






















- Promise, async, await
 - 1. □□□□
 - 2. Callback □□
 - 3. Callback -> Promise -> async/await

Promise, async, await

1.

#

-                   

- 10초, 1초 후에 10을 출력하는 코드를 작성하시오.
- 10을 출력한 후 1초 후에 1을 출력하는 코드를 작성하시오.
- 10을 출력한 후 1초 후에 1을 출력하는 코드를 작성하시오.

```
let a = 10
```

```
setTimeout(function callback() {
  console.log('a : ', a)
}, 3000)
```

```
console.log('Finished')
```

```
//Finished
```

```
//a: 10
```

10초 후 10 출력

- 10초 후 10을 출력하는 코드를 작성하시오.
- 10, 1초 후에 1을 출력하는 코드를 작성하시오.
- 10을 출력한 후 1초 후에 1을 출력하는 코드를 작성하시오.
- 10을 출력한 후 1초 후에 1을 출력하는 코드를 작성하시오.

10초 후 10 출력, 1초 후 1 출력

10을 출력한 후 1초 후에 1을 출력하는 코드를 작성하시오. 10을 출력한 후, 1초 후에 1을 출력하는 코드를 작성하시오.

10을 출력한 후 1초 후에 1을 출력하는 코드를 작성하시오?

10을 출력한 후 1초 후에 1을 출력하는 코드를 작성하시오. 10을 출력한 후 1초 후에 1을 출력하는 코드를 작성하시오. 10을 출력한 후 1초 후에 1을 출력하는 코드를 작성하시오.

10초 후 10 출력, 1초 후 1 출력

1. API (10초 후 10 출력, 1초 후 1 출력)

이 코드는 API에서 데이터를 가져오는 함수를 정의하고, 3초 후에 데이터를 가져오는 데 사용됩니다. 이 코드는 콘솔에 메시지를 출력하고, 3초 후에 데이터를 가져오는 데 사용됩니다.

```
function fetchDataFromAPI() {
  console.log("데이터를 가져오는 중입니다.");
  // 3초 후에 데이터를 가져오는 데 사용됩니다
  setTimeout(() => {
    console.log("데이터를 가져왔습니다!");
    // 데이터를 가져오는 데 사용됩니다
  }, 3000);

  console.log("데이터를 가져오는 중입니다.");
}

fetchDataFromAPI();
```

<코딩>

```
const fs = require('fs');
const path = require('path');
const { exec } = require('child_process');

// 데이터를 가져오는 데 사용됩니다
```

이 코드는 API에서 데이터를 가져오는 함수를 정의하고, 3초 후에 데이터를 가져오는 데 사용됩니다. 이 코드는 콘솔에 메시지를 출력하고, 3초 후에 데이터를 가져오는 데 사용됩니다.

2. 파일 시스템 (fs 모듈을 사용하여)

이 코드는 파일 시스템에서 데이터를 가져오는 함수를 정의하고, 3초 후에 데이터를 가져오는 데 사용됩니다. 이 코드는 콘솔에 메시지를 출력하고, 3초 후에 데이터를 가져오는 데 사용됩니다.

<코딩> (Node.js 모듈)

```
const fs = require('fs');

console.log("데이터를 가져오는 중입니다.");

fs.readFile('example.txt', 'utf8', (err, data) => {
  if (err) {
    console.log("데이터를 가져오는데 실패했습니다:", err);
  } else {
    console.log("데이터를 가져왔습니다:", data);
  }
});
```

```
    console.log("data :", data);
  }
});

console.log("data is not null");
```

<code>

```
data is not null
data is not null
(data is not null)
data is not null
```

data is not null. data is not null, data is not null
data is not null.

3. data is not null

data is not null. data is not null. data is not null.
data is not null. data is not null. data is not null.

<code>

```
document.getElementById("submitButton").addEventListener("click", function() {
  console.log("data is not null");

  // data is not null
  setTimeout(() => {
    console.log("data is not null!");
  }, 2000);

  console.log("data is not null");
});
```

<code>

```
data is not null
data is not null
(2 data is not null)
data is not null!
```


2. Callback

1. Callback (Callback)

Callback?

