Node.js

-

01

Node.js 란

들어가기 전에

01 Node.js

Node.js

JavaScript는 객체 기반의 **스크립트 프로그래밍 언어**이다.

자바스크립트는 본래 **동적인 DOM**(애니메이션, HTML 핸들링)을 위해 넷스케이프 커뮤니케이션즈 코퍼레이션의 브렌던 아이크(Brendan Eich)가 개발하였다.

당연하게도 이 언어는 **웹 브라우저 내에서 주로 사용**한다.
하지만 최근 **Node.js와 같은 런타임 환경과 같이 서버 프로그래밍(범용적으로)에도 사용되고 있다.**

자바스크립트가 **썬 마이크로시스템즈의 자바와 구문이 유사한 점도 있지만, 이는 사실 두 언어 모두 C 언어의 기본 구문에 바탕을 뒀기 때문이고, 자바와 자바스크립트는 직접적인 관련성이 없다.**

자바스크립트는 브라우저마다 지원되는 버전이 달랐기 때문에 **크로스 브라우징 문제**가 발생했었지만, 최근 ECMA2015 이후부터 어느정도 **표준이 정해진 상태**이다. 그럼에도 불구하고 IE의 옛버전에서는 호환성 문제가 발생하니 주의 해야 한다.

01 Node.js

Node.js®는 Chrome V8 JavaScript 엔진으로 빌드된 JavaScript 런타임입니다. Node.js는 이벤트 기반, 논 블로킹 I/O 모델을 사용해 가볍고 효율적입니다. Node.js의 패키지 생태계인 npm은 세계에서 가장 큰 오픈 소스 라이브러리 생태계이기도 합니다.

Ryan Dahl: Node.js의 개발자 (2010년)

01 Node.js

런타임이란

특정 언어로 만든 프로그램들을 실행할 수 있는 환경.

자바스크립트 런타임 = **자바스크립트 실행기**

자바스크립트 개발 목적: 웹 브라우저 위에서 동작하기 위한 언어

- ☐ 브라우저 외에서도 자바스크립트를 실행하고 싶다.
- ☐ 속도가 너무 느려서 대부분 실패
- ☐ 구글의 V8엔진 사용한 크롬으로 속도가 빨라짐
- ☐ 라이언 달이 개발

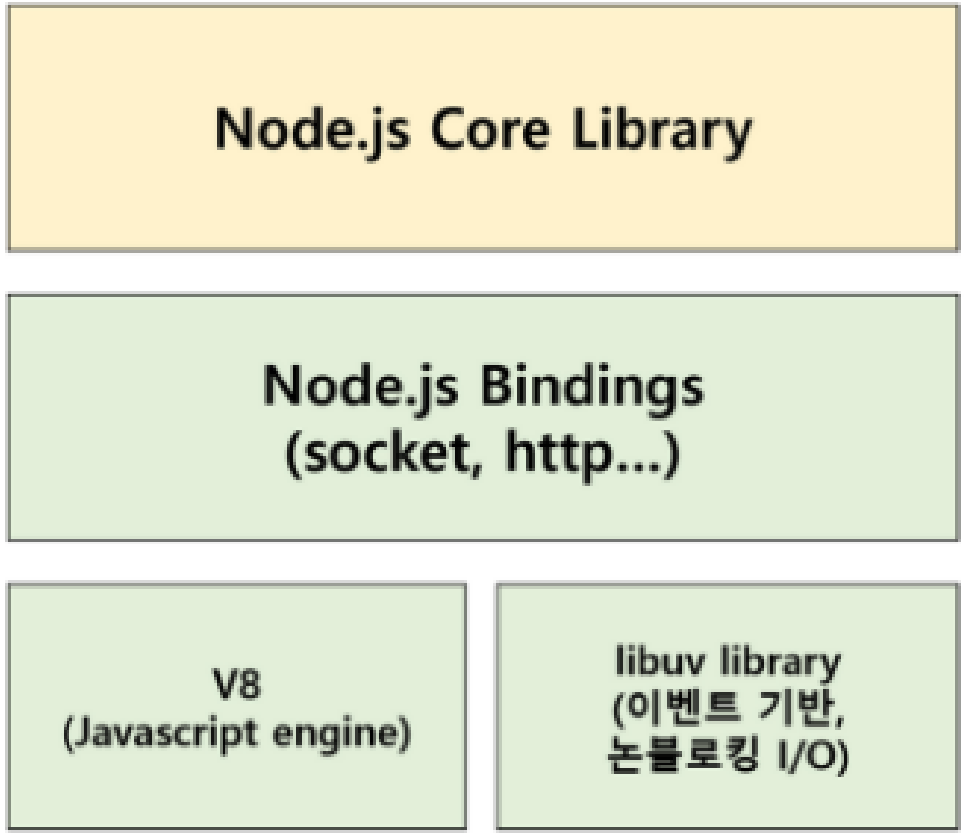
01 Node.js

Node.js = Javascript Runtime 환경
= 자바스크립트를 실행하기 위한 환경
= JS를 브라우저 없이 실행시킬 수 있는 환경

NPM = (Node Package Manager[module])
= JS 오픈소스 라이브러리 생태계

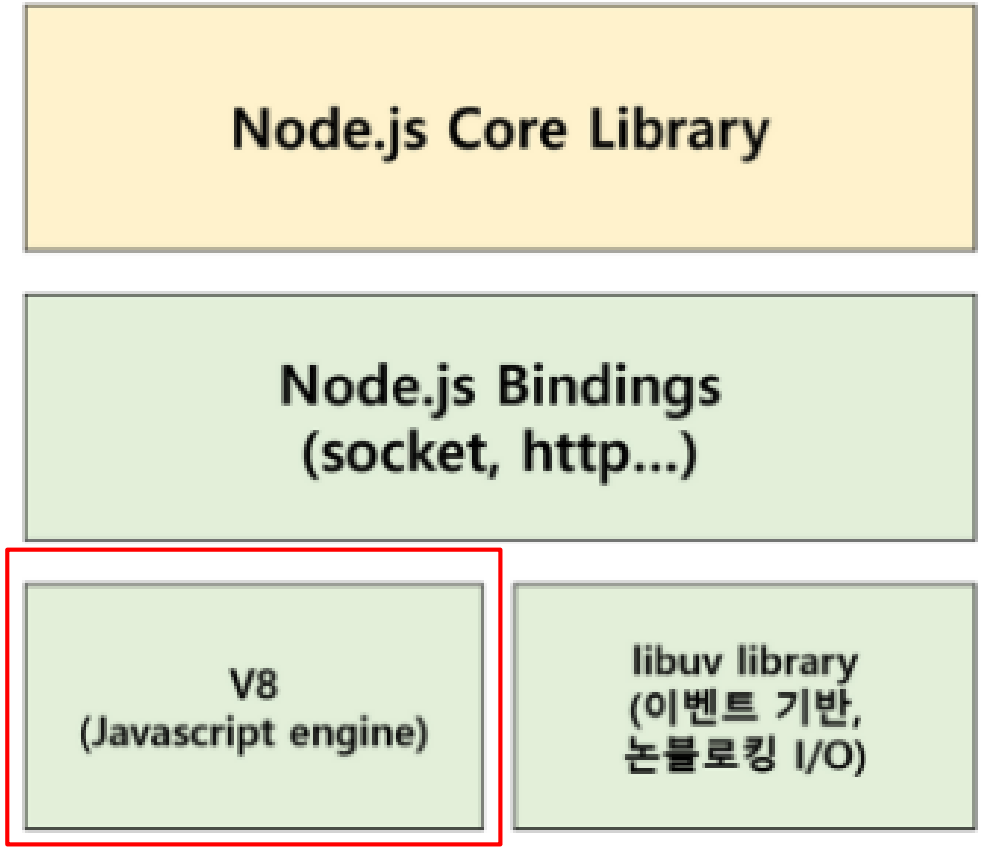
01 Node.js

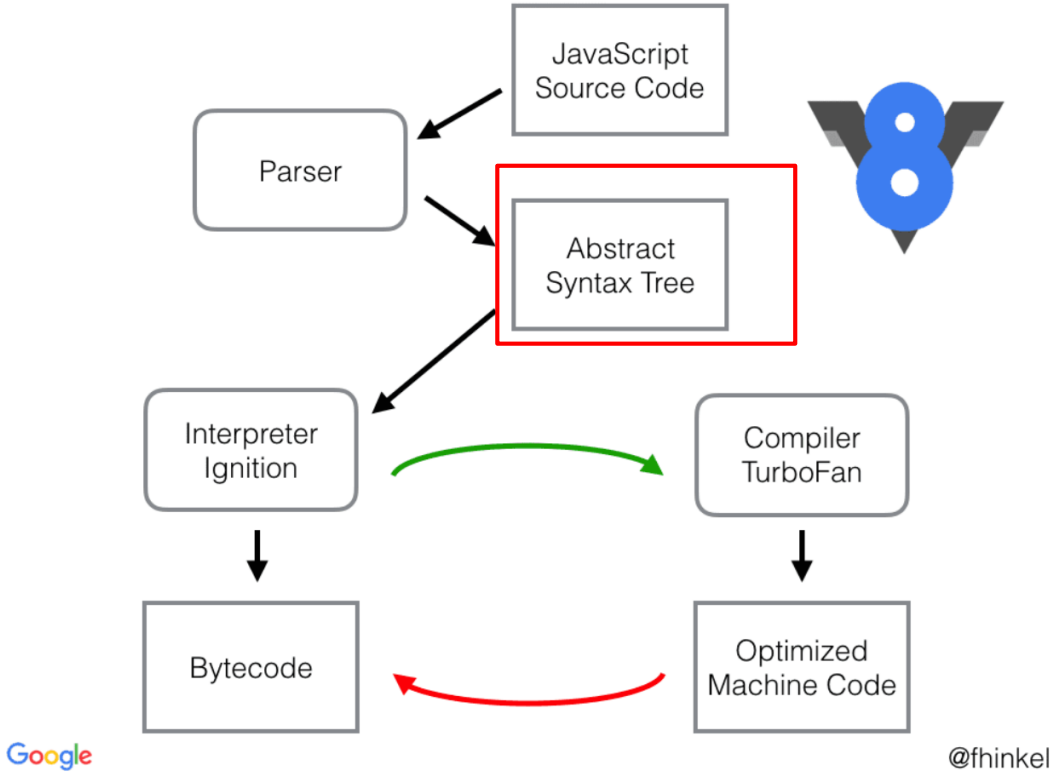
노드의 내부 구조



01 Node.js

노드의 내부 구조





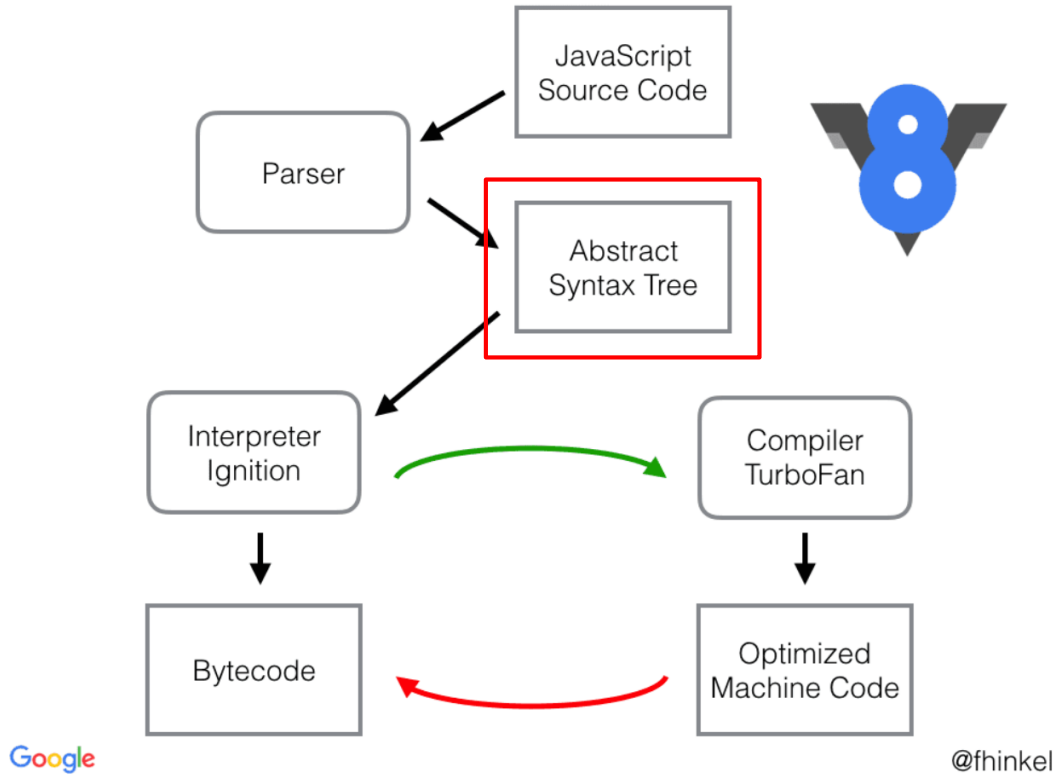
JSConf EU 2017에서 발표한 Franziska Hinkelmann님의 자료

