

# Node.js Server

Node.js & Javascript Basic

2023-1 T.G.WinG 강의식 스터디

---

배승호



# Javascript는 어떤 언어일까?

## ❖ 자바스크립트(Javascript, JS)는 객체 기반의 스크립트 프로그래밍 언어

- 주로 웹 브라우저에서 웹 페이지를 구성하는데 사용되는 언어이다
- 다른 응용 프로그램의 내장 객체에도 접근할 수 있는 기능을 가지고 있다
- **Node.js**와 같은 런타임 환경과 같이 **서버 프로그래밍**에도 활용되고 있다

## ❖ 자바스크립트는 자바와 관련이 없음

- 처음 언어를 만든 당시에는 Mocha라는 이름으로 지었으나, 언어가 **Java 개발자들을 타겟**으로 하고 있었기에 이름을 확정 짓는 과정에서 마케팅 전략으로 Javascript라는 이름으로 최종 결정되었다

## ❖ 그래도 자바와 유사한 점은 존재

- Java와 JS는 완전히 다른 언어이지만, 두 언어의 구문은 **C/C++ 구문을 Base**로 만들었기 때문에 Basic한 구문은 굉장히 유사하다
- 하지만, **내부 동작 로직은 완전히 다르므로** Javascript를 온전히 사용하려면 JS에 대해서 명확히 이해해야 한다

# 기초 문법 실습

❖ C/C++ 구문을 베이스로 하기 때문에 **기초 문법은 매우 유사하다**

❖ 출력 함수

```
console.log("Hello, World!")
```

❖ 변수 할당

```
var 정수형_변수 = 1234  
var 실수형_변수 = 12.34  
var 문자열_변수 = "1234"  
var 리스트_변수 = [1,2,3,4]  
var 객체_변수 = {1:2, 3:4}
```



# 기초 문법 실습

## ❖ if 조건문

```
if(1 === 1){  
    console.log("1이면 1이지~")  
} else if{  
    console.log("2 겠느냐~")  
} else{  
    console.log("3도 아니겠지~")  
}
```



# 기초 문법 실습

## ❖ for 반복문

```
for(var i = 0; i < 10; i++){  
    console.log("이러면 10번 반복하겠죠?" )  
}
```

## ❖ while 반복문

```
var i = 0  
while(i < 10){  
    console.log("이래도 10번 반복하겠죠?" )  
    i++  
}
```



# 기초 문법 실습

## ❖ 함수 선언과 호출

```
function 더하기(숫자1, 숫자2){  
    return 숫자1 + 숫자2  
}  
  
console.log(더하기(2,3))
```



# Node.js는 어떤 녀석일까?

## ❖ 비동기 이벤트 기반 Javascript Runtime

- Javascript를 웹 브라우저가 아닌 Server에서도 사용할 수 있도록 만든 **실행 환경**이다
  - Node.js는 언어가 아닌 JS를 실행하기 위한 프로그램(환경)이다
- Google의 **V8 Javascript 엔진 위에서 동작한다**
  - 기존에 존재했던 IE의 Chakra나 Firefox의 Spider Monkey 엔진보다 월등히 수행 능력이 좋은 엔진이다
- **확장 가능한(Scalable) 네트워크 애플리케이션**을 구축하도록 설계되었다
  - 내장 HTTP 서버 라이브러리를 포함하고 있어 Apache나 Nginx 같은 별도의 웹 서버없이 동작이 가능하다



# Node.js의 Event-Driven

## ❖ Node.js는 V8과 더불어 **libuv** 라이브러리를 사용한다

- libuv 라이브러리는 Node.js의 특성인 **Event-driven, Non Blocking I/O 모델**을 구현한다

## ❖ Event-driven

- 이벤트가 발생할 때, 미리 지정해둔 작업을 수행하는 방식을 의미한다
  - **이벤트 리스너**에 특정 이벤트가 발생할 때 무엇을 할지 **Callback 함수**에 등록한다
  - 이벤트가 발생하면 리스너에 등록해둔 Callback 함수를 호출하며, 이벤트가 완료된 후 Node.js는 다음 이벤트가 발생할 때 까지 대기한다