Callback is a function that is passed as an argument to another function, and is executed after the function has finished its execution.

```
function sayHello() {  
  console.log("Hello!");  
}  
  
function executeCallback(callback) {  
  console.log("Before callback");  
  callback(); // Call the callback function  
  console.log("After callback");  
}  
  
executeCallback(sayHello);
```

<Execution Order>

1. executeCallback is called, sayHello is passed as an argument
2. console.log("Before callback") is executed
3. sayHello() is called, "Hello!" is printed
4. console.log("After callback") is executed

<Execution Order>

```
Before callback  
Hello!  
After callback
```


2. 堆棧 (Call Stack) 是什麼？

堆棧(Call Stack)

堆棧是一個用於追蹤函數調用的數據結構。

當一個函數被調用時，它會被添加到堆棧中，並執行其代碼。當函數執行完畢後，它會被從堆棧中移除。

堆棧的運作方式類似於一個堆棧，只能從頂端添加或移除元素。這確保了函數調用的順序性。

3. 事件循環 (Callback Queue) 是什麼？

事件循環(Callback Queue)

事件循環是一個用於處理异步操作的機制。

當一個异步操作完成時，它的回调函數會被添加到事件循環中，並等待被執行。

```
console.log("Start");

setTimeout(() => {
  console.log("Timeout callback");
}, 1000);

console.log("End");
```

<輸出結果>

- console.log("Start") 輸出 → "Start"
- setTimeout 延遲 → 1000ms 後 → 執行回调函數
- console.log("End") 輸出 → "End"
- 1000ms 後，回调函數被添加到事件循環 → "Timeout callback"

<輸出結果>

Start
End
Timeout callback

4. 타이머 타이머

타이머 타이머 타이머

타이머 타이머

타이머 타이머 타이머, 타이머 타이머 타이머 타이머

타이머 타이머 타이머 타이머 타이머 타이머 타이머 타이머

타이머 타이머 + 타이머 타이머 + 타이머 타이머

```
console.log("Start");

setTimeout(() => {
  console.log("Callback 1");
}, 1000);

setTimeout(() => {
  console.log("Callback 2");
}, 500);

console.log("End");
```

<타이머 타이머>

1. console.log("Start") 타이머 → 타이머 타이머 타이머 → "Start" 타이머
2. setTimeout 타이머 → 타이머 타이머 타이머 타이머 타이머 (500ms 타이머 1000ms)
3. console.log("End") 타이머 → 타이머 타이머 타이머 → "End" 타이머
4. 500ms 타이머, 타이머 타이머 타이머(Callback 2) 타이머 타이머 타이머 타이머 타이머 → 타이머 타이머 → "Callback 2" 타이머
5. 1000ms 타이머, 타이머 타이머 타이머(Callback 1) 타이머 타이머 타이머 타이머 타이머 → 타이머 타이머 → "Callback 1" 타이머

<div div>

Start

End

Callback 2

Callback 1

div

div div div div div

1. div(Call Stack)

JavaScript의 실행 엔진은 코드를 실행할 때 Call Stack을 사용합니다. 이 구조는 코드가 실행되는 순서를 추적하는 데 사용됩니다.

2. div(Callback Queue)

이 구조는 setTimeout, Promise 등 비동기 작업을 실행할 때 사용됩니다. 이 구조는 코드가 실행되는 순서를 추적하는 데 사용됩니다.

3. div(Event Loop)

이 구조는 이벤트 루프를 실행할 때 사용됩니다. 이 구조는 코드가 실행되는 순서를 추적하는 데 사용됩니다.

4. div(Callback div)

이 구조는 콜백 함수를 실행할 때 사용됩니다. 이 구조는 코드가 실행되는 순서를 추적하는 데 사용됩니다.

divdivdiv

div div

```
function asd(asdf, callback) {  
  console.log("a");  
}
```

```
callback(); // sdf() 실행  
console.log("b");  
}  
  
function sdf() {  
  console.log("c");  
}  
  
asd("name", sdf); // 실행
```

실행 순서(시간 순서)

asd("name", sdf) 실행: 먼저 asd 실행 -> console.log("a") 실행: "a" 출력 -> 실행 후,
console.log 실행 후 실행 순서

callback() 실행: callback()은 실행된 sdf 실행된 후 sdf() 실행 -> sdf 실행 후 실행된 후 -> sdf
실행된 후 console.log("c")은 실행된 "c"을 출력. ->실행 후, sdf는 실행된 후 실행.

먼저 asd 실행 후 console.log("b")은 실행. -> "b" 출력 -> console.log("b")은 실행된 후 -
> asd 실행 후 실행된 후 실행된 후

<실행 순서>

```
a  
c  
b
```

실행 순서 (setTimeout, Promise.then, fetch)은 실행된 후 실행된 후 실행된 후
실행된 후 실행된 후 실행된 후 실행된 후. 실행, 실행된 후 실행된 후
실행된 후 실행된 후 실행된 후 실행된 후.