https://astexplorer.net/

Note

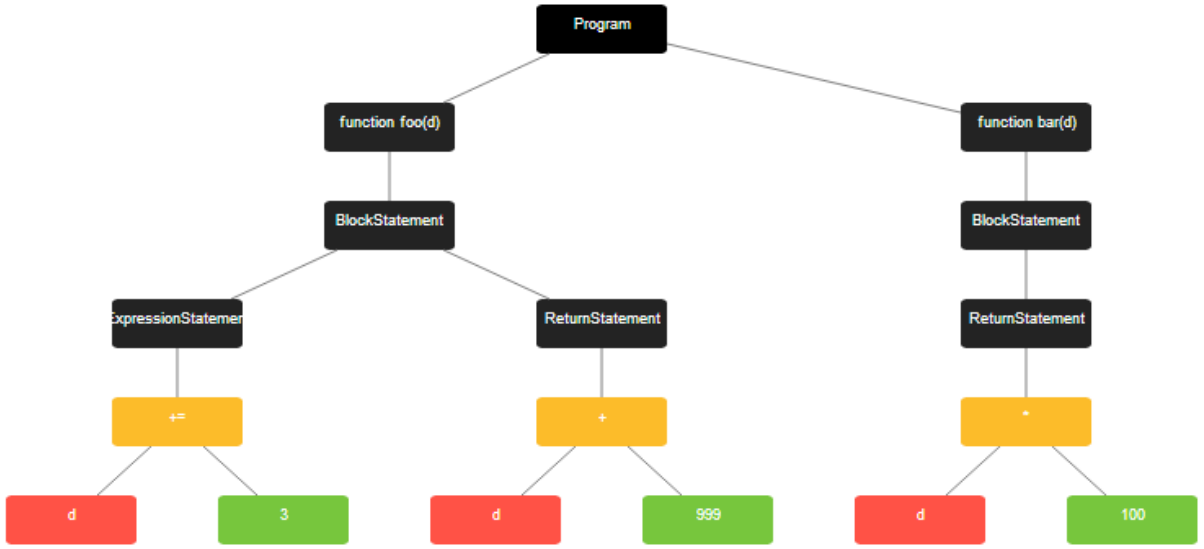
01 Node.js

v8엔진 동작 방식



JSConf EU 2017에서 발표한 Franziska Hinkelmann님의 자료

```
function foo(d) {
  d += 3;
    return d+999
}
function bar(d) {
    return d*100
}
```

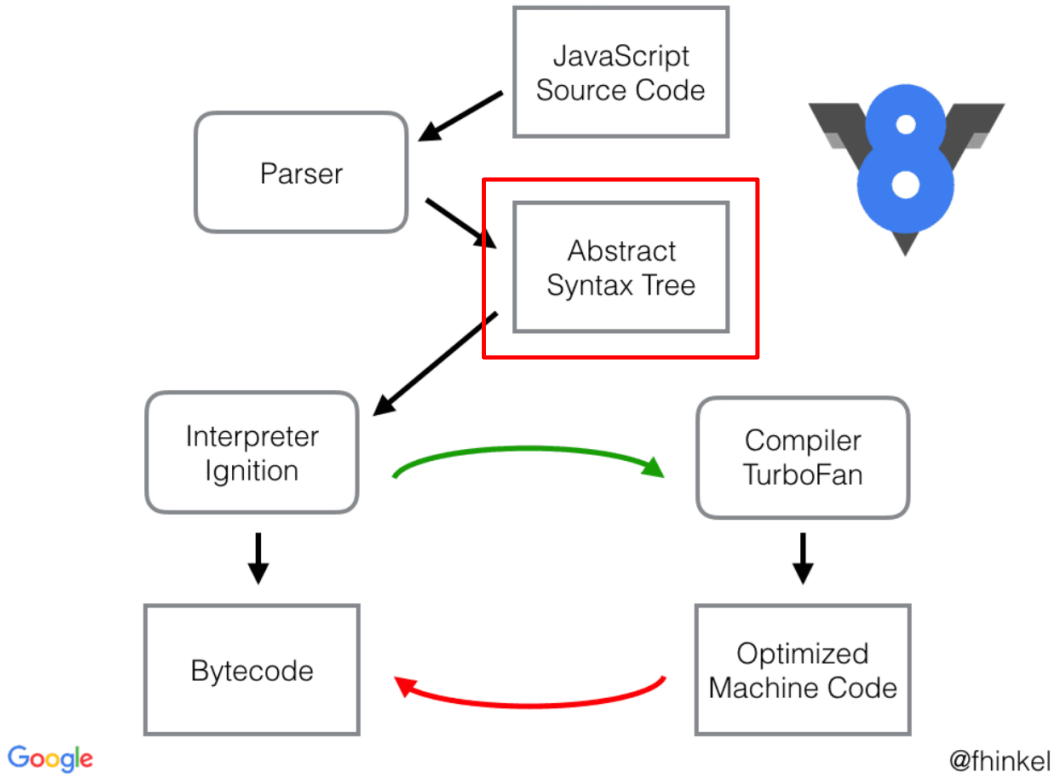


Betweak's Note

https://astexplorer.net/

01 Node.js

v8엔진 동작 방식



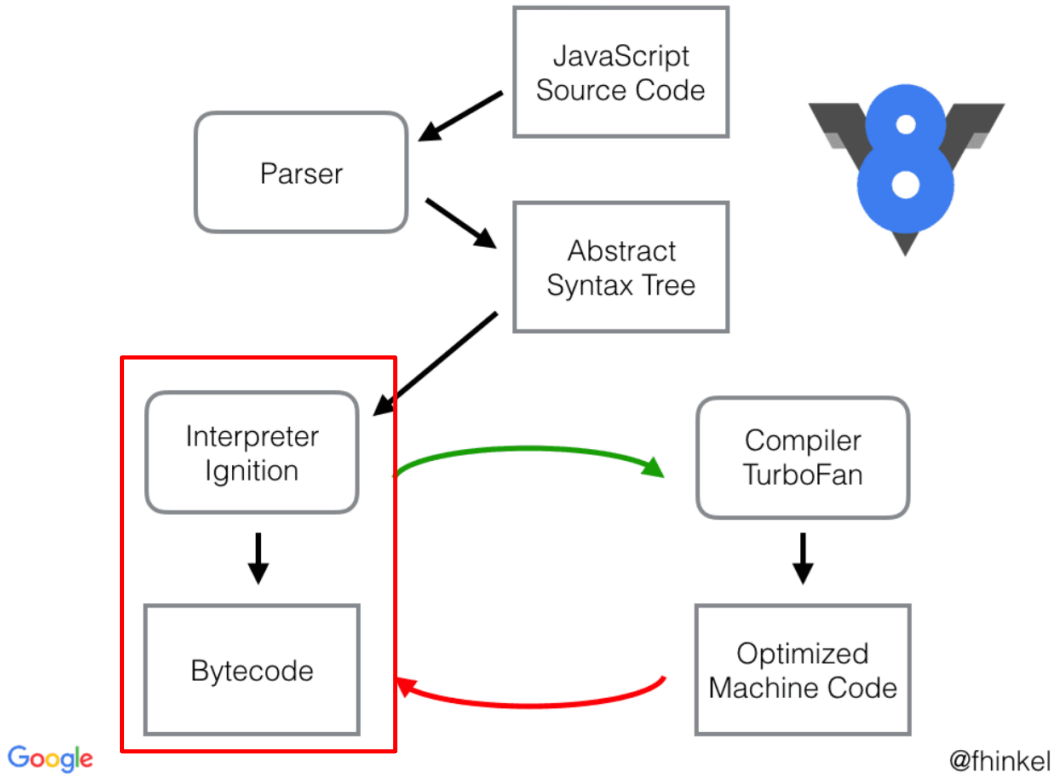
JSConf EU 2017에서 발표한 Franziska Hinkelmann님의 자료

```
- Program {
    type: "Program"
    start: 0
    end: 516
  - body: [
    + VariableDeclaration {type, start, end, declarations, kind}
    + FunctionDeclaration {type, start, end, id, expression, ... +4}
    - FunctionDeclaration {
        type: "FunctionDeclaration"
        start: 481
        end: 516
      + id: Identifier {type, start, end, name}
        expression: false
        generator: false
        async: false
        params: [ ]
      - body: BlockStatement {
          type: "BlockStatement"
          start: 498
          end: 516
        - body: [
          - ReturnStatement {
              type: "ReturnStatement"
              start: 502
              end: 514
            + argument: Literal {type, start, end, value, raw}
            }
          ]
        }
      }
  ]
    sourceType: "module"
}
```

https://astexplorer.net/

01 Node.js

v8엔진 동작 방식



JSConf EU 2017에서 발표한 Franziska Hinkelmann님의 자료

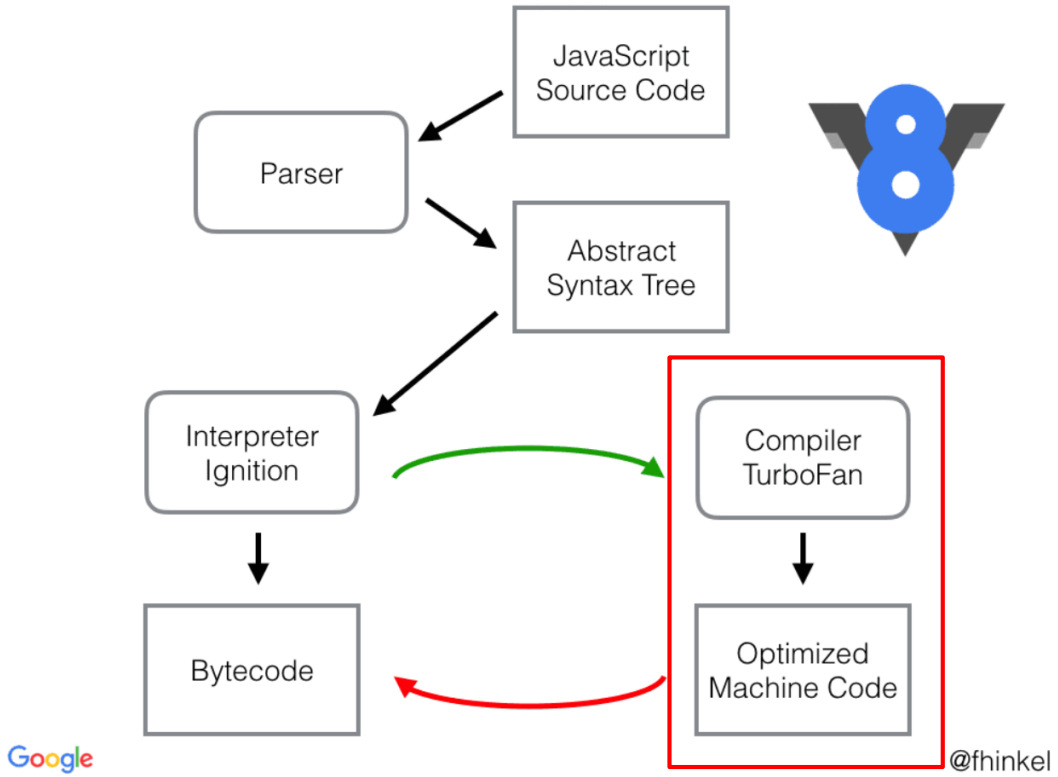
Ignition

JS 코드를 바이트 코드(ByteCode)로 변환하는 인터프리터
원본 소스 코드보다 컴퓨터가 해석하기 쉬운 바이트 코드로 변환
☐ 수시로 코드를 파싱(Parsing)하는 작업을 최소화
☐ 코드의 양 작아짐
☐ 메모리 공간 효율적 관리

바이트 코드로 변환할 경우 장점
- 파싱시 오버헤드 감소(js에 비해 파싱이 편함)
- 최적화 편리함 (동적 타이핑의 한계로 인한 최적화 과정에서)

01 Node.js

v8엔진 동작 방식



JSConf EU 2017에서 발표한 Franziska Hinkelmann님의 자료

TurboFan

JS 최적화 기법 □ 핫한 function 을 캐싱해두자.