## ❖ Event loop

- 이벤트 루프는 여러 이벤트가 동시에 발생했을 때 **어떤 순서로 Callback 함수를 호출 할지 판단**한다
- Node.js는 이벤트가 종료될 때까지 **이벤트 처리를 위한 작업을 반복**하므로 Loop라고 한다





# Callback Function

- ❖ 다른 함수에 **인자(argument)**로 전달되어 그 **함수 내부에서 호출되는 함수**
  - 주로, 비동기적인 상황에서 자주 사용된다
    - 파일을 읽거나 네트워크 요청을 보내는 등의 **외부에서 작업이 필요한 상황**
    - 해당 작업이 완료되었을 때 호출할 콜백 함수를 지정하여 **작업이 완료될 때까지 대기하지 않고 다른 작업을 수행할 수 있다.**
- ❖ **setTimeout(callback, ms)**
  - ms(밀리초) 후에 callback 함수를 호출하는 내장 함수
  - 대표적인 **비동기 함수**
  - 예시)


```
function cb( ){  
    console.log( "미나상~ 오래기다렸습니다~" )  
}  
  
setTimeout(cb, 2000)
```

# Arrow Function

## ❖ ES6(2015)에서 도입된 JS의 새로운 함수 표현 방식

- 함수를 간략하게 표현할 수 있어 코드의 가독성을 높일 수 있다
- 주로 Callback 함수에 사용된다
- 항상 익명함수로 선언되며 호출하려면 변수에 할당하여 사용해야 한다

```
function cb(){  
  console.log("미나상~ 오래기다렸습니다~")  
}  
  
setTimeout(cb, 2000)
```



```
setTimeout(() => {  
  console.log("2초 후에 실행됨")  
}, 2000)
```

# Arrow Function

## ❖ 일반형

```
const 화살표함수 = (인자1, 인자2) => {  
  // 함수 본체  
}
```

## ❖ 인자가 한 개 일 때

```
const 화살표함수 = 인자1 => {  
  // 함수 본체  
}
```

## ❖ 본체의 표현식이 하나일 때 (반환값은 표현식의 값)

```
const 화살표함수 = 인자1 => 인자1 + 인자1
```



❖ 1+1, 2+2, 3+3의 결과를 차례대로 출력하는 코드를 작성하라

❖ 조건:

- 1+1의 값이 출력된 후 2초 후에 2+2의 결과가 출력되어야 한다
- 3+3의 값이 출력되는 시점은 2+2의 값이 출력된 직후이다



❖ 1+1, 2+2, 3+3의 결과를 차례대로 출력하는 코드를 작성하라

❖ 조건:

- 1+1의 값이 출력된 후 2초 후에 2+2의 결과가 출력되어야 한다
- 3+3의 값이 출력되는 시점은 2+2의 값이 출력된 직후이다

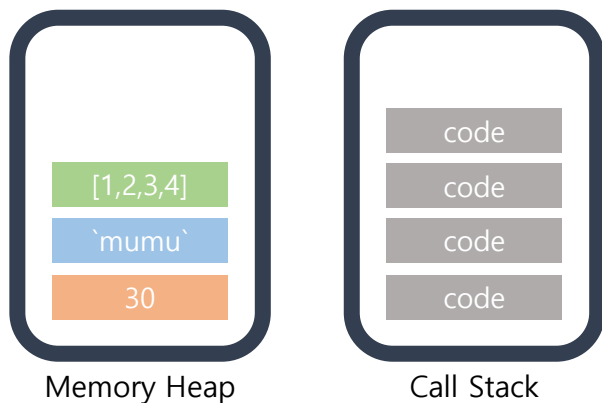
```
console.log(1+1)
setTimeout(() => {
  console.log(2+2)
  console.log(3+3)
}, 2000)
```



