

## 3.1 비동기와 타이머 ~

## 3.4 프로미스와 **ASYNC/AWAIT**

## 3.1 비동기와 타이머

**동기** → 앞선 작업이 완전히 끝난 후에 다음 작업이 실행되는 것

**비동기** → 앞선 작업이 끝나지 않았는데도 다음 작업이 실행되는 것

# 비동기

**작성한 코드 순서와 다르게 실행되는 코드**

대표적인 예 : ) 타이머 함수

지정한 시간(밀리초) 뒤에 지정한 작업을 수행하는 함수

## SECTION 01

# setTimeout()

## setTimeout()

- **setTimeout()** : 지정한 시간 뒤에 코드가 실행

→ 첫 번째 인수 넣은 함수가 지정한 밀리초 후에 실행

형식

`setTimeout(함수, 밀리초);`

### 첫 번째 인수 : 함수

특정 작업(지정한 시간까지 기다리기) 이후에 추가로 실행되므로 **콜백 함수**로 볼 수 있음.

### 두 번째 인수 : 밀리초

밀리초 단위 이므로 초에 1000을 곱해 넣기.

## setTimeout()

- setTimeout() 예시

EX:) 2초 뒤에 메시지 표시

```
setTimeout(() => {  
  console.log('2초 후에 실행됩니다. ');  
}, 2000);
```

< 3

2초 후에 실행됩니다.

메시지가 출력되기 전에 나오는 숫자  
: setTimeout() 함수에서 반환하는 **타이머 아이디**

## setTimeout()

- **setTimeout()** 예시

**EX:)** 첫 번째 인수에 넣는 함수를 외부에서 가져오기

```
const callback = () => {  
  console.log('2초 후에 실행됩니다.');
```

```
};  
  
setTimeout(callback, 2000);
```



`setTimeout()`

**SETTIMEOUT() 함수를 비동기라고 하는 이유는 ?**

**setTimeout() 뒤에 나오는 코드가 더 먼저 실행되기 때문**

## setTimeout()

- **setTimeout() 예시**

**EX:)** 코드 작성 순서와 실제 실행 순서 차이

```
console.log(1);  
setTimeout(() => {  
  console.log(2);  
}, 2000);  
console.log(3);  
// 1, 3, 2
```

→ **1, 3, (2초 기다리고) 2가 찍힘.**

## setTimeout()

- setTimeout() 예시

**EX:)** 여러 개 setTimeout() 함수와 동기 함수

```
setTimeout(() => {  
  console.log(3);  
}, 5000);  
setTimeout(() => {  
  console.log(2);  
}, 3000);  
setTimeout(() => {  
  console.log(1);  
}, 2000);  
console.log('내가 먼저');  
// 내가 먼저, 1, 2, 3
```

**setTimeout() 에 넣은 함수는  
동기 코드가 실행되고 난 뒤 실행됨.**

## setTimeout()

- **setTimeout() 예시**

**EX:)** setTimeout() 함수 여러번 실행시키기

```
const callback = () => {  
  console.log('2초마다 실행됩니다. ');  
  setTimeout(callback, 2000);  
}  
setTimeout(callback, 2000);
```

### **setTimeout() 콜백함수에서 다시 setTimeout() 함수 호출하기**

setTimeout() 내부의 callback()에서  
다시 setTimeout()을 호출하므로 2초마다  
setTimeout()이 반복해서 실행

## SECTION 02

# setInterval()

## setInterval()

- **setInterval()** : 지정한 시간마다 주기적으로 지정한 함수를 실행

→ 자바스크립트가 제공하는 자체적으로 반복 기능을 수행하는 함수

형식    `setInterval(함수, 밀리초);`

```
setInterval(() => {  
  console.log('2초마다 실행됩니다.');
```

2초마다 한 번씩 메시지가 출력

## SECTION 03

# clearTimeout()과 clearInterval()

`clearTimeout()`과 `clearInterval()`

## `setTimeout()`과 `setInterval()`

**문제점** → 웹 페이지를 닫을 때 까지 실행  
중간에 끄는 방법 필요

**해결 방안** → 타이머 정리 함수를 이용



**타이머 정리 함수로 실행 취소**

**setTimeout() → clearTimeout()**

**setInterval() → clearInterval()**

### 타이머 함수를 사용해 실행 취소 방법

setTimeout()과 setInterval() 함수는 해당 타이머를 나타내는 타이머 아이디를 반환  
→ 이 값을 clearTimeout()과 clearInterval() 함수에 넣으면 취소할 타이머를 지정할 수 있음.

```
형식  const 아이디 = setTimeout(함수, 밀리초);  
      clearTimeout(아이디);  
      const 아이디 = setInterval(함수, 밀리초);  
      clearInterval(아이디);
```

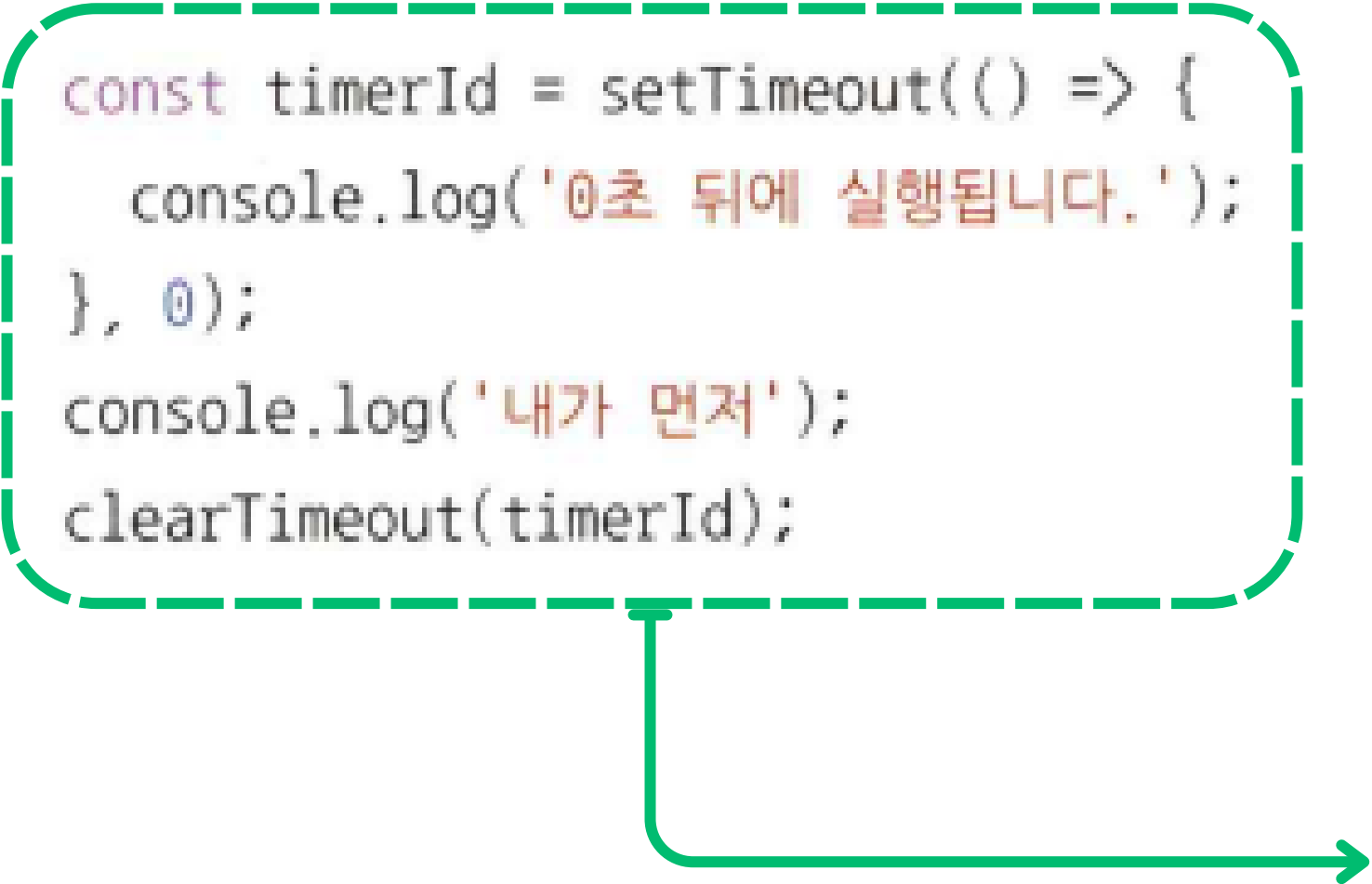
단, clearTimeout() 함수는 setTimeout() 함수의 콜백 함수가 아직 실행되지 않았을 때만 취소 가능