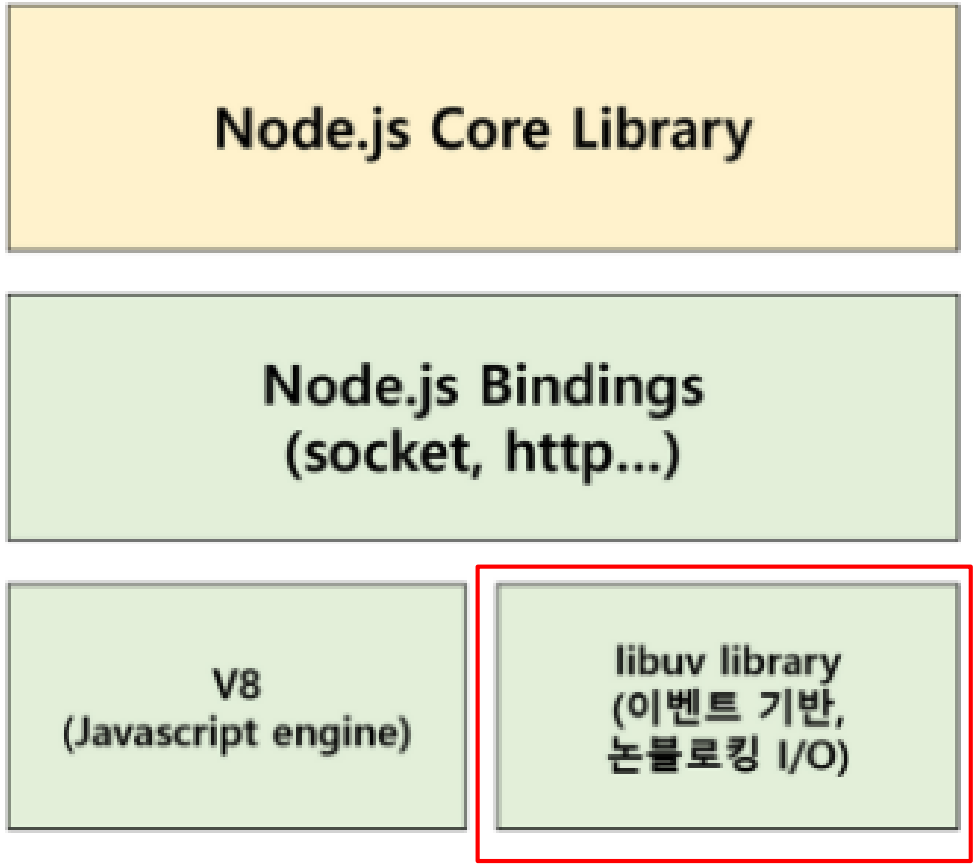
V8은 런타임 중에 Runtime Profiler를 통해 함수나 변수들의 호출 빈도 및 데이터 속성을 모으라고 시킴.

□ 해당 내용을 **캐싱**하여 바이트 코드에 반영

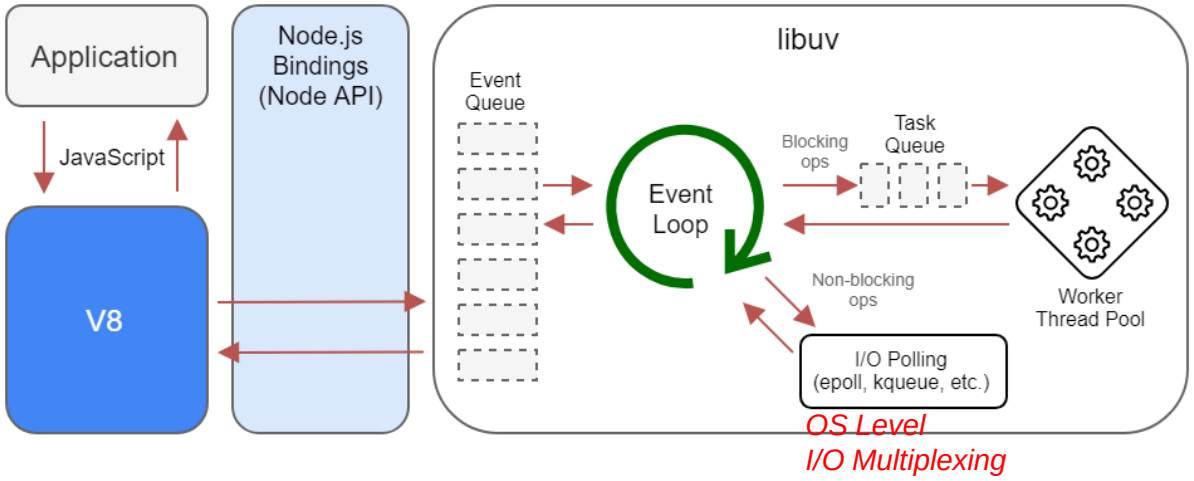
□ 자주 사용하는 형태로 구성하도록 바이트 코드 최적화
(함수나 객체의 메모리 주소값 조회 최소화)

01 Node.js

노드의 내부 구조



Node.js application은 단일 스레드/이벤트 기반 모델



Application 자체: Event Queue + Event LOOP □ 단일 스레드
BackGround 에서는 비동기 스레드 풀이 존재
OS 비동기 Interface 우선 사용 + 비동기 스레드 풀

01 Node.js

노드의 내부 구조

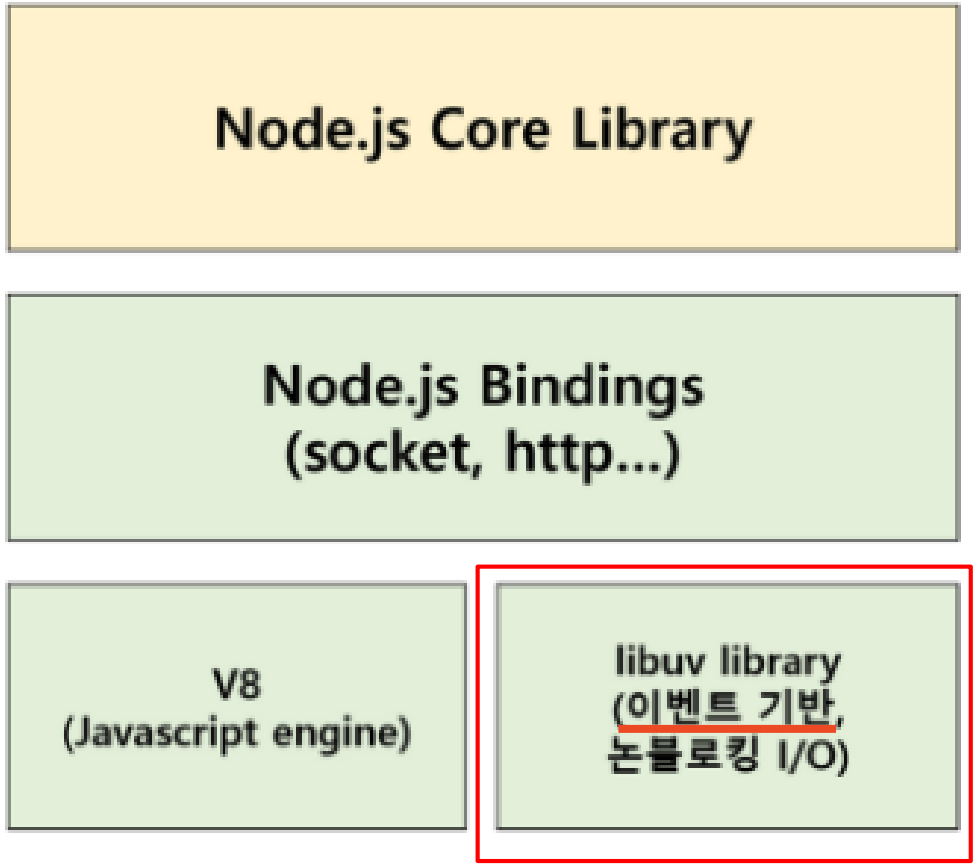
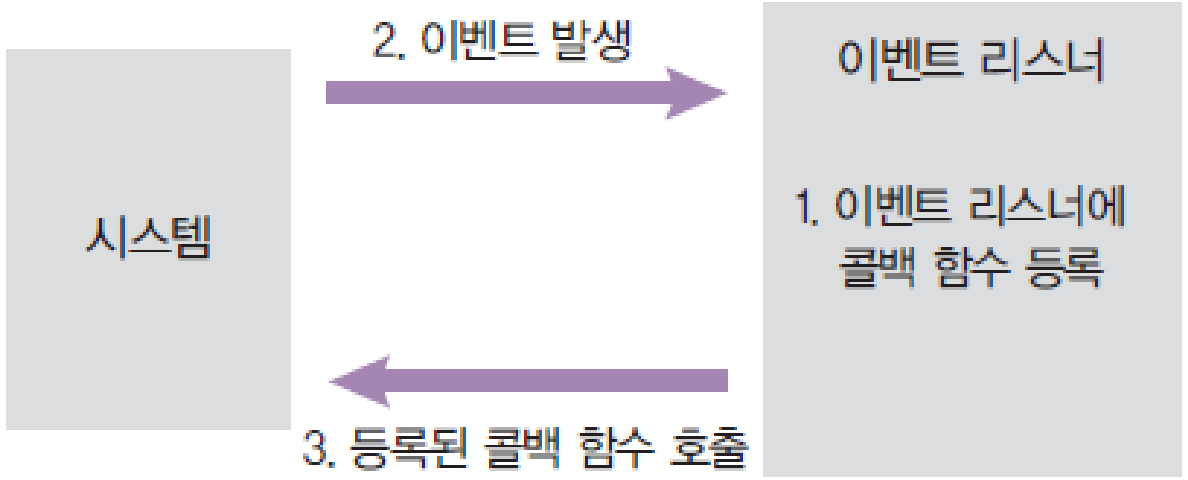


그림 1-4 이벤트 기반

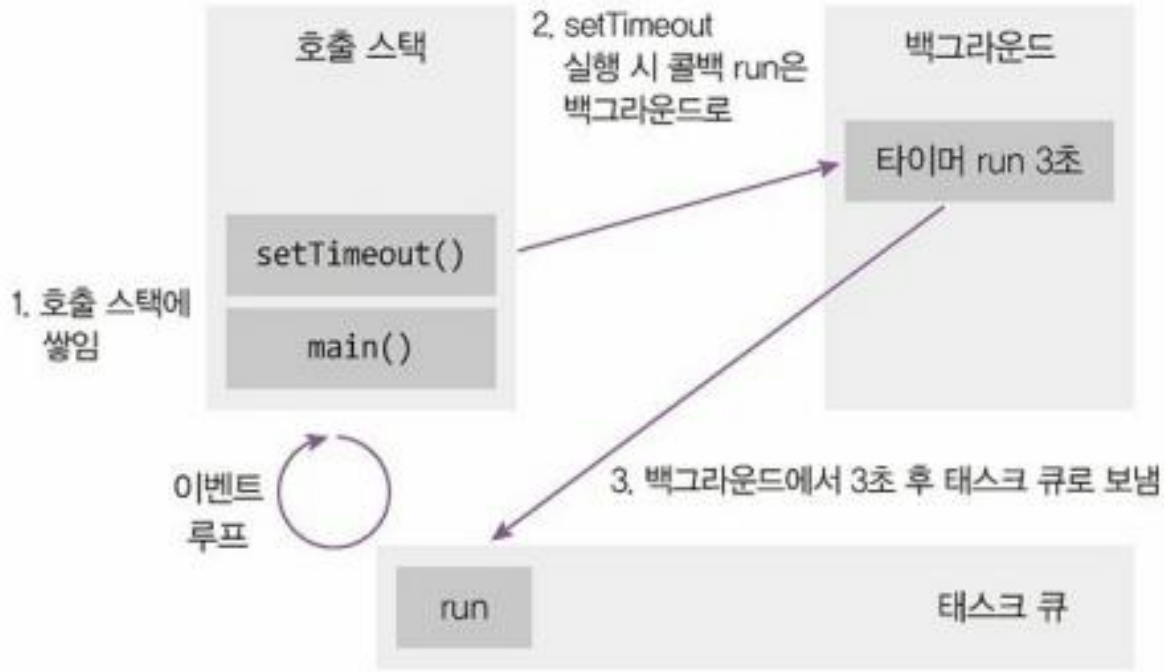


01 Node.js

이벤트 기반?

```
function run() {
  console.log('3초 후 실행');
}
console.log('시작');
setTimeout(run, 3000);
console.log('끝');
```

예상 결과는?