# Javascript Engine

## ❖ JS 엔진은 크게 **Memory Heap**과 **Call Stack**으로 이루어짐

- **Memory Heap:** 메모리 할당이 일어나는 메모리 영역
- **Call Stack:** 실행할 코드가 쌓이는 영역



## ❖ Call Stack은 JS내의 모든 코드를 수행해주지 않음

- Call Stack으로 들어온 **비동기 코드**(setTimeout, Ajax, etc)는 **외부 쓰레드에 의해** 수행하게 된다
- 브라우저 환경에서는 **Web API**
- Node.js 환경에서는 **libuv**

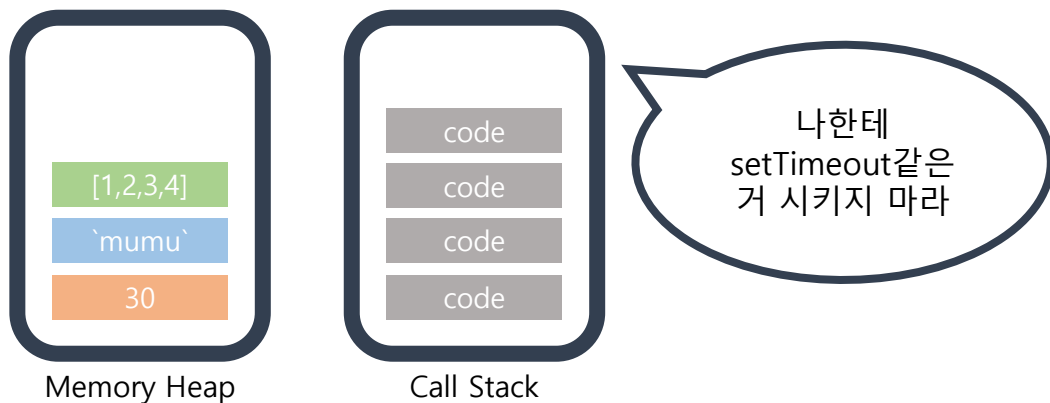
Stack: 후입선출(LIFO)로 마지막에 들어간 것이 먼저 나가는 자료구조



# Javascript Engine

## ❖ JS 엔진은 크게 **Memory Heap**과 **Call Stack**으로 이루어짐

- **Memory Heap:** 메모리 할당이 일어나는 메모리 영역
- **Call Stack:** 실행할 코드가 쌓이는 영역



## ❖ Call Stack은 JS내의 모든 코드를 수행해주지 않음

- Call Stack으로 들어온 **비동기 코드**(setTimeout, Ajax, etc)는 **외부 쓰레드에 의해** 수행하게 된다
- 브라우저 환경에서는 **Web API**
- Node.js 환경에서는 **libuv**

## ❖ C++로 작성된 Node.js가 사용하는 비동기 I/O 라이브러리

- 운영체제의 커널을 추상화한 Wrapping 라이브러리



- 쉽게 말해서 JS가 처리하지 못하는 동작을 운영체제에게 시키는 것
  - 생각보다 순수하게 JS가 처리할 수 있는 작업은 많이 없음
  - 네트워크, 타이머 등과 같이 라이브러리 없이 구현하기 어려운 동작들은 거의 다라고 생각하면 편함
- 작업이 끝난 비동기 함수의 Callback 함수는 libuv가 Node.js의 **Callback Queue**에 넘겨줌
  - 예를 들어, 타이머가 지난 setTimeout의 Callback 함수



# Callback Queue와 Event Loop

## ❖ Callback Queue

- 처리가 끝난 **비동기 함수의 Callback 함수가 모이는** 자료구조
- 이곳에 들어온 Callback들은 바로 실행되는 것이 아님
- Call Stack이 완전히 비워질 때까지 기다리는 **대기** 과정을 가짐