실행된 후 실행된 후

asd 실행

```
asd
```

console.log("a") 실행

```
asd  
console.log
```

sdf ☐

asd

sdf

console.log("c") ☐

asd

sdf

console.log

sdf ☐ → **console.log("b")** ☐

asd

console.log

asd ☐

(☐ ☐)

3. Callback -> Promise -> async/await

1. Callback

Callback 是函数回调函数。

它接收一个函数 2 个参数 Dain 回调函数 回调函数 回调函数。

```
function getName(cb) {  
  setTimeout(() => {  
    cb("Dain");  
  }, 2000);  
}
```

它接收回调函数 回调函数 getName 回调函数 回调函数 回调函数。

```
getName((name) => {  
  console.log(name);  
})  
// 2秒 回调 Dain
```

它接收 getName 回调函数 Dain 回调函数 3 秒 回调函数 回调函数?

回调函数 getName 回调函数 回调函数 回调函数 2 秒 回调 Dain 回调函数 回调函数。

```
getName((name) => {  
  console.log(name);  
})  
  
getName((name) => {  
  console.log(name);  
})
```

```

getName((name) => {
  console.log(name);
})
// 20
// Dain
// Dain
// Dain

```

이름을 받아서 콘솔에 출력하는 함수를 3번 호출하면 어떻게 될까요?

20, Dain, Dain, Dain 이렇게 출력될 것입니다.

```

function getName(cb) {
  setTimeout(() => {
    cb("Dain");
  }, 2000);
}

function getAge(cb) {
  setTimeout(() => {
    cb(30);
  }, 2000);
}

function getAddress(cb) {
  setTimeout(() => {
    cb("Seoul");
  }, 2000);
}

```

이름, 나이, 주소를 받아서 console.log로 출력하는 함수를 어떻게 만들까요?

```

getName((name) => {
  getAge((age) => {
    getAddress((address) => {
      console.log(name, age, address)
    })
  })
})

```

이름, 나이, 주소가 있는 사용자 name, age, address를 가져오는 함수를 작성합니다.

3초 후, 2초 후, 6초 후 Dain 30 Seoul를 로그로 출력합니다.

이 함수가 Promise를 반환합니다. Promise는 3초 후 resolve, 2초 후 reject를 합니다.

이 함수가 Promise를 반환하는지 확인합니다?

2. Promise

Promise는 비동기 작업을 처리하는 데 사용됩니다.

이름, 나이, 주소를 가져오는 getName, getAge, getAddress 함수를 Promise로 변환합니다.

```
function getName() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve("Dain");
    }, 2000);
  })
}

function getAge() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve(30);
    }, 2000);
  })
}

function getAddress() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve("Seoul");
    }, 2000);
  })
}
```

이 함수는 Promise를 반환합니다. Promise는 2초 후 resolve를 하고, 2초 후 reject를 합니다.

getName().then((res) => {

```
  console.log(res);
})
```

```
getAge().then((res) => {
  console.log(res);
})
```

```
getAddress().then((res) => {
  console.log(res);
})
```

getName().then((res) => {

getAge().then((res) => {

Promise

```
.all([getName(), getAge(), getAddress()])
.then((res) => {
  const [name, age, address] = res;
  console.log(name, age, address)
})
```

Promise.all()는 모든 Promise가 resolve된 후, 결과를 배열로 반환합니다.

getName, getAge, getAddress 각각 Promise를 반환하는 함수를 Promise.all()로 감싸면, Promise.all()은 모든 Promise가 resolve된 후, 결과를 배열로 반환합니다. 예를 들어, Promise.all([getName(), getAge(), getAddress()])는 Promise.all()이 반환하는 Promise가 resolve된 후, [name, age, address]를 반환합니다.

3. async/await

async/await은 Promise를 더 쉽게 사용할 수 있도록 도와줍니다.

async/await은 Promise를 더 쉽게 사용할 수 있도록 도와줍니다. async 함수는 Promise를 반환합니다.

await은 Promise가 resolve될 때까지 기다린 후, 결과를 반환합니다. 예를 들어, await getName()은 Promise가 resolve된 후, name을 반환합니다.

```
(async () => {  
  const name = await getName();  
  const age = await getAge();  
  const address = await getAddress();  
  
  console.log(name, age, address);  
})();
```



Promise의 async/await는 코드를 좀 더 읽기 쉽습니다. Promise를 사용하면 코드가 좀 더 복잡해집니다. Promise를 사용하면 코드가 좀 더 복잡해집니다. Promise를 사용하면 코드가 좀 더 복잡해집니다.