▲ 그림 1-6 이벤트 루프 1

01 Node.js

이벤트 기반?

```
function run() {
  console.log('3초 후 실행');
}
console.log('시작');
setTimeout(run, 3000);
console.log('끝');
```

예상 결과는?

01 Node.js

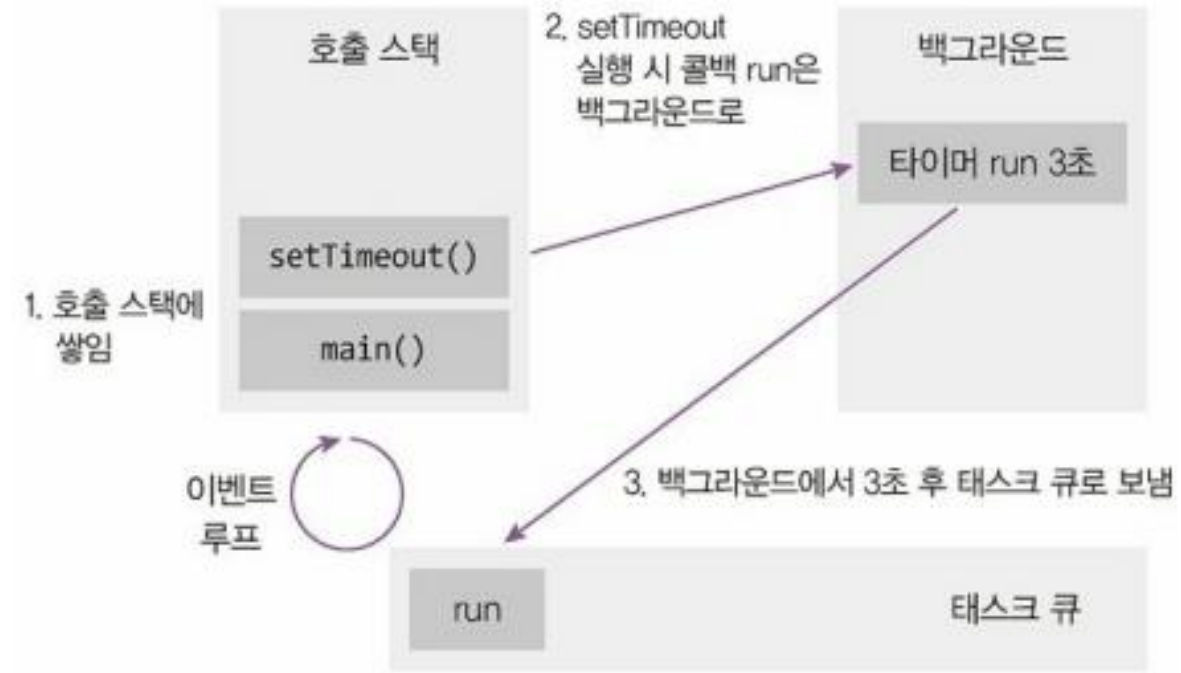
이벤트 기반?

```
function run() {
  console.log('3초 후 실행');
}
console.log('시작');
setTimeout(run, 3000);
console.log('끝');
```

예상 결과는?

콘솔

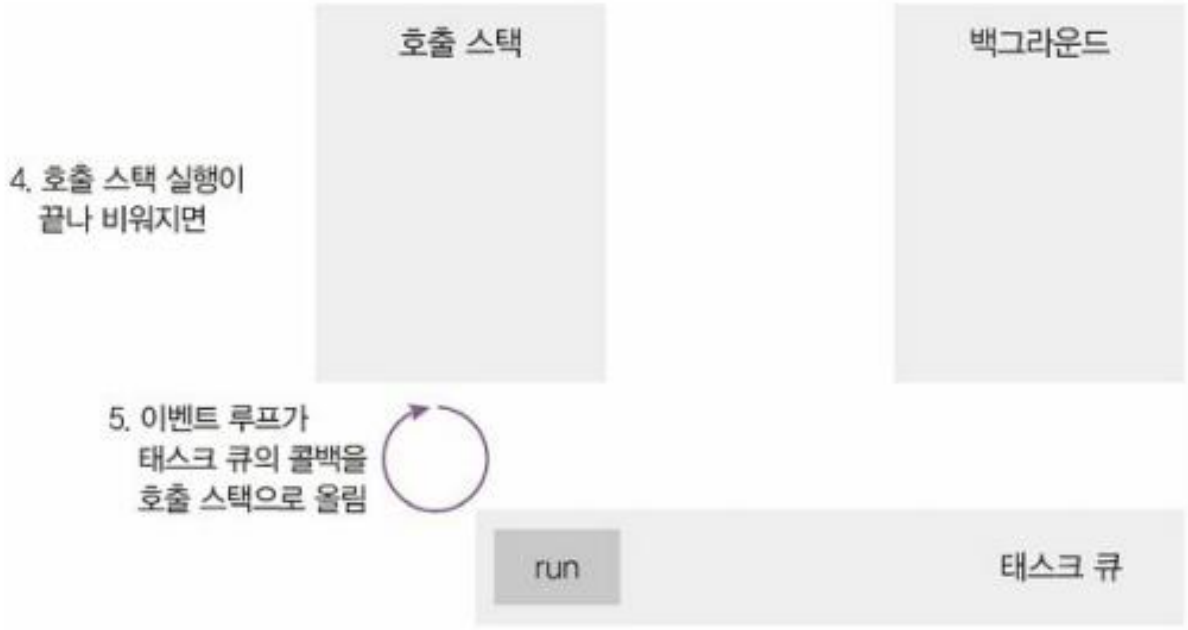
```
시작
끝
3초 후 실행
```



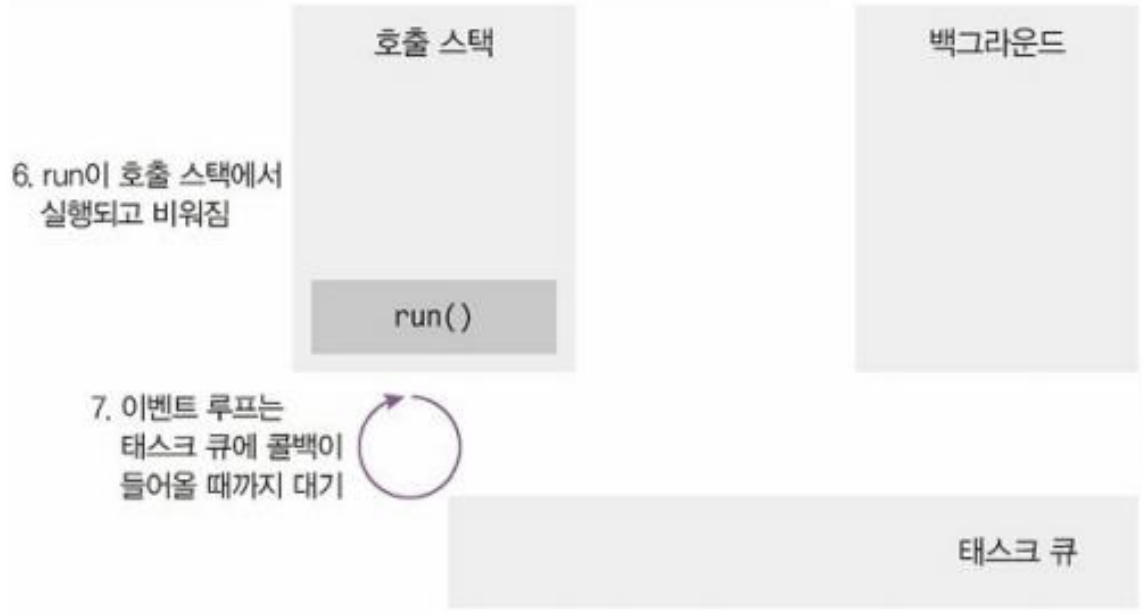
▲ 그림 1-6 이벤트 루프 1

01 Node.js

이벤트 기반?