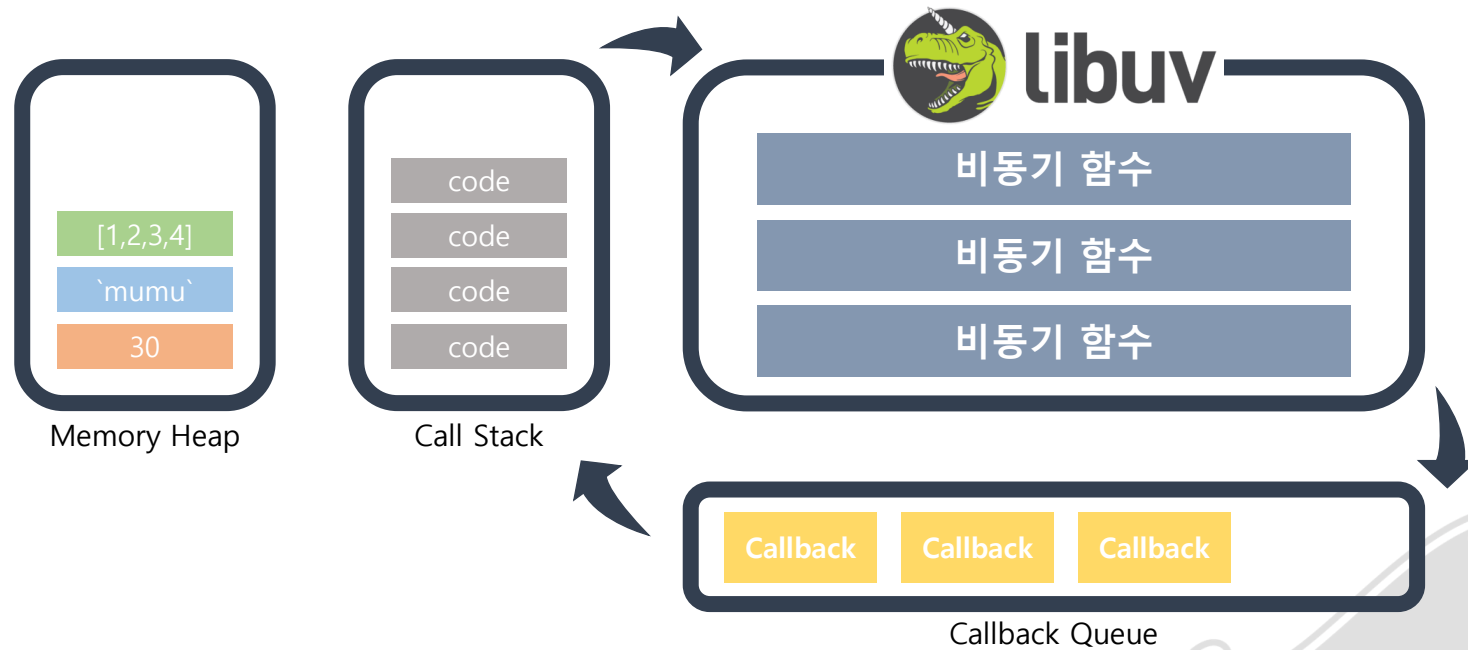
## ❖ Event Loop

- Node.js가 비동기 작업을 관리하기 위한 구현체로 비동기 작업들을 관리하는 역할
- JS가 실행되는 동안 Callback Queue와 Call Stack을 감시하며 **Call Stack이 비워지면 Callback Queue의 함수들을 Call Stack으로 넘겨주는 역할**을 한다

Queue: 선입선출(FIFO)로 먼저 들어온 것이 먼저 나가는 자료구조



# Javascript의 동작 원리



# 문제

❖ 다음 코드의 실행 결과를 예상해보세요

```
console.log("[1번째] console.log");  
setTimeout(() => {  
    console.log("[2번째] console.log");  
}, 0);  
console.log("[3번째] console.log");
```



# 문제

❖ 다음 코드의 실행 결과를 예상해보세요

```
console.log("[1번째] console.log");  
setTimeout(() => {  
    console.log("[2번째] console.log");  
}, 0);  
console.log("[3번째] console.log");
```

❖ 정답

```
[1번째] console.log  
[3번째] console.log  
[2번째] console.log
```



# Hoisting

## ❖ var 키워드

- 기본적인 변수 선언문이지만 **치명적인 약점**을 지니고 있음



# Hoisting

## ❖ var 키워드

- 기본적인 변수 선언문이지만 **치명적인 약점**을 지니고 있음
- 아래 코드의 결과를 예상해봅시다

```
console.log(나는변수짱)  
var 나는변수짱 = "나는변수짱"
```