## setTimeout()

- clearTimeout() 예시

실행결과 : '내가 먼저' 출력, '0초 뒤에 실행됩니다.' 출력되지 않음.

```
const timerId = setTimeout(() => {  
  console.log('0초 뒤에 실행됩니다.');
```



```
}, 0);  
console.log('내가 먼저');  
clearTimeout(timerId);
```

### 실행 결과 이유

비동기 함수를 현재 실행 중인 동기 코드가 끝난 다음에 실행함.

<순서>

1. 동기 코드가 바로 실행
2. clearTimeout이 위에 등록해둔 setTimeout 타이머를 취소

## 3.2 스코프와 클로저

## SECTION 01

# 블록 스코프와 함수 스코프

모든 변수는 스코프가 존재

- var → 함수 스코프
- let → 블록 스코프

함수 스코프(var)

```
function b() {  
  var a = 1;  
}  
console.log(a);  
Uncaught ReferenceError: a is not defined
```

에러가 나는 이유는 ?

a는 함수 b() 안에 선언된 변수라서 함수 바깥에서는 접근 불가능

**함수 스코프** = 함수를 경계로 접근 가능 여부가 달라지는 것

▶ 함수가 끝날 때 함수 내부의 변수도 같이 사라진다고 할 수 있음.

모든 변수는 스코프가 존재

- var → 함수 스코프
- let → 블록 스코프

블록 스코프(let)

```
if (true) {  
  let a = 1;  
}
```

```
a;
```

```
Uncaught ReferenceError: a is not defined
```

에러가 나는 이유는 ?

let은 **블록 스코프**이기 때문에 블록 바깥에서는 블록 안에 있는 let에 접근할 수 없음. (const도 블록 스코프를 가짐)

▶ 블록이 끝날 때 내부의 변수도 같이 사라짐.

**\*\*블록이란\*\***

if 문, for 문, while 문, 함수에서 볼 수 있는 중괄호를 의미

## SECTION 02

# 클로저와 정적 스코프



클로저 : 외부 값에 접근하는 함수

모든 자바스크립트 함수는 클로저가 될 수 있음.

- 클로저 예시

```
const a = 1;  
const func = () => {  
  console.log(a);  
};
```

func() 함수는 자신 외부에 있는 변수 a를 사용하고 있음  
▶ func() 함수는 클로저

- 클로저 예시

함수 반환 예시

```
const func = (msg) => {  
  return () => {  
    console.log(msg);  
  };  
};
```

반환값인 익명 함수는 자신의 외부에 있는 msg 매개 변수를 사용함.

msg는 func()의 매개변수로 받은 값이지만,  
내부함수가 msg를 참조하므로 클로저가 됨. (반환된 함수도 클로저)

- 클로저 예시

정적 스코프 와 동적 스코프

```
const func = () => {  
  console.log(a);  
};  
if (true) {  
  const a = 1;  
  func();  
};
```

### 에러가 나는 이유는 ?

func()는 a가 선언되기 전에 만들어짐.

▶ 자바스크립트는 정적 스코프를 따름 (함수 선언 위치 기준으로 스코프 결정)

#### **\*\*정적 스코프\*\***

함수가 어디서 호출됐는지가 아니라 어디서 선언됐는지 기준으로 스코프 결정

#### **\*\* 동적 스코프\*\***

선언된 위치가 아니라 호출된 위치에 따라 접근할 수 있는 값이 달라짐.

## SECTION 03

# let과 var를 사용한 결과가 다른 이유

## let과 var를 사용한 결과가 다른 이유

예시 : ) for 문에서 setTimeout과 함께 var/let을 사용 했을 때 다른 결과가 나오는 이유

- 기본 코드 (var 사용)

```
const number = [1, 3, 5, 7];
for (var i = 0; i < number.length; i++) {
  setTimeout(() => {
    console.log(number[i]);
  }, 1000 * (i + 1));
}
```

결과 : 모두 undefined 출력

(1초 ~ 4초 후 전부 number[4], 즉 존재하지 않는 인덱스 출력)

→ 이유는 ?

- var는 함수 스코프 → i 는 전역에서 하나만 공유
- 반복문이 먼저 끝나고 i === 4 일 때 콜백이 실행
- 클로저 안의 i 는 모두 같은 4를 참조

## let과 var를 사용한 결과가 다른 이유

예시 : ) for 문에서 setTimeout과 함께 var/let을 사용 했을 때 다른 결과가 나오는 이유

- let 사용 시, 문제 해결

```
const number = [1, 3, 5, 7];
for (let i = 0; i < number.length; i++) {
  setTimeout(() => {
    console.log(number[i]);
  }, 1000 * (i + 1));
}
```

결과 : 1초 후 1, 2초 후 3, 3초 후 5, 4초 후 7 출력

이유는 ?