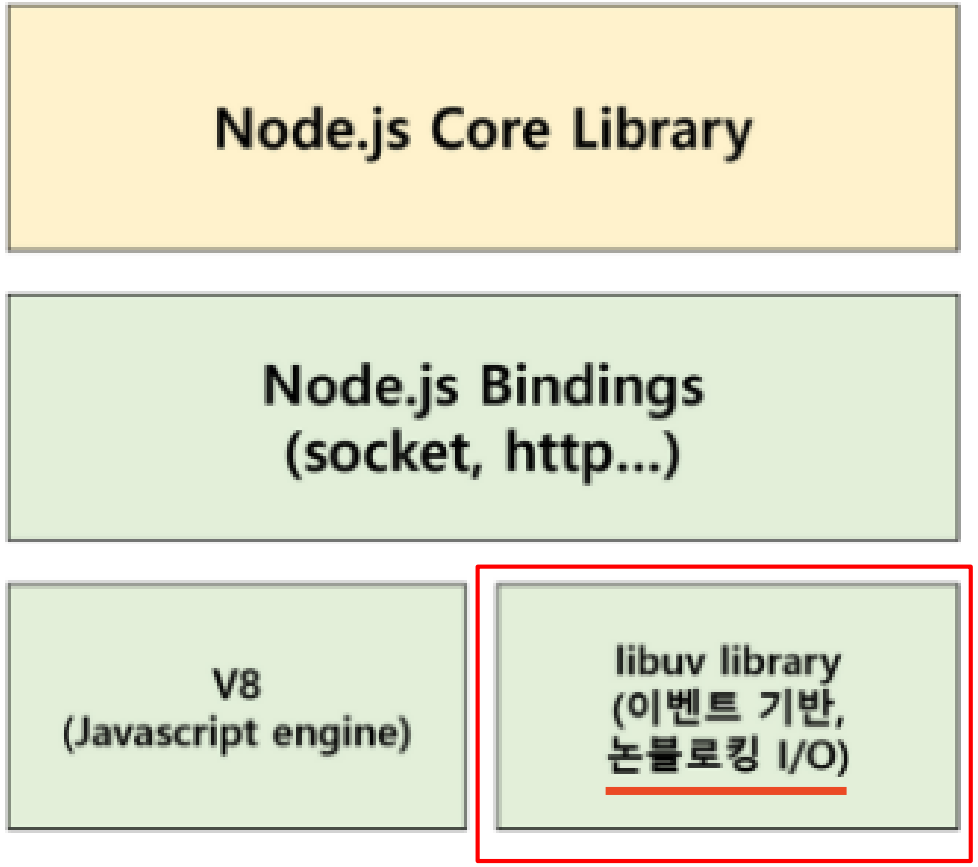
▲ 그림 1-7 이벤트 루프 2

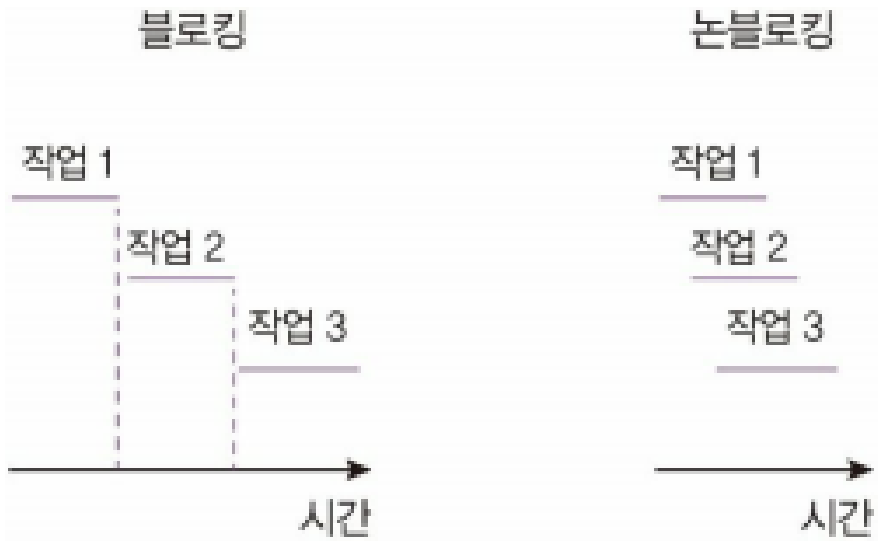


▲ 그림 1-8 이벤트 루프 3

01 Node.js

노드의 내부 구조





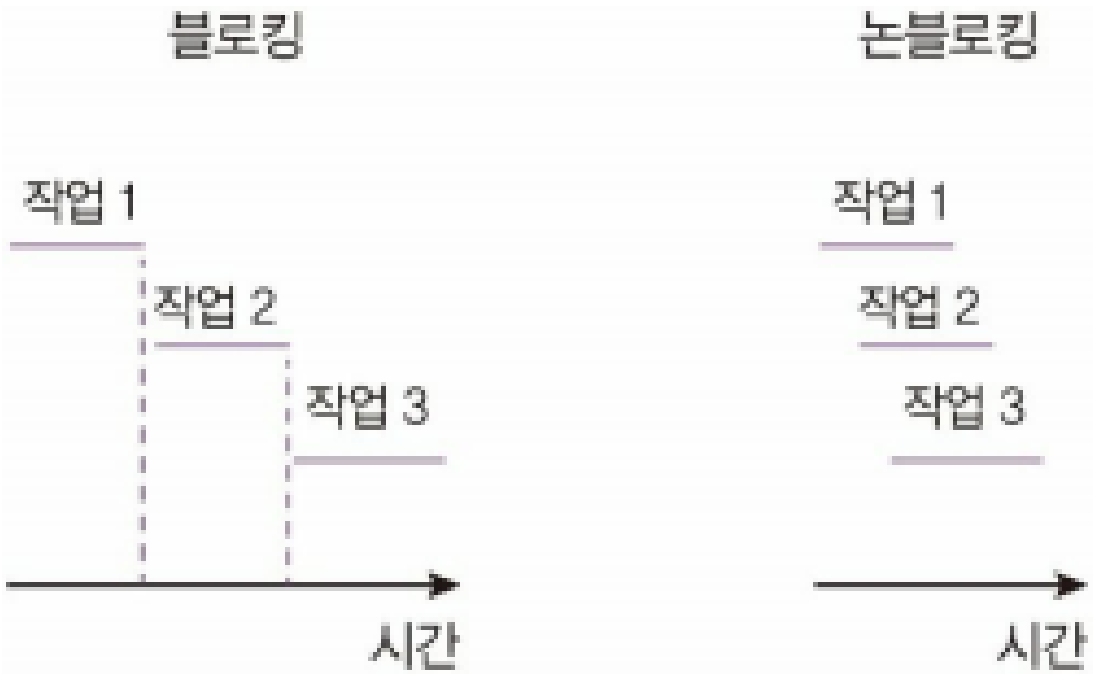
▲ 그림 1-9 블로킹과 논블로킹

CPU는 프로세스별로 점유!
□ 그럼 CPU를 제외하고 오래 걸리는 작업들은?
□ I/O (입출력) □ I/O는 Network I/O를 포함

01 Node.js

Blocking vs Non Blocking

직접 제어할 수 없는 대상을 처리하는 방식 (I/O, 쓰레드 동기화)



▲ 그림 1-9 블로킹과 논블로킹

CPU는 프로세스별로 점유!
□ 그럼 CPU를 제외하고 오래 걸리는 작업들은?
□ I/O (입출력) □ I/O는 Network I/O를 포함

I/O 관련 코드가 삽입되어 있으면,
코드 진행은 Blocking
□ I/O가 완료될 때까지 다음 task가 실행되지 않음.
□ CPU자원의 효율성 떨어짐.

□ **코드 진행을 Non-Blocking하게 진행하자!**

01 Node.js

Blocking vs Non Blocking

직접 제어할 수 없는 대상을 처리하는 방식 (I/O, 쓰레드 동기화)

```
// 오래 걸리는 작업이라고 가정
function longRunningTask() {
  // 오래 걸리는 작업
  console.log('작업 끝');
}

console.log('시작');
longRunningTask();
console.log('다음 작업');
```


예상 결과는?

01 Node.js

Blocking vs Non Blocking

직접 제어할 수 없는 대상을 처리하는 방식 (I/O, 쓰레드 동기화)

```
// 오래 걸리는 작업이라고 가정
function longRunningTask() {
  // 오래 걸리는 작업
  console.log('작업 끝');
}

console.log('시작');
longRunningTask();
console.log('다음 작업');
```

콘솔

```
시작
작업 끝
다음 작업
```

- ☐ longRunningTask가 모두 완료될 때까지 다음 작업을 실행하지 못함
- ☐ Blocking 되어 있는 동안 다른 작업을 할 수 없으므로, CPU자원 낭비
- ☐ 수행 시간이 오래걸리게 됨

오래걸리는 I/O 작업을 백그라운드로 보내자!

예상 결과는?

01 Node.js

Blocking vs Non Blocking

직접 제어할 수 없는 대상을 처리하는 방식 (I/O, 쓰레드 동기화)

```
1  function longRunningTask() {
2    // 오래 걸리는 작업
3    console.log("오래 걸리는 작업 끝");
4  }
5  console.log("시작");
6  setImmediate(longRunningTask);
7  console.log("종료")
```

예상 결과는?

01 Node.js

Blocking vs Non Blocking

직접 제어할 수 없는 대상을 처리하는 방식 (I/O, 쓰레드 동기화)

Blocking 방식의 예

```
const fs = require('fs');
const data = fs.readFileSync('/file.md'); // blocks here until file is read
console.log(data);
moreWork(); // will run after console.log
```

fs.readFileSync() 파일을 동기적으로 읽는다.

- **fs.readFileSync함수는 파일의 모든 데이터를 읽을 때까지 값을 Return(반환)하지 않음**
- **코드진행이 Blocking 됨**

01 Node.js

Blocking vs Non Blocking

직접 제어할 수 없는 대상을 처리하는 방식 (I/O, 쓰레드 동기화)

NonBlocking의

```
1  const fs = require('fs');
2  const promise = fs.readFile('./sample.txt', (err, data) => {
3    if (err) throw err;
4    console.log("data읽기 완료");
5  });
6  console.log("promise: ", promise);
7  console.log("작업완료.")
```

fs.readFile() 함수는 파일을 비동기적으로 읽는다.

- fs.readFile함수는 파일을 읽는 것을 기다리지 않고 바로 값을 반환함. (undefined)
- 다음 코드를 바로 실행함. (console.log("작업완료"); 코드진행이 Non-Blocking 됨

01 Node.js

동기(Synchronous) vs 비동기(Asyncronous)

Blocking과 Non-Blocking을 설명하다가 동기적, 비동기적 이라는 표현이 나왔습니다.

동기(Synchronous)
□ 동기방식은 코드 진행 영역에서 함수를 호출하고 호출된 함수의 **작업의 실행 상태를 계속 신경을 씀.**

비동기(Asynchronous)
□ 비동기방식은 코드 진행 영역에서 함수를 호출하고 호출된 함수의 **작업의 실행 상태를 신경쓰지 않음**

01 Node.js

(Blocking vs Non-Blocking) && (Synchrouns vs Asynchrnous)

Blocking vs Non-Blocking

□ 코드 진행이 멈추는지 여부에 따라 결정

Syncrhonous vs Asynchronous

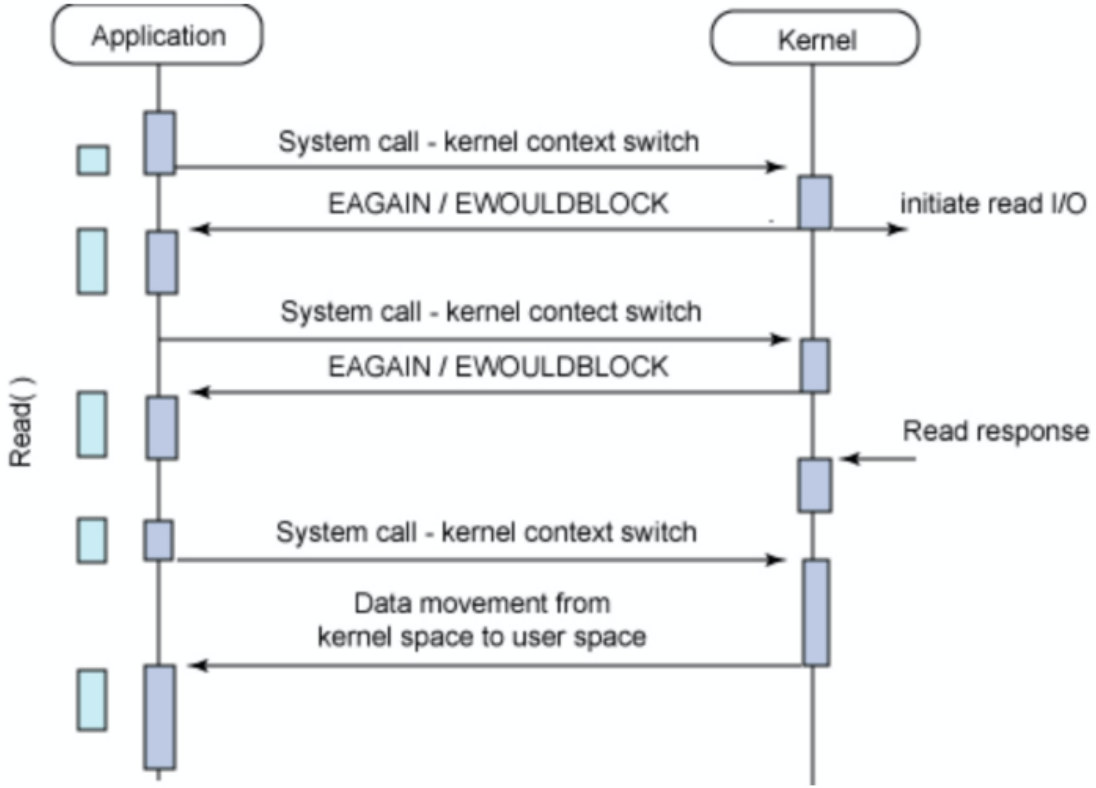
□ 호출하는 함수가 호출되는 함수의 작업 완료를 신경쓰는지에 대한 여부

일반적으로
(Syncrhonous – Blocking)
(Non-Blocking - Asyncrhonous)
이 함께 사용되는 경우가 일반적으로 많을 수 밖에 없다.

□ 코드진행이 멈춘다는 건 실행 상태를 안다는 것과 마찬가지이고,
□ 함수의 함수의 작업 완료를 신경쓰지 않는다는 건 코드가 실행 상태를 몰라도 되도록 Non-Blocking하게 진행할 것이기 때문!

