

## 2. Callback 함수

### 1. 콜백 함수 (Callback)

콜백 함수?

콜백 함수란 어떤 함수를 호출할 때, 그 함수가 종료된 후 호출될 함수를 전달하는 기법이다.

```
function sayHello() {  
  console.log("Hello!");  
}  
  
function executeCallback(callback) {  
  console.log("Before callback");  
  callback(); // 콜백 함수 호출  
  console.log("After callback");  
}  
  
executeCallback(sayHello);
```

<실행 결과>

1. executeCallback 함수 호출, sayHello 함수가 호출됨
2. console.log("Before callback") 실행
3. sayHello() 실행, "Hello!" 출력
4. console.log("After callback") 실행

<결과 출력>

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

## 2. 堆棧 (Call Stack) 是什麼？

### 堆棧(Call Stack)

堆棧是 JavaScript 引擎用來追蹤函式調用的棧結構。當一個函式被調用時，它會被推入堆棧。當該函式執行完畢後，它就會從堆棧中彈出。堆棧中的每個元素都包含函式的上下文、參數和返回地址。堆棧的增長和縮減是動態的，它反映了當前正在執行的函式調用鏈。

## 3. 事件循環 (Callback Queue) 是什麼？

### 事件循環(Callback Queue)

事件循環是 JavaScript 引擎用來處理异步操作的機制。它負責調度那些已經完成但尚未被執行的回调函式。事件循環會不斷檢查是否有新的异步操作完成，如果有，它就會將這些操作的回调函式推入到事件循環的隊列中，等待主執行緒處理。

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

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

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

<執行順序>

1. console.log("Start") 執行 → "Start" 輸出
2. setTimeout 執行 → 將回调函式推入到事件循環隊列
3. console.log("End") 執行 → "End" 輸出
4. 1秒後，事件循環隊列中的回调函式被取出 → "Timeout callback" 輸出

<執行順序>

Start  
End  
Timeout callback

## 4.

   ,    

  +   +  


















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

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

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

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

< 

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

<div>

Start

End

Callback 2

Callback 1

## # div

div div div div div

### 1. div(Call Stack)

JavaScript은 동기 방식과 비동기 방식을 지원하며, 비동기 방식은 Call Stack을 사용하여 처리됩니다. 이 방식은 비동기 작업을 처리하는 데 사용됩니다.

### 2. div(Callback Queue)

이 방식은 비동기 작업을 처리하는 데 사용됩니다. 이 방식은 비동기 작업을 처리하는 데 사용됩니다. 이 방식은 비동기 작업을 처리하는 데 사용됩니다.

### 3. div(Event Loop)

이 방식은 비동기 작업을 처리하는 데 사용됩니다. 이 방식은 비동기 작업을 처리하는 데 사용됩니다.

### 4. div(Callback div)

이 방식은 비동기 작업을 처리하는 데 사용됩니다. 이 방식은 비동기 작업을 처리하는 데 사용됩니다.

## # div

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 후 -> console.log("a") 실행: "a" 실행 -> 실행 후,  
console.log 실행 후 실행 후 실행

callback() 실행: callback() 실행 후 실행 sdf 실행 후 실행 sdf() 실행 -> sdf 실행 후 실행 후 실행 -> sdf  
실행 후 실행 console.log("c") 실행 후 실행 "c" 실행 후 실행. -> 실행 후, sdf 실행 후 실행 후 실행.

실행 asd 실행 후 실행 console.log("b") 실행 후 실행. -> "b" 실행 -> console.log("b") 실행 후 실행 후 실행 -  
> asd 실행 후 실행 후 실행 후 실행 후 실행

<실행 순서>

a  
c  
b

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

실행 순서 실행

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

```
(  )
```