- let은 블록 스코프
- 반복문이 돌 때 마다 새로운 i가 생성됨.
- 클로저가 각각의 i를 기억함. → 의도한 대로 동작

## let과 var를 사용한 결과가 다른 이유

예시 : ) for 문에서 setTimeout과 함께 var/let을 사용 했을 때 다른 결과가 나오는 이유

- var로 해결 하는 법 - 고차 함수 helper()

```
const number = [1, 3, 5, 7];
function helper(j) {
  return () => {
    console.log(number[j], j);
  };
}

for (var i = 0; i < number.length; i++) {
  setTimeout(helper(i), 1000 * (i + 1));
}
```



- helper(i)는 매번 다른 j 값을 가진 함수를 리턴
- 각 클로저 안에서 j 값이 고정
- var여도 정확하게 동작 가능

## let과 var를 사용한 결과가 다른 이유

예시 : ) switch 문에서 스코프 주의사항

```
const type = 'a';
switch (type) {
  case 'a':
    let name = '제로초'; // ✅ X
    break;
  case 'b':
    let name = '레오'; // ✅ X
    break;
  case 'c':
    let name = '체리'; // ✅ X
    break;
}
```

문제점

: 모든 case에서 같은 스코프 → 변수 중복 선언 에러 발생



## let과 var를 사용한 결과가 다른 이유

예시 : ) switch 문에서 스코프 주의사항

```
switch (type) {  
  case 'a': {  
    let name = '제로초';  
    break;  
  }  
  case 'b': {  
    let name = '레오';  
    break;  
  }  
  case 'c': {  
    let name = '체리';  
    break;  
  }  
}
```