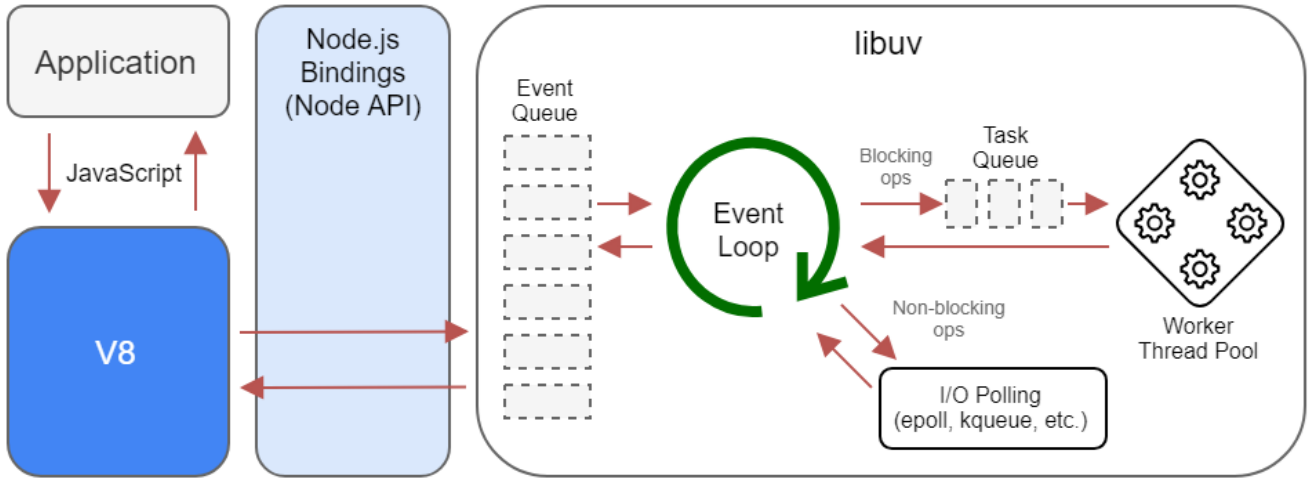
01 Node.js

Synchronous Non-Blocking I/O



01 Node.js

싱글 스레드 기반



프로세스: 운영체제로부터 자원을 할당받은 **작업**의 단위.
스레드: 프로세스가 할당받은 자원을 이용하는 **실행 흐름**의 단위.

Event Loop가 싱글쓰레드!

EventLoop는 싱글쓰레드이기 때문에 Blocking 되어선 안된다.
□ 비용이 비싼 작업들(CPU, I/O)은 비동기로 처리하자.

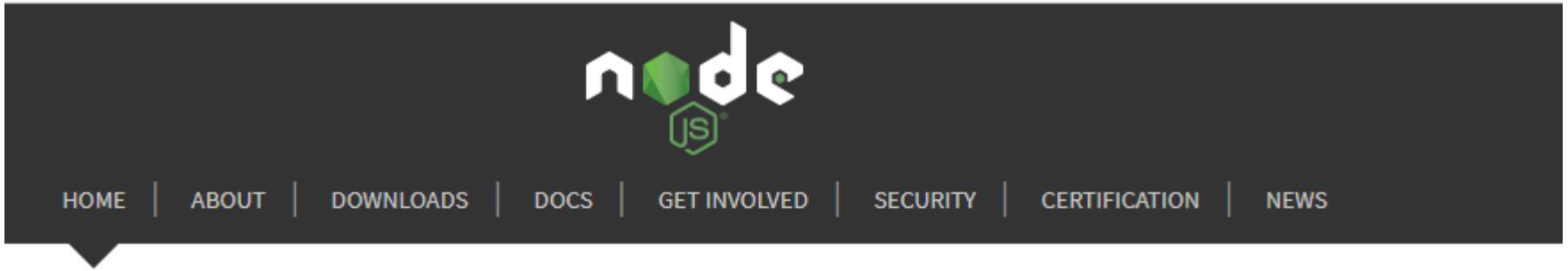
01 Node.js

참고자료

- https://www.youtube.com/watch?v=8aGhZQkoFbQ&t=492s
- https://nodejs.org/en/docs/guides/
- https://nodejs.dev/learn/introduction-to-nodejs

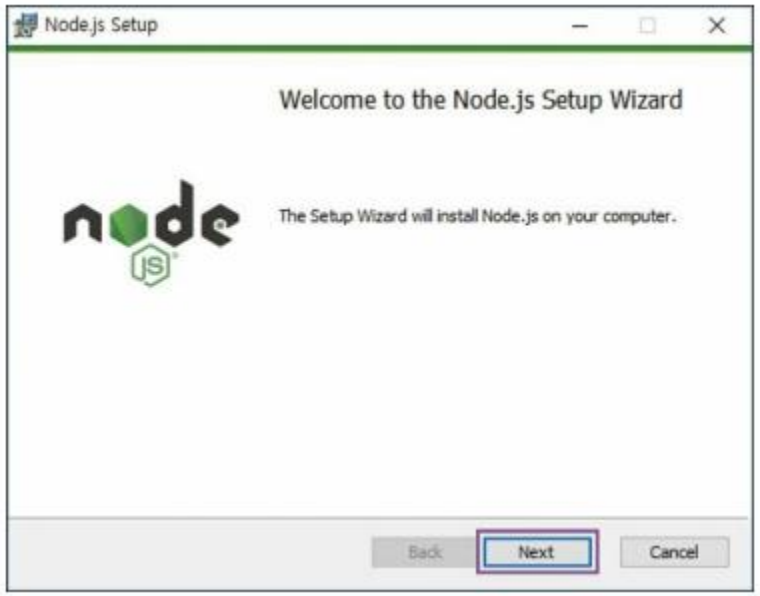
02 개발환경 설정

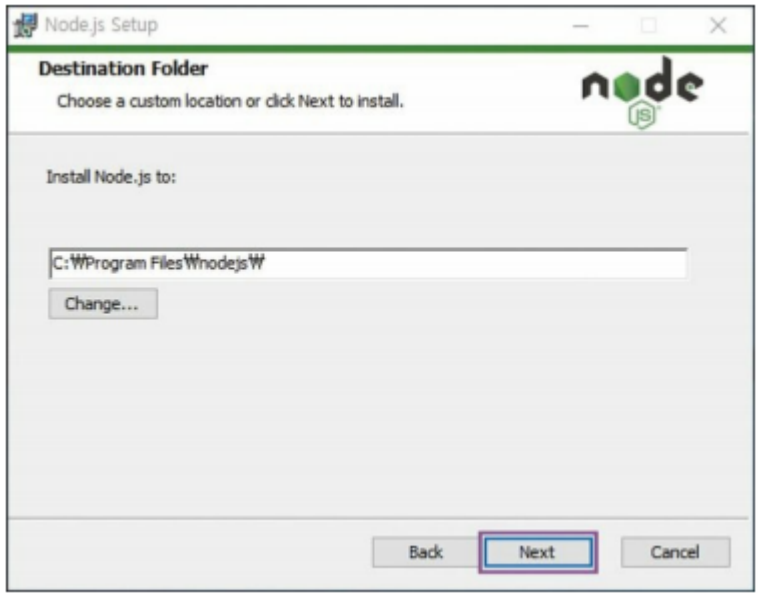
[Windows]



02 개발환경 설정

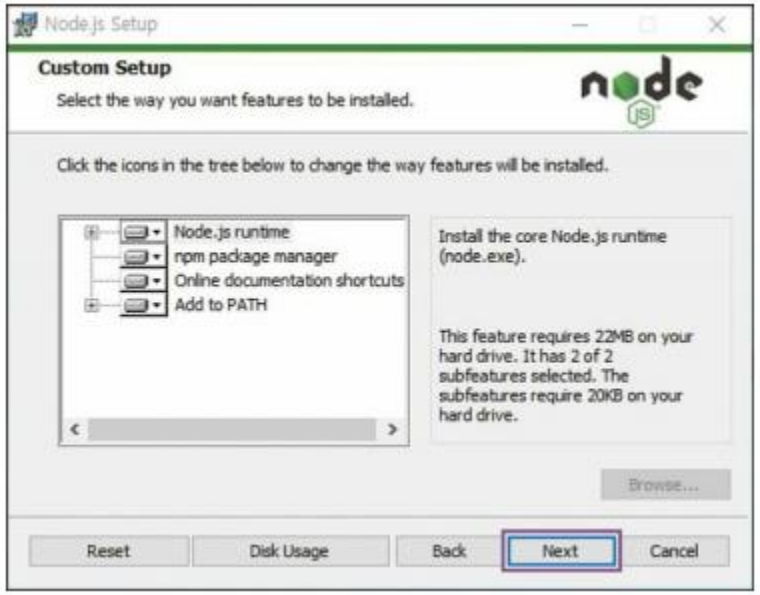
[Windows]

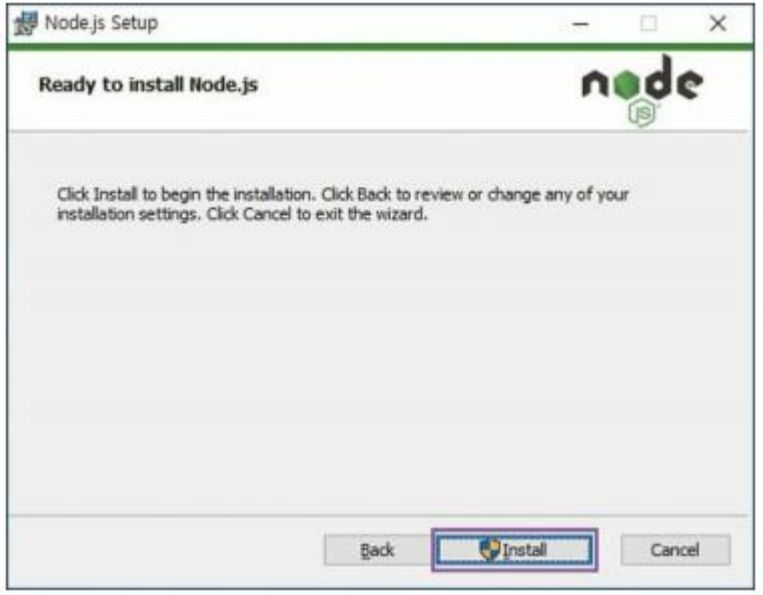


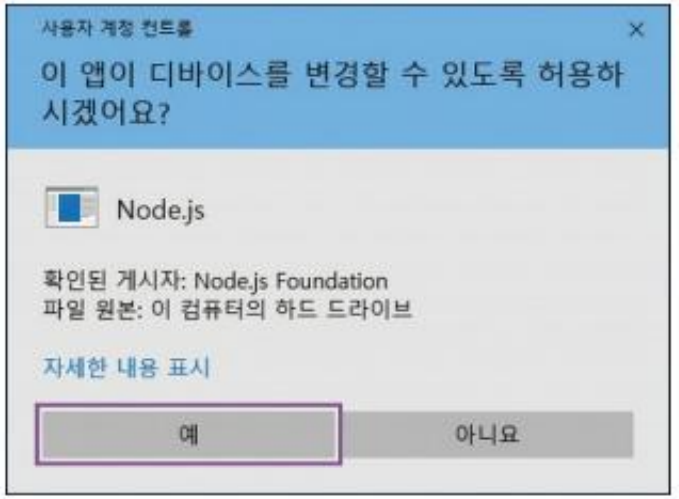


02 개발환경 설정

[Windows]







02 개발환경 설정

[MAC]

```
$ brew install nvm
$ mkdir ~/.nvm
$ vim .zshrc
```

```
# NVM
export NVM_DIR=~/.nvm
source $(brew --prefix nvm)/nvm.sh
```

```
$ source .zhsrc
```

