the bathroom~ 🚽
Program ends
(3초 후)
Done waiting 3 seconds! ⏱️
I'm here! 😊
```

```
// 1. Waiting function (returns a message after waiting)
const wait = (seconds) => new Promise(resolve =>
  setTimeout(() => resolve(`Done waiting ${seconds} seconds! ⏱️`), seconds
* 1000)
);

// 2. Bathroom function
async function goToBathroom() {
  console.log("Going to the bathroom~ 🚽");

  const result = wait(3);  // ❌ await 없이 호출
  console.log(result);     // ❌ 이 시점에서 Promise 객체가 출력됨

  console.log("I'm here! 😊");
}

// 3. Execution part
goToBathroom();
console.log("Program ends");
```

🔴 실행 결과:

```
Going to the bathroom~ 🚽
Promise { <pending> }     // 아직 끝나지 않은 Promise가 출력됨
I'm here! 😊
Program ends
(3초 후)
```

✅ 참고: 3초 후 "Done waiting 3 seconds! ⏱️" 메시지는 출력되지 않음.
왜냐하면 await로 결과를 기다리지 않고, result를 콘솔에 찍는 시점에 Promise는 아직 실행 중이기 때문이에요.