해결 방법

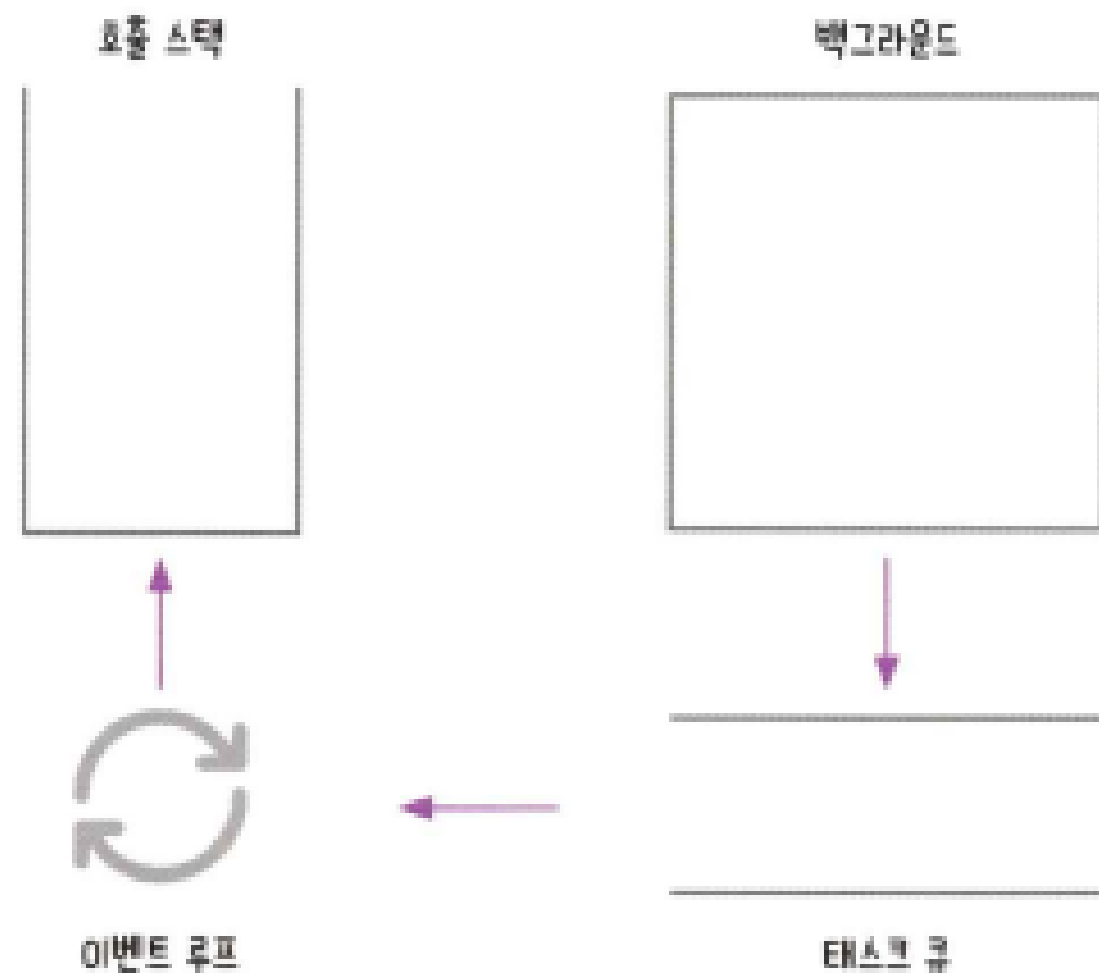
: 각 case에 중괄호로 블록 스코프 생성

## 3.3 호출 스택과 이벤트 루프

- 동기 / 비동기 코드 실행 순서 이해하기

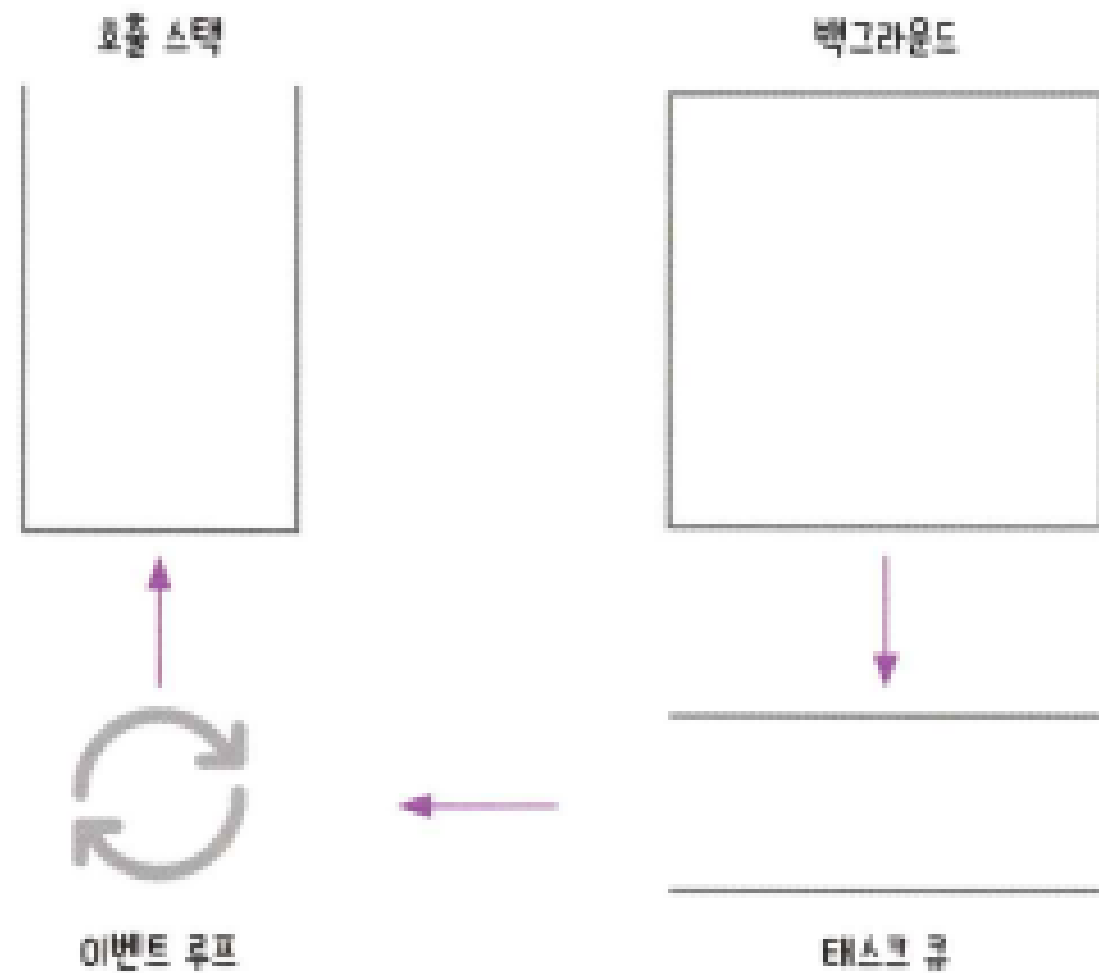
자바 스크립트는 싱글 스레드 언어 → 한 줄 씩 순서대로 코드 실행 (동기)

하지만 비동기 함수는 시간이 지난 후 실행 → 차이를 이해하기 위해서 **호출 스택** 과 **이벤트 루트**를 알아야함.



- 호출 스택 : 실행 중인 동기 코드가 쌓이고 빠지는 공간
- 백그라운드 : setTimeout, DOM 이벤트 등의 타이머와 비동기 작업이 대기하는 공간
- 태스크 큐 : 백그라운드에서 준비된 콜백 함수들이 줄 서는 공간
- 이벤트 루프 : 호출 스택이 비면, 태스크 큐에서 함수를 꺼내 실행하는 관리자

- 동기 / 비동기 코드 실행 순서 이해하기



흐름

1. 동기 함수 → **호출 스택**에 올라가 실행
2. `setTimeout`, `addEventListener` 같은 비동기 함수는 → **백그라운드**로 넘어감
3. 지정 시간이 지나면 콜백 → **태스크 큐에 등록**
4. **호출 스택이 비었을 때**, 이벤트 루프가 큐에서 꺼내 실행

**\*\*이벤트 루프\*\***

태스크 큐에서 호출 스택으로 함수를 이동시키는 존재

SECTION 01

# 호출 스택

### 호출 스택이란 ?

- 자바스크립트는 함수를 호출할 때마다 → 그 함수를 \*\* 호출 스택 \*\* 에 쌓음
  - 실행이 끝나면 → 스택에서 빠져나감
- ▶ 나중에 들어온 함수가 먼저 끝나야 밑에있는 함수도 끝낼 수 있음 (LIFO 구조)

## 기본 예제 흐름

```
function a() {  
  b();  
}  
function b() {  
  console.trace();  
}  
a();
```

### 실행 순서

1. a() 실행 → 스택에 a 쌓임
2. b() 실행 → 스택에 b 쌓임
3. console.trace() 실행  
→ b, a, anonymous 순서로 스택 확인

**\*\* console.trace() \*\***

: 함수의 호출 스택을 보여주는 메서드

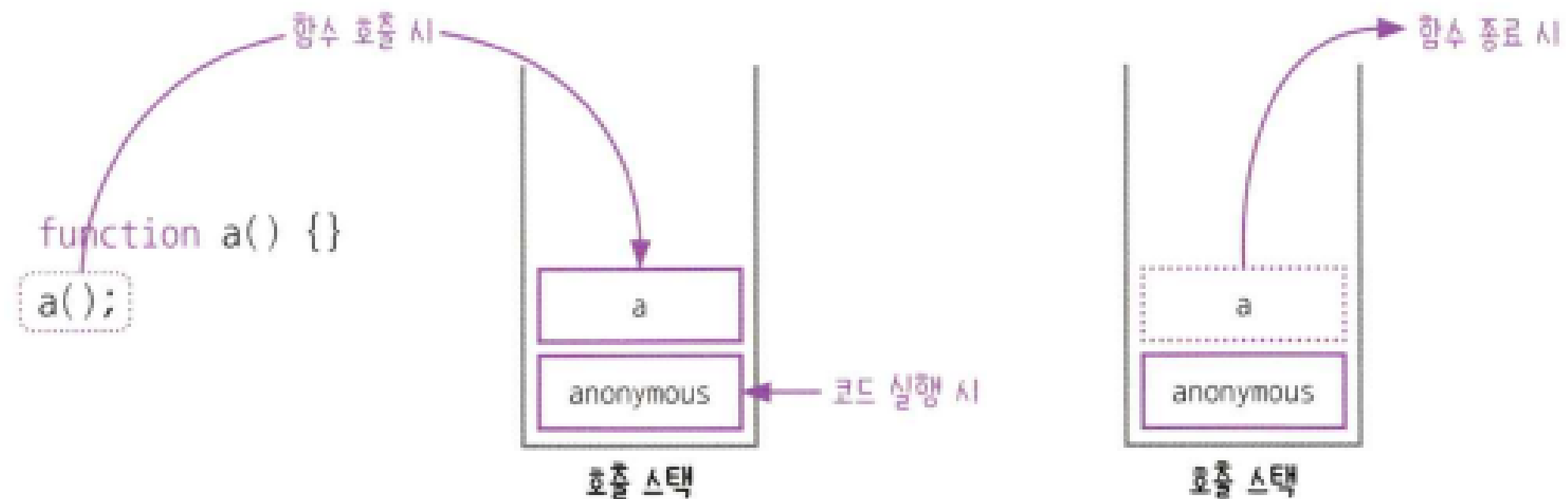
## 기본 예제

```
function a() {
  b();
}
function b() {
  console.trace();
}
a();
```