# Hoisting

## ❖ var 키워드

- 기본적인 변수 선언문이지만 **치명적인 약점**을 지니고 있음
- 아래 코드의 결과를 예상해봅시다

```
console.log(나는변수짱)  
var 나는변수짱 = "나는변수짱"
```

- 결과

```
undefined
```



# Hoisting

## ❖ var 키워드

- 기본적인 변수 선언문이지만 **치명적인 약점**을 지니고 있음
- 아래 코드의 결과를 예상해봅시다

```
if(true){  
    var 나는변수짱 = "나는변수짱"  
}  
  
console.log(나는변수짱)
```





# Hoisting

## ❖ var 키워드

- 기본적인 변수 선언문이지만 **치명적인 약점**을 지니고 있음
- 아래 코드의 결과를 예상해봅시다

```
if(true){  
    var 나는변수짱 = "나는변수짱"  
}  
  
console.log(나는변수짱)
```

- 결과

나는변수짱



# Hoisting

## ❖ var 키워드

- 기본적인 변수 선언문이지만 **치명적인 약점**을 지니고 있음
- 방금의 문제 같이 변수 선언 전에 참조가 가능하며 값은 **undefined**로 초기화 되어 있음
- 전역변수와 지역변수의 개념이 모호하여 if Scope안에서 선언된 변수가 Scope 밖에서도 죽지 않음
  - 일반적인 Scope 개념과는 달리 var는 함수만을 Scope로 가짐

## ❖ 호이스팅

- 변수가 코드 등장 전에 미리 선언되는 현상

```
console.log(나는변수짱)  
var 나는변수짱 = "나는변수짱"
```



```
var 나는변수짱 = undefined  
console.log(나는변수짱)  
나는변수짱 = "나는변수짱"
```

# let과 const

## ❖ let 키워드

- var와 달리 호이스팅이 일어나지 않는 것처럼 동작하는 변수 선언문

```
console.log( 나는변수짱 )  
let 나는변수짱 = "나는변수짱"
```

**Error**

```
Process exited with code 1  
Uncaught ReferenceError: Cannot access '나는변수짱' before initialization  
    at <anonymous> (d:\algorithm\1517. BubbleSort\.js:1:13)  
    at Module._compile (internal/modules/cjs/loader:1159:14)  
    at Module._extensions..js (internal/modules/cjs/loader:1213:10)  
    at Module.load (internal/modules/cjs/loader:1037:32)  
    at Module._load (internal/modules/cjs/loader:878:12)  
    at executeUserEntryPoint (internal/modules/run_main:81:12)  
    at <anonymous> (internal/main/run_main_module:23:47)
```

# let과 const

## ❖ let 키워드

- var와 달리 **호이스팅이 일어나지 않는 것처럼 동작하는** 변수 선언문
- 실제로 호이스팅이 일어나지 않는 것은 아니다
- 코드가 등장하기 전까지 let 변수는 **일시적으로 죽은 공간**(Temporal Dead Zone, TDZ)에 위치한다
- TDZ에 위치한 변수들은 **참조가 절대 불가능**하다



# let과 const

## ❖ let 키워드

- var와 달리 **호이스팅이 일어나지 않는 것처럼 동작**하는 변수 선언문
- 실제로 호이스팅이 일어나지 않는 것은 아니다
- 코드가 등장하기 전까지 let 변수는 **일시적으로 죽은 공간**(Temporal Dead Zone, TDZ)에 위치한다
- TDZ에 위치한 변수들은 **참조가 절대 불가능**하다

## ❖ const 키워드

- let과 같이 TDZ를 지원하는 변수 선언문
- 상수를 선언할 때 사용하며, 선언과 초기화를 동시에 진행해야한다
  - Why? 상수라서 값을 재지정할 수 없기 때문
- 보통은 객체를 저장하는 변수에 사용한다



# 감사합니다

---