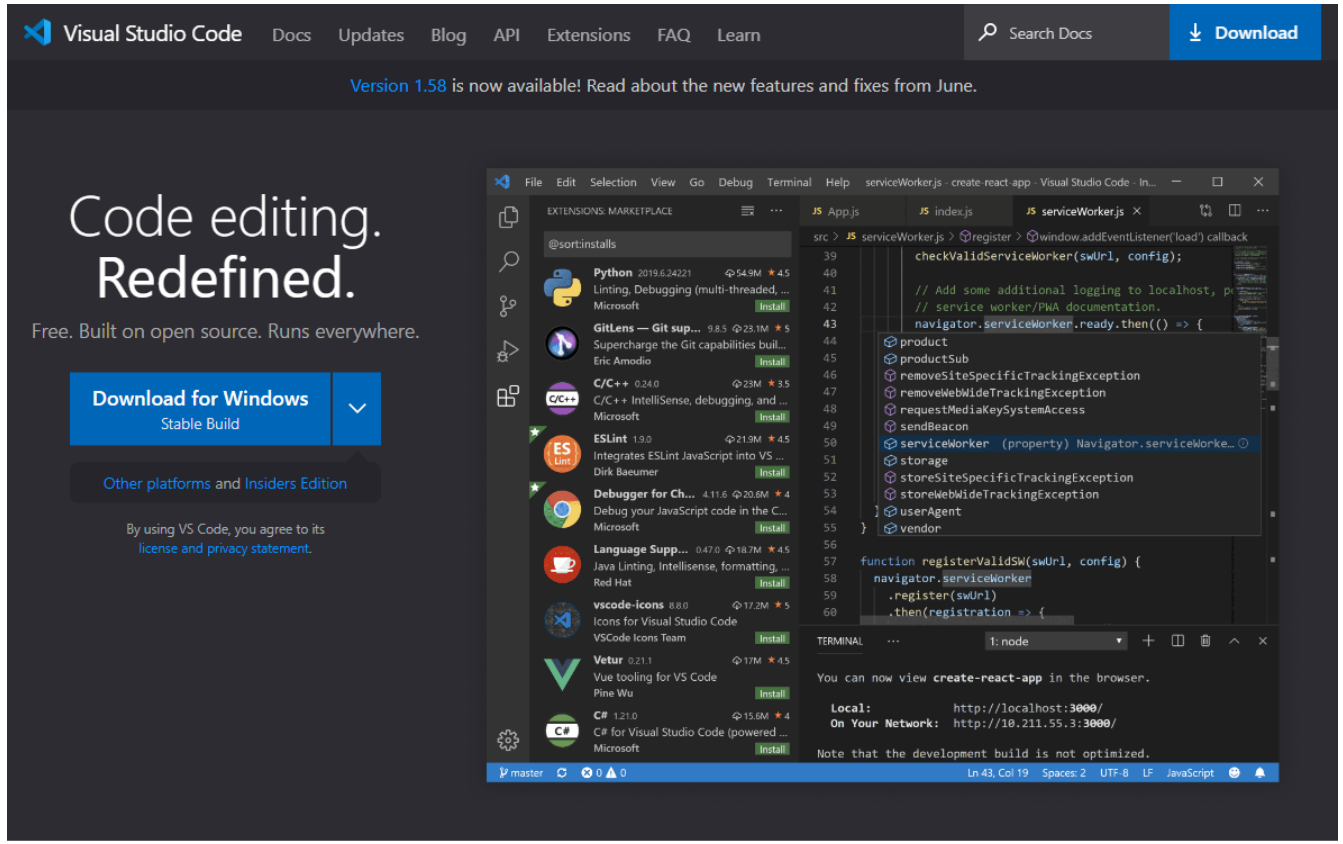
```
$ nvm install node  # "node" is an alias for the latest version
$ nvm use node
```

02 개발환경 설정

> node --version # node version 확인
> npm --version # npm version 확인

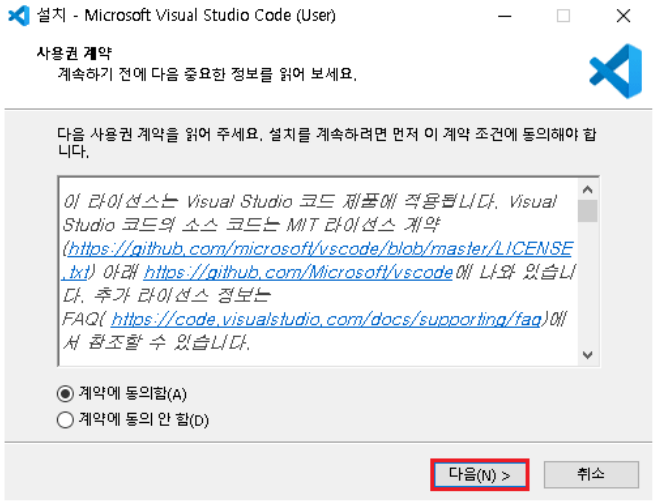
02 개발환경 설정

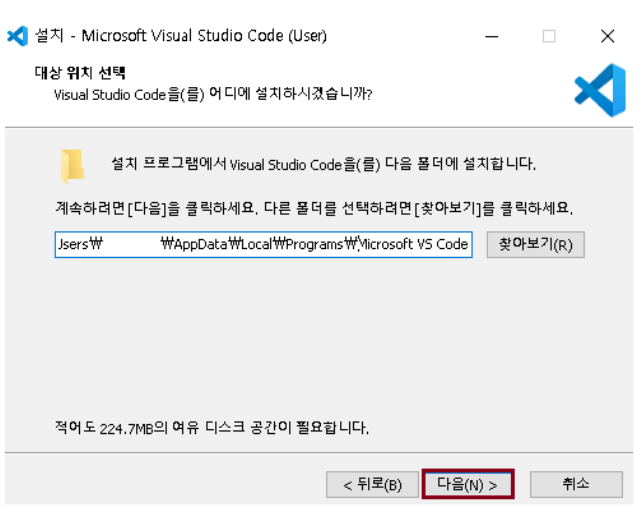
VS CODE 설치 [Windows]

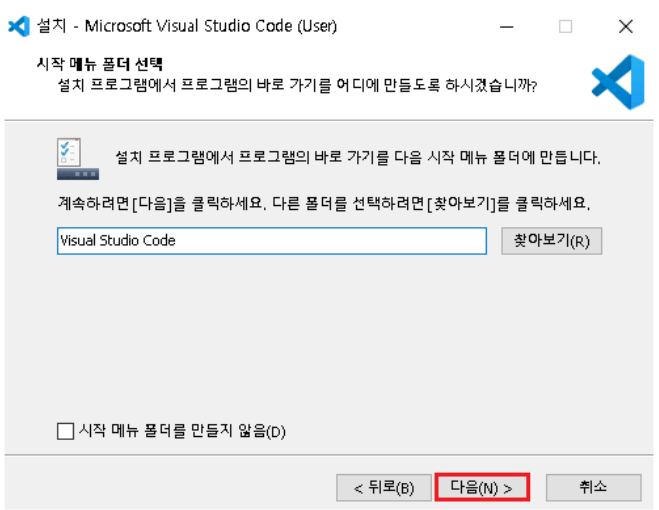


02 개발환경 설정

VS CODE 설치 [Windows]

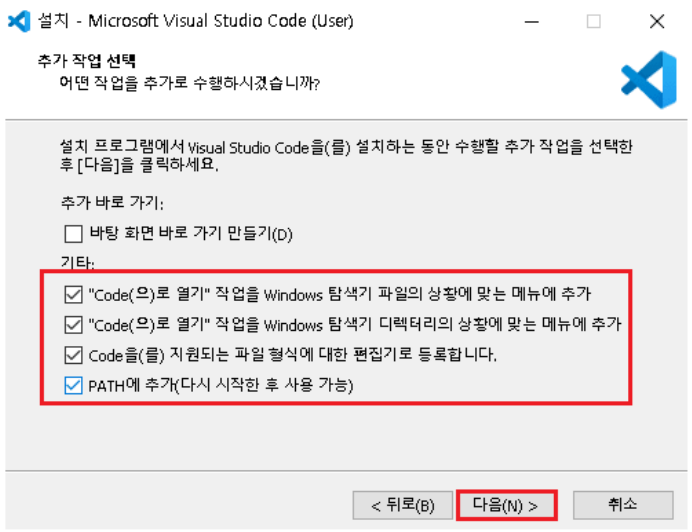


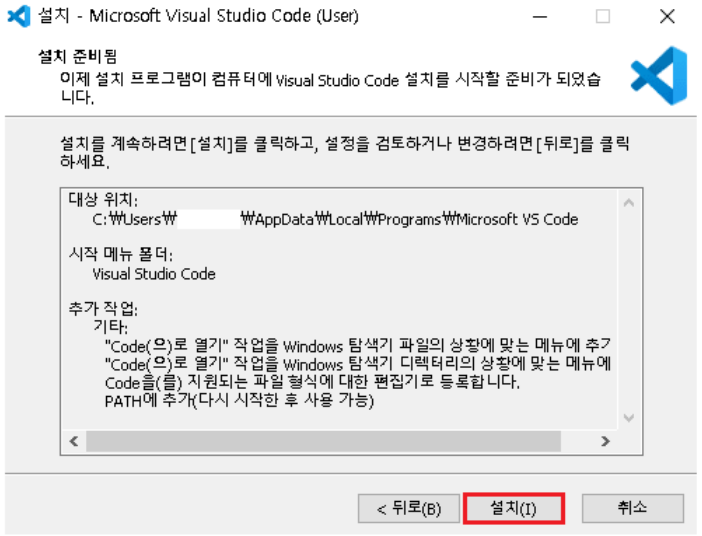




02 개발환경 설정

VS CODE 설치 [Windows]





02 개발환경 설정

VS CODE 설치 [MAC]

```
brew cask install visual-studio-code
```

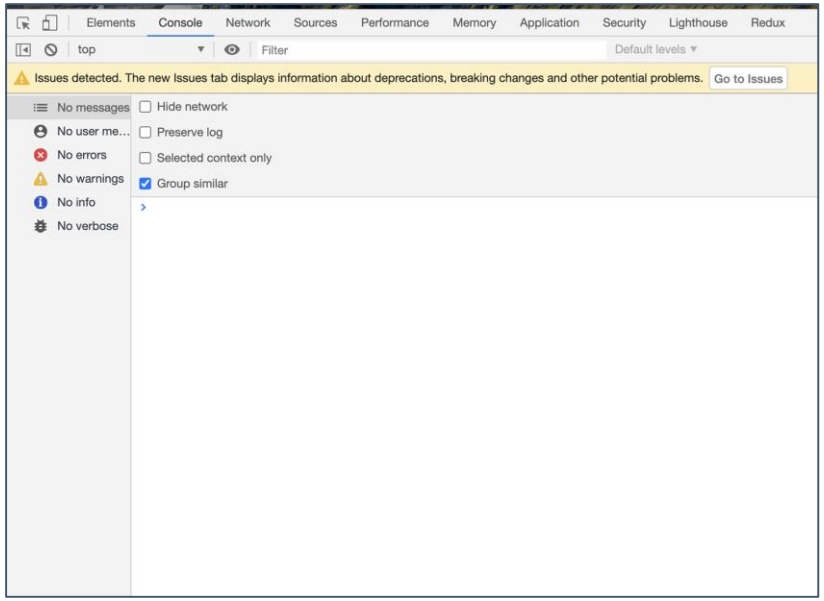
02

JS 살펴보기

02 JavaScript 기초

Javascript 실행방법

개발자 도구에서 실행



Node.js 환경에서 실행 (node <filename>)

basic_js > JS 1. variable.js > ...

```
1   // 안티 패턴
2   var name = 'global';
3   // 전역 변수
4   function func() {
5     console.log(name); // 'undefined' 가 기록된다.
6     var name = 'local';
7     console.log(name); // 'local' 이 기록된다.
8   }
9   func();
10
```

PROBLEMS 4 OUTPUT DEBUG CONSOLE TERMINAL

```
~/betweak/lectures/node_js/basic_js  js기초  node 1.\ variable.js
undefined
local
```

02 JavaScript 기초 문법

변수 선언 & 상수 선언

var **let** **const**

변수선언
(값 변경 가능)

상수선언
(값 변경 불가)

02 JavaScript 기초 문법

변수 선언 & 상수 선언

var **let**

변수선언
(값 변경 가능)

let과 const는 ES6(ECMA 2015) 이후에 나온 키워드이다.

기존에는 var만이 변수를 선언할 수 있었지만, 함수 scope를 가진 var 덕분에 **Hoisting 문제(선언을 함수 스코프의 최상단으로 올려버림)**가 자주 발생하였고, 그에 따라 Block Scope를 가진 let과 const가 개발되었다.