## 실행 결과

```
console.trace
b
a
(anonymous)
```

그림 3-3 a() 함수 호출과 종료 시 호출 스택 구조



호출 스택은 위에서 쌓이고 위에서 빠짐

**\*\*anonymous\*\***

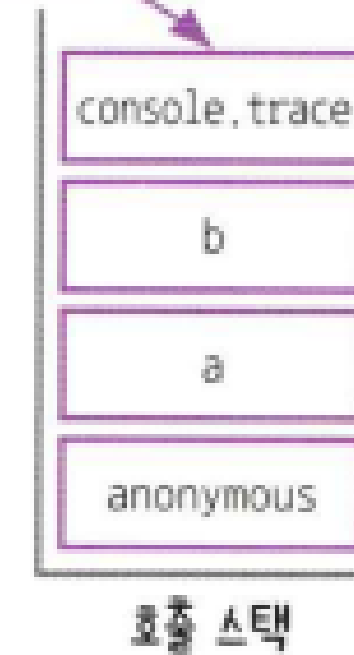
: 자바스크립트 코드가 처음 실행될 때 호출되는 함수



## 기본 예제

```
function a() {  
  b();  
}  
function b() {  
  console.trace();  
}  
a();
```

```
function a() {  
  b();  
}  
function b() {  
  console.trace();  
}  
a();
```



## 실행 결과

```
console.trace  
b  
a  
(anonymous)
```

**\*\* 빠지는 순서 \*\***

1. console.trace가 빠짐
2. b 빠짐
3. a 빠짐
4. anonymous 빠짐

SECTION 02

이벤트 루프

## 기본 예제

```
const timerId = setTimeout(() => {  
  console.log('0초 뒤에 실행됩니다.');
```

```
}, 0);
```

```
console.log('내가 먼저');
```

## 출력 결과

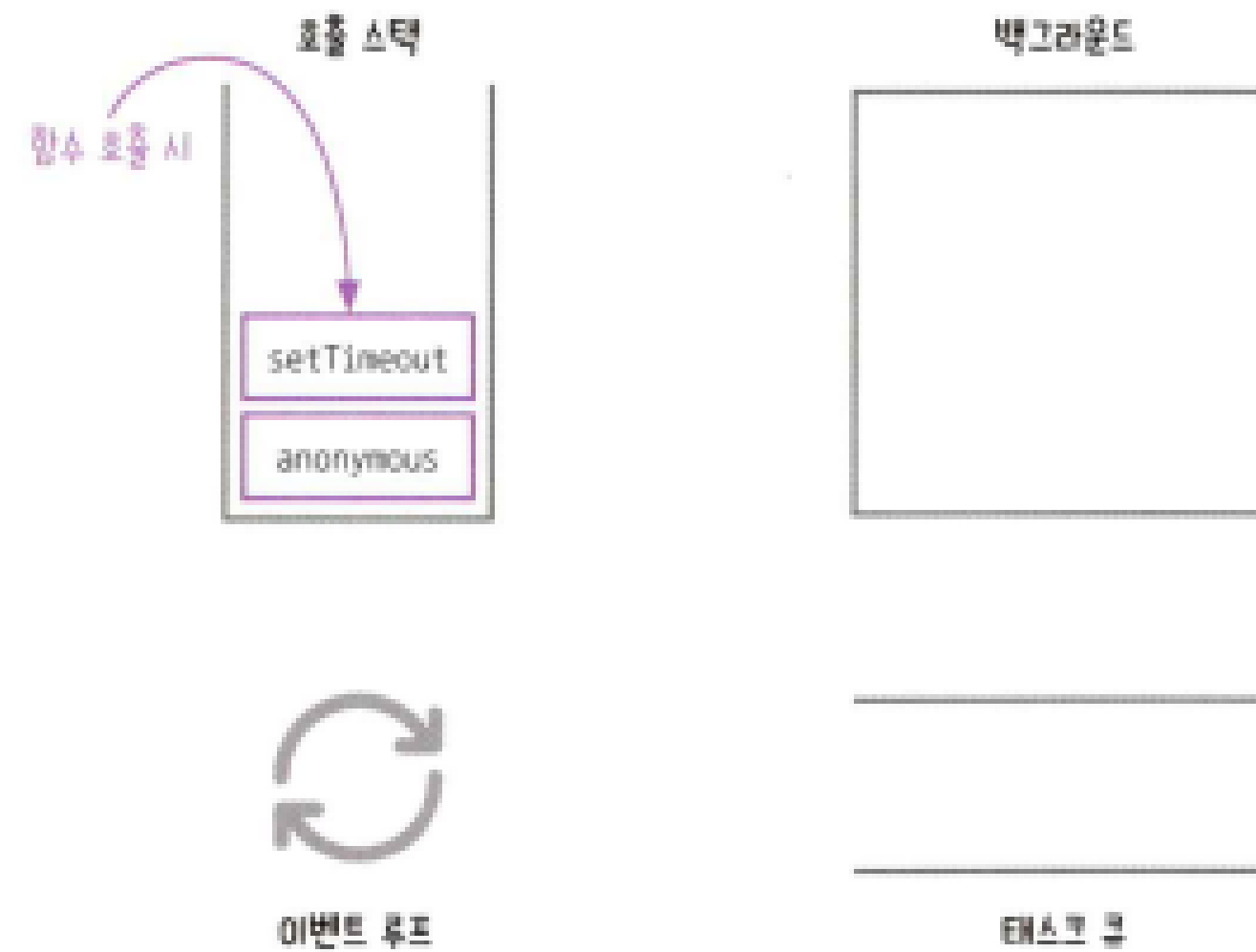
내가 먼저  
0초 뒤에 실행됩니다.

## 0초인데도 나중에 실행 되는 이유?

**\*\* 실행 흐름 \*\***

1. setTimeout()이 호출 → 콜백은 백그라운드로 이동
2. console.log('내가 먼저')가 바로 실행 (동기)
3. 지정 시간이 지나면 → 태스크 큐로 이동
4. 호출 스택이 비어있을 때 → 이벤트 루프가 큐에서 콜백을 꺼내 실행

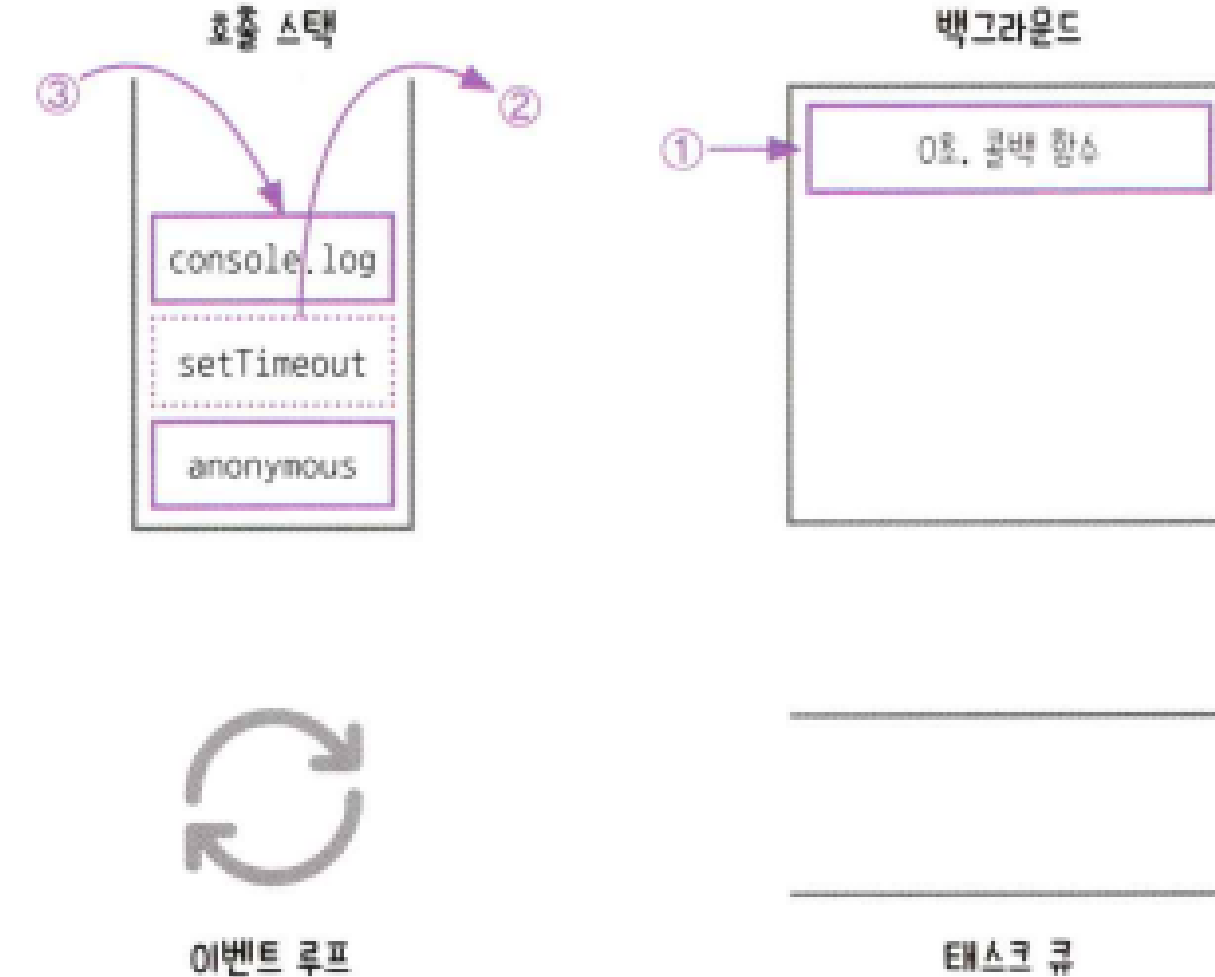
## 1. setTimeout() 함수 호출 시



## 이벤트 루프

### 1. setTimeout() 함수 종료 시

- ① setTimeout() 함수는 콜백 함수를 백그라운드로 보내고 종료
- ② setTimeout() 함수는 종료 되면서 호출 스택을 빠져나감
- ③ console.log('내가 먼저')가 실행  
setTimeout() 함수가 콘솔에 아무런 출력을 하지 않고 빠져나가므로  
console.log()가 setTimeout() 함수보다 먼저 실행되는 것처럼 보임



SECTION 03

# 재귀 함수

## 3.4 프로미스와 ASYNC/AWAIT

SECTION 01

프로미스



## 프로미스

new를 붙여 Promise 클래스를 호출 하면, 프로미스 객체 생성

형식

```
const <프로미스 객체> = new Promise((resolve, reject) => {  
  resolve(); // 프로미스 성공  
  // 또는  
  reject(); // 프로미스 실패  
});
```

인수로 콜백함수를 넣기

콜백 함수의 매개 변수 : resolve()와 reject()

▶ 콜백 함수 내부에서는 둘 중 하나만 호출

\*\* 둘 다 호출 하면 먼저 호출한 함수만 유효

then() 콜백함수 : resolve() 함수를 호출할 때 실행  
resolve()의 인수로 전달한 값은 then() 콜백 함수의 매개 변수로 전달

형식    <프로미스 객체> .then(<콜백 함수>);

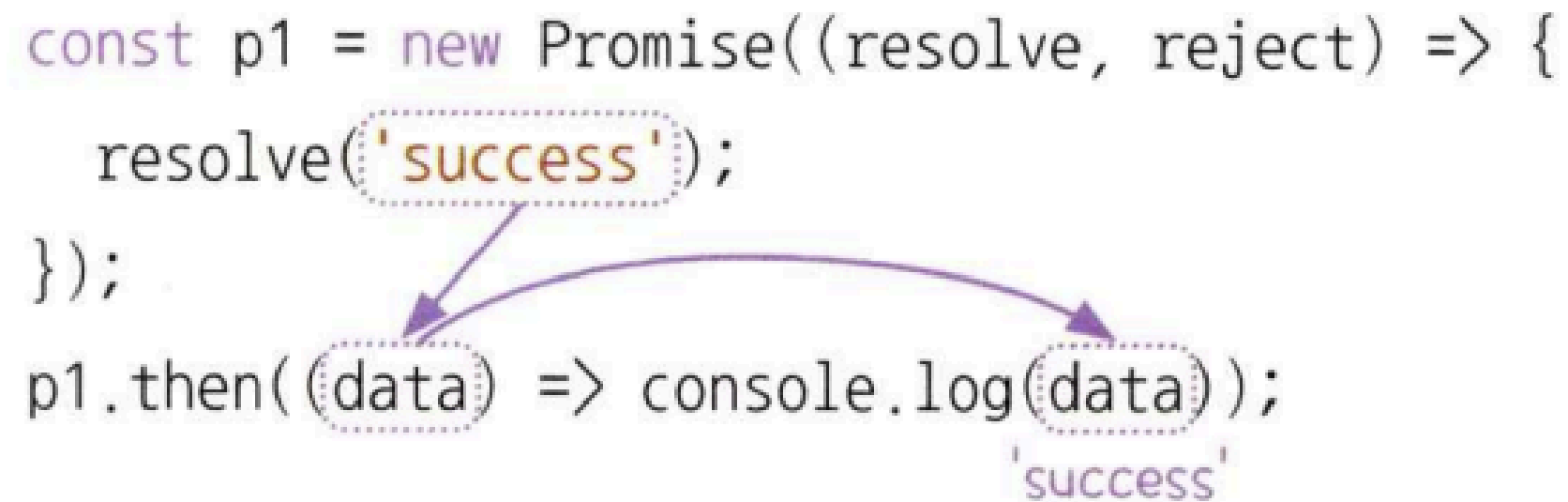
// 또는

<프로미스 객체> .catch(<콜백 함수>);

catch() 콜백함수 : reject() 함수를 호출할 때 실행  
reject()의 인수로 전달된 값은 catch() 콜백 함수의 매개 변수로 전달

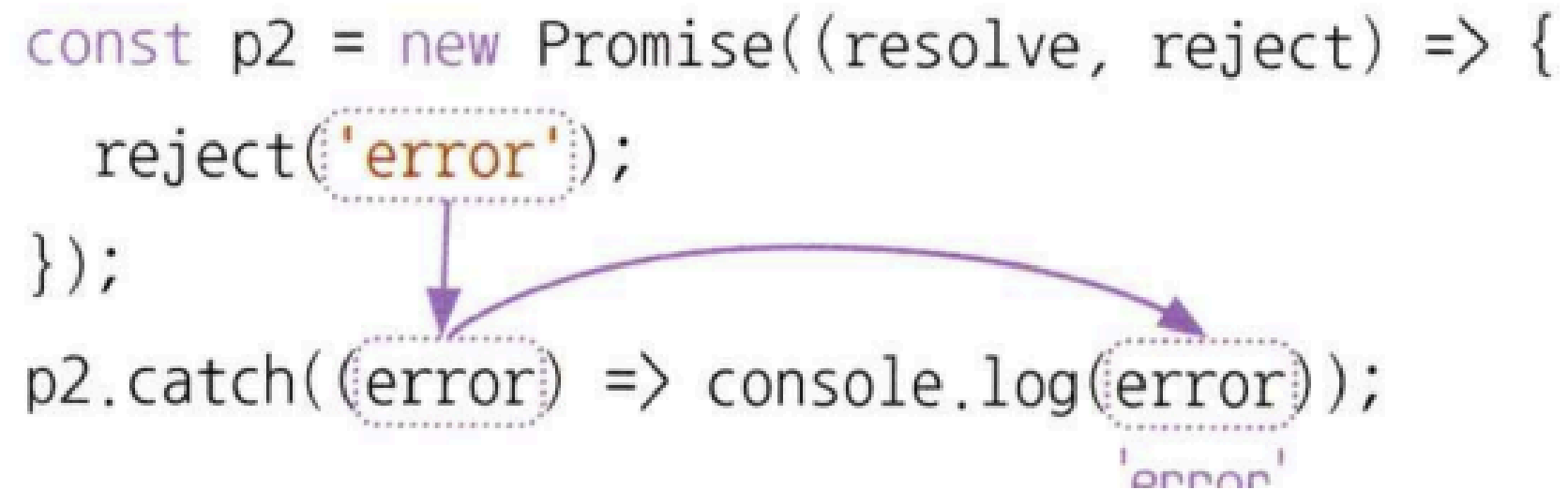
- resolve() 호출 시 then() 실행, reject() 호출 시 catch() 실행 예시

```
const p1 = new Promise((resolve, reject) => {  
  resolve('success');  
});  
p1.then((data) => console.log(data));
```



The diagram illustrates the execution flow for a successful promise. A purple arrow originates from the string 'success' (which is enclosed in a dashed purple oval) within the resolve function of the Promise constructor. This arrow points to the parameter 'data' (also in a dashed purple oval) of the then() method. A second purple arrow then points from 'data' to the console.log() function, which is shown with 'data' as its argument. Finally, a curved purple arrow points from the 'data' parameter to the value 'success' (in a dashed purple oval) that is printed to the console.

```
const p2 = new Promise((resolve, reject) => {  
  reject('error');  
});  
p2.catch((error) => console.log(error));
```



The diagram illustrates the execution flow for a rejected promise. A purple arrow originates from the string 'error' (enclosed in a dashed purple oval) within the reject function of the Promise constructor. This arrow points to the parameter 'error' (also in a dashed purple oval) of the catch() method. A second purple arrow then points from 'error' to the console.log() function, which is shown with 'error' as its argument. Finally, a curved purple arrow points from the 'error' parameter to the value 'error' (in a dashed purple oval) that is printed to the console.

- `reject()`를 호출했을 때, `catch()` 메서드를 붙이지 않으면 에러 발생

**\*\* but, `resolve()`를 호출했을 때, `then()`을 안붙여도 에러 발생 안함. \*\***

▶ 이유는 ?

`resolve`는 성공 상태로 `promise`를 마무리 짓기 때문에 성공 결과를 처리하지 않더라도

자바스크립트는 문제 삼지 않음. (결과를 사용하지 않았다는 뿐임)

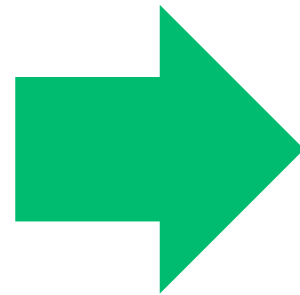
`reject`는 실패 상태로 처리 하지 않으면 자바스크립트는 에러를 띄움

---

```
const p2 = new Promise((resolve, reject) => {  
  reject('error');  
}); // Uncaught (in promise) error
```

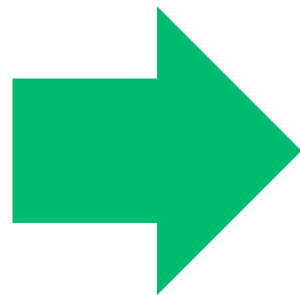
- 비동기 코드를 프로미스 문법으로 전환하는 예제

비동기 코드



```
const timerId = setTimeout(() => {  
  console.log('0초 뒤에 실행됩니다.');
```

프로미스 코드



```
const setTimeoutPromise = new Promise((resolve, reject) => {  
  setTimeout(() => {  
    resolve();  
  }, 0);  
});  
setTimeoutPromise.then(() => {  
  console.log('0초 뒤에 실행됩니다.');
```

- 비동기 코드를 promise로 바꾸는 이유

## 1. 가독성 향상

- ▶ 콜백 중첩 없이 .then()으로 순차 실행 가능

```
setTimeoutPromise(1000)  
  .then(() => fetchData())  
  .then(data => render(data));
```

- 비동기 코드를 promise로 바꾸는 이유

## 2. 에러 처리 통합이 쉬움

- ▶ catch() 하나로 전체 비동기 흐름의 에러를 한 번에 처리 가능  
try-catch 없이도 안정적으로 코드 작성 가능

```
function doTask(callback) {  
  try {  
    throw new Error('Error!');  
  } catch (e) {  
    callback(e);  
  }  
}  
  
doTask((err) => {  
  if (err) console.error('Error:', err.message);  
});
```

```
function doTask() {  
  return new Promise((resolve, reject) => {  
    reject(new Error('Error!'));  
  });  
}  
  
doTask()  
  .then(() => {  
    console.log('Success!');  
  })  
  .catch(err => {  
    console.error('Error:', err.message);  
  });
```

- 비동기 코드를 promise로 바꾸는 이유

## 3. 결과 값을 저장하고 꺼낼 수 있음

▶ 결과를 저장하고 있다가 then()으로 꺼내 쓸 수 있음.

```
const p = new Promise(resolve => setTimeout(() => resolve('완료!'), 1000));  
p.then(result => console.log(result)); // → '완료!'
```



- 비동기 코드를 promise로 바꾸는 이유

### 3. then()이나 catch() 메서드를 나중에 붙일 수 있음.

```
const p1 = new Promise((resolve, reject) => {  
  resolve('프로미스 작업을 합니다.');
```

});  
console.log('다른 일을');  
console.log('열심히');  
console.log('하다가');

p1.then(console.log);

다른 일을  
열심히  
하다가  
프로미스 작업을 합니다.

- **then()이나 catch() 메서드는 연달아서 사용 가능**

▶ 비동기 로직을 순차적으로 연결 가능

```
const promise = setTimeoutPromise(0);
promise
  .then(() => {
    return 'a';
  })
  .then((data) => {
    console.log(data); // a
    return 'b';
  })
  .then((data) => {
    console.log(data); // b
  });
```

- finally()

▶ then()과 catch()의 실행이 끝난 후에 finally() 가 있으면 무조건 실행

```
const promise = setTimeoutPromise(0);
promise
  .then(() => {
    console.log('0초 뒤에 실행됩니다. ');
  })
  .catch((err) => {

    console.log('에러 발생 시 실행됩니다. ');
  })
  .finally((err) => {
    console.log('성공이든 실패든 무조건 실행됩니다. ');
  });
```

## SECTION 02

# async/await

## promise로 작성한 코드는 코드 작성 순서와 실행 순서 다름

```
const setTimeoutPromise = (ms) => new Promise((resolve, reject) => {  
  setTimeout(resolve, ms);  
});  
setTimeoutPromise(1000).then(() => {  
  console.log('1초 뒤에 실행됩니다.');});  
console.log('내가 먼저');
```

내가 먼저  
1초 뒤에 실행됩니다.

- **setTimeoutPromise(1000)는 비동기 작업**
  - **then() 안의 코드는 나중에 실행됨(1초 뒤)**
  - **console.log('내가 먼저')는 동기 코드 ... 바로 실행**
- ▶ 비동기 코드를 백그라운드에 밀어두고 바로 밑에 있는 코드를 실행

## async/await

promise로 작성한 코드는 코드 작성 순서와 실행 순서 다름 → **async/await문으로 해결**

```
const setTimeoutPromise = (ms) => new Promise((resolve, reject) => {  
  setTimeout(resolve, ms);  
});
```

```
await setTimeoutPromise(1000);  
console.log('1초 뒤에 실행됩니다.');
```

```
console.log('내가 나중에');
```

1초 뒤에 실행됩니다.

내가 나중에

- await : 해당 작업이 끝날 때까지 기다려줌
- setTimeoutPromise(1000)이 끝날 때까지 아래 코드 실행 안함
- 1초 뒤, 아래 콘솔들이 찍힘

- 프로미스가 아닌 비동기 코드에는 await를 적용하는 것은 의미가 없음 .

▶ setTimeout()은 프로미스가 아니므로 await가 적용 X

해결 방안 : setTimeout()을 프로미스로 바꾼 후 await를 붙이면 됨.

---

```
await setTimeout(() => {  
  console.log('1초 뒤에 실행됩니다.');
```

```
}, 0);  
  
console.log('내가 먼저');
```

내가 먼저

1초 뒤에 실행됩니다.

---

- 함수 내부에서 await를 사용하면 에러 발생

▶ await는 async 함수에서 사용 가능

해결 방안 : main() 함수를 async 함수로 전환

전환 방법 : function 예약어 앞에 async를 붙이면 됨.

```
const setTimeoutPromise = (ms) => new Promise((resolve, reject) => {
  setTimeout(resolve, ms);
});

function main() {
  await setTimeoutPromise(1000);
  console.log('1초 뒤에 실행됩니다. ');
  console.log('내가 나중에');
}

main();
```

Uncaught SyntaxError: await is only valid in async functions and the top level bodies of modules



- 함수 내부에서 await를 사용하면 에러 발생

▶ await는 async 함수에서 사용 가능

해결 방안 : main() 함수를 async 함수로 전환

전환 방법 : function 예약어 앞에 async를 붙이면 됨.

```
const setTimeoutPromise = (ms) => new Promise((resolve, reject) => {  
  setTimeout(resolve, ms);  
});
```

```
async function main() {  
  await setTimeoutPromise(1000);  
  console.log('1초 뒤에 실행됩니다.');
```

```
  console.log('내가 나중에');
```

```
}
```

```
main();
```

1초 뒤에 실행됩니다.

내가 나중에

- 화살표 함수도 앞에 `async`를 붙여 `async` 함수로 만들기 가능

```
const setTimeoutPromise = (ms) => new Promise((resolve, reject) => {  
  setTimeout(resolve, ms);  
}));
```

```
const main = async () => {  
  await setTimeoutPromise(1000);  
  console.log('1초 뒤에 실행됩니다.');
```

```
}  
main();
```

1초 뒤에 실행됩니다.

## SECTION 03

# try-catch 문으로 에러 처리하기

## try-catch 문으로 에러 처리하기

- **await** 쓸 때, 에러 처리 하기

- ▶ **await**는 **catch**를 사용 불가능

1. p1 이라는 Promise가 만들어짐. → 실패 상태
2. **await** p1 → p1 기다림
3. 실패 → **await**가 바로 에러 던짐
4. **try-catch** 없으면 실행 중단 및 콘솔에 에러 뜸

```
const p1 = new Promise((resolve, reject) => {  
  reject('에러!');  
});  
  
await p1;  
Uncaught 에러!
```

결론 :: 에러 처리를 반드시 **try-catch**문으로 해야함.

## try-catch 문으로 에러 처리하기

- await 쓸 때, 에러 처리 하기

▶ try-catch문으로 에러 처리 하기

1. p1 이라는 Promise가 만들어짐 → 즉시 reject('에러') 호출 → 실패 상태
2. await p1 → p1 기다림
3. 실패 → await가 바로 에러 던짐
4. try-catch 있으니 catch 블록으로 넘어가 에러 처리됨.

```
const p1 = new Promise((resolve, reject) => {  
  reject('에러!');  
});  
try {  
  await p1;  
} catch (error) {  
  console.log(error);  
}
```

에러!

- try-catch문에도 finally 문 추가 가능

```
const p1 = new Promise((resolve, reject) => {  
  reject('에러!');  
});  
try {  
  await p1;  
} catch {  
  console.log('에러인 경우');  
} finally {  
  console.log('성공이든 에러든 마지막에 실행됩니다.');}
```

에러인 경우

성공이든 에러든 마지막에 실행됩니다.