⇒ 개발 단계에서 코드의 실행 예측이 어려워짐

02 JavaScript 기초 문법

변수 선언 & 상수 선언

```
function sample(){
  console.log(name1);
  if (true){
    let name1 = 2;
  }
  console.log(name1);
}
sample()
```

```
function sample(){
  if (true){
    let name1 = 2;
  }
  console.log(name1);
}
sample()
```

```
function sample(){
  console.log(name1);
  if (true){
    var name1 = 2;
  }
  console.log(name1);
}
sample()
```

예측 결과는?

02 JavaScript 기초 문법

변수 선언 & 상수 선언

var **let** **const**

변수선언
(값 변경 가능)

상수선언
(값 변경 불가)

var는 있다는 것만 알아두세요.
(요즘엔 거의 없지만 이전 브라우저에서 호환성 문제가 생길 수 있습니다.)

우리는 let과 const만 사용할 겁니다.

⇒ 나중에 옛 브라우저에서 사용할 경우 babel을 이용해서 버전을 트랜스파일하여 쓸 수 있습니다.

02 JavaScript 기초 문법

Primitive DataType(원시 데이터 타입)

String

```
str1 = “안녕하세요"
typeof(str1)
```

Number

```
int1 = 2
typeof(int1)
```

Boolean

```
bool1 = true
typeof(bool1)
```

null

```
age = null; // ‘존재하지 않는(nothing)’ 값, ‘비어 있는(empty)’ 값, ‘알 수 없는(unknown)’ 값
console.log(age);
```

undefined

```
age; // 값이 할당되지 않은 상태
typeof age;
```

02 JavaScript 기초 문법

Object (객체) 와 array

Object

```
obj = {a: function(){}, b: 3}
typeof obj
```

Array

```
arr = [1, 2, 3]
typeof arr
Array.isArray(arr)
```

null

```
age = null;
typeof(age); // object (오류, 하위 호환성 위해 수정 X)
console.log(age);
```

02 JavaScript 기초 문법

연산자

+	-	*	/
**	%	++	--
거듭제곱	모듈러 연산		

02 JavaScript 기초 문법

비교 연산자

< **<=** **>** **>=**

== **!=**

데이터 형 반영하지 않고 비교

10 == ‘10’ // true

=== **!==**

데이터 형 반영하여 비교

10 === ‘10’ // false

02 JavaScript 기초 문법

논리연산자

||	&&	!
OR 연산자	AND 연산자	NOT

02 JavaScript 기초 문법

js 세미콜론

js statement 뒤에는 세미콜론을 붙여줘야 함.

```
1    console.log("A"); console.log("B");
```

만약 안붙여도 특별한 조건 하에서는 자동으로 세미콜론이 삽입됨. (Automation Semicolon Insertion)
(개행이 있고, 다음 statement와 연결했을 때 말이 되지 않으면)

```
1    console.log("A")
2    console.log("B")
```

잘못된 경우

```
1    a =  1 + 2;
2    console.log(a)
3    (3+4)
```

세미콜론 웬만하면 붙이자!

02 JavaScript 기초 문법

조건문 if

```
if(조건식) {
    실행문;
} else if(조건식 2) {
    실행문 2;
} else {
    실행문 3;
}
```

다른 언어와 비슷

```
function sample(num1){
    if (!num1 || typeof num1 != 'number'){
        console.log("num1 은 숫자여야 합니다.");
        return false;
    }
    if (num1 > 10){
        console.log("10보다 큼.");
    } else if (num1 < 10){
        console.log("10보다 작음.");
    } else{
        console.log("10");
    }
}
sample(null)
```

02 JavaScript 기초 문법

물음표 연산자 (<조건> ? true반환값 : else반환값)

```
1  let age = 2;
2
3  let message = (age < 3) ? "아기야 반가워!" :
4    (age < 18) ? '안녕!' :
5    (age < 100) ? '환영합니다!' :
6    '나이가 아주 많으시거나, 나이가 아닌 값을 입력 하셨군요!';
7
8  console.log(message);
```

02 JavaScript 기초 문법

switch 문

let 변수=초깃값;

```
switch(변수) {
    case 값 1:
        실행문 1;
        break;
    case 값 2:
        실행문 2;
        break;
    default:
        실행문 3;
```

다른 언어와 비슷

```
// let str1 = '1';
// let str1 = 3;
let str1 = 2; // default

switch(str1) {
  case '1':  // if (x === 'value1')
    console.log("1 입력!");
    break;

  case '2':  // if (x === 'value2')
    console.log("2 입력!")
    break;

  case 3:
  case '3':  // if (x === 'value2')
    console.log("3 입력!");

  case '4':
    console.log("4입력!");
    break;
  default:
    console.log("['1','2','3', 3, '4'] 에 포함 X");
}
```

02 JavaScript 기초 문법

간단한 실습1. 다음 switch 코드를 if문으로 변경하여 작성

```
switch (browser) {
  case 'Edge':
    console.log( "Edge를 사용하고 계시네요!" );
    break;

  case 'Chrome':
  case 'Firefox':
  case 'Safari':
  case 'Opera':
    console.log( '저희 서비스가 지원하는 브라우저를 사용하고 계시네요.' );
    break;

  default:
    alert( '현재 페이지가 괜찮아 보이길 바랍니다!' );
}
```

02 JavaScript 기초 문법

반복문

while문

```
let 변수=초깃값;
while(조건) {
  실행문;
}
```

```
let i = 0;
while (i < 3) {
  // 0, 1, 2가 출력됩니다.
  console.log(i);
  i++;
}
```

do while 문

```
let 변수=초깃값;
do {
실행문
} while(조건);
```

```
let i = 0;
do {
  // 0, 1, 2가 출력됩니다.
  console.log(i);
  i++;
} while (i < 3)
```

02 JavaScript 기초 문법

반복문

for (begin; condition; step)

```
for (begin; condition; step) {
    // ... 반복문 본문 ...
}
```

```
for (let i = 0; i < 3; i++) {
    // 0, 1, 2가 출력됩니다.
    console.log(i);
}
```

for <변수> in <object>

```
for (let item in Object) {
    // ... 반복문 본문 ...
}
```

```
const obj1 = {
    a: 1,
    b: 2,
    c: 3
}
for (let key in obj1){
    console.log(key, obj1[key]);
}
```

for <변수> of <array>

```
for (let item of Array) {
    // ... 반복문 본문 ...
}
```

```
const arr = ['a', 'b', 'c', 'd'];
for (let item of arr){
    console.log(item);
}
```

02 JavaScript 기초 문법

loop 제어 (continue / break)

continue

```
for (let i=0; i<10; i++){
  if (i%2 === 0){
    continue;
  }
  // 1, 3, 5, 7, 9
  console.log(i);
}
```

break

```
for (let i=0; i<10; i++){
  if (i === 4){
    break;
  }
  // 1, 2, 3
  console.log(i);
}
```

02 JavaScript 기초 문법

간단한 실습2. 다음 switch 코드를 if문으로 변경하여 작성

사용자가 100보다 큰 숫자를 입력하도록 안내하기.
사용자가 조건에 맞지 않은 값을 입력한 경우 반복문을 사용해 동일한 입력을 받습니다.

사용자가 오직 숫자만 입력한다고 가정하고 답안을 작성하도록 해봅시다.
숫자가 아닌 값이 입력되는 예외 상황은 처리하지 않아도 됩니다.

> 이번 코드는 브라우저에서 진행합니다.
// (node.js에는 prompt 없으며, 동기적으로 받는 내장 함수 X)

browser에서 입력받기
> prompt(“알림 내용”);

02 JavaScript 기초 문법

함수 선언

```
function <함수이름> (인자){
        ..코드
}
```

```
1   console.log(min(3, 5)) // 성공
2
3   function min(a, b){
4     return a<b ? a : b
5   }
6
7   console.log(min(3, 5))
```

함수 선언문 정의 되기 전에 실행 가능.

함수 표현식과 화살표 함수

```
const 함수이름 = (인자)=>{
        ..코드
}
const 함수이름 = function(인자) {
        ..코드
}
```

```
1   // console.log(max1(1, 2)) // 오류
2   const max1 = function(a, b){
3     return a>b ? a : b;
4   }
5
6   const max2 = (a, b) =>{
7     return a>b ? a : b;
8   }
9   // return값만 있을땐 다음과 같이 작성도 가능
10  const max3 = (a, b) => (a>b ? a : b);
11
12  console.log(max1(5, 7))
13  console.log(max2(1, 2))
14  console.log(max3(3, 5))
```

함수 표현 전엔 호출 불가능.

02 JavaScript 기초 문법

간단한 실습3. (브라우저 진행) 함수

ask 함수를 정의하여라.
confirm을 통해 입력받고,
사용자가 yes를 클릭하면, “동의”
사용자가 no를 클릭하면 “비동의”를
alert로 출력하여라.

[심화]
ask 함수를 정의하여라.
prompt를 통해 입력받아라.

사용자가 yes를 클릭하면, 진행할 기능도 인자로 전달받고. 호출하여야 함.
사용자가 no를 클릭하면, 진행할 기능도 인자로 전달받아고, 호출하여야 함.

02 JavaScript 기초 문법

객체의 특성

```
1   const KEY = "SAMPLE"
2   let johnProfile = {
3     name: "John",
4     sayHi: function() {
5       console.log(this.name, ": 친구야 반갑다!");
6     },
7     [KEY]: "sampleValue",
8   };
9
10  johnProfile.sayHi();
11  console.log(johnProfile["sayHi"]);
12  console.log(johnProfile.SAMPLE);
```

- object는 속성(property)과 메서드를 가진다.
- this 키워드 사용 가능
- []로 key값 접근 가능
- key가 숫자형일땐 오름차순 정렬, 그 외에는 추가한 순서대로 정렬
- 변수의 값을 key로 사용 가능

02 JavaScript 기초 문법

Primitive Type의 메서드

- object는 속성(property)과 메서드를 가진다.
☐ Primitive Type을 object화 시키자니 너무 무겁다.
☐ 원시값의 속성에 접근하면 wrapper 객체를 만들어 사용하자.

Number

String

Boolean

```
let num1 = 0;
let num2 = new Number(0); // 생성자로 사용 말자.

console.log(typeof(num1), Boolean(num1)); // number false
console.log(typeof(num2), Boolean(num2)); // object true

let str1 = "hello world";
let str2 = str1.toUpperCase(); //  object wrapper 만들고 사라짐.
console.log(typeof(str1), str1); // string, hello world
console.log(typeof(str2), str2); // string, HELLO WORLD
```

02 JavaScript 기초 문법

Template Literal

```
> let var1 = “나는 변수”
> let sample = `변수가 문자열에 ${var1}  들어갈 수 있습
니다.`  // back-tick(대부분 tab 위에 존재)
> console.log(sample)
```

02 JavaScript 기초 문법

구조분해 할당

```
let arr = ["Kim", "Shin"];

// 구조 분해 할당
let [name1, name2] = arr;

console.log(name1); // Kim
console.log(name2);  // Shin

let arr2 = ['a', 'b', 'c', 'd'];
let [v1, v2, ...v3] = arr2;
console.log(v1); // 'a'
console.log(v2); // 'b'
console.log(v3); // ['c', 'd']
```

```
let options = {
  title: "Menu",
  width: 100,
  height: 200,
  k1: 'v1',
  k2: 'v2',
};

// { 객체 프로퍼티: 목표 변수 }
let {width, height: h, ...rest} = options;

console.log(width); // width
console.log(h); // height
console.log(rest); // { title: 'Menu', k1: 'v1', k2: 'v2' }
```

02 JavaScript 기초 문법

배열 관련 함수.

```
let arr = [1, 2, 3]
```

함수

push(n) - 값 추가

pop() - 마지막 값 꺼내고 반환

slice(start, end) - index로 subset 만들기

join(token) - 배열을 token으로 연결해서 문자열 만들기 (String.split과 반대)

concat(arr) - 배열을 추가해서 새로운 배열 만들기 (배열 연결하기)

filter(function) - 아이템을 순회하면서 함수가 true를 return 하는 값만 모아 새로운 배열 만들기

map(function) - 아이템을 순회하면서 function의 return값을 모아 새로운 배열 만들기

02 JavaScript 기초 문법

try, catch문

```
try{
  // 코드 진행
} catch (err){
  // 에러 핸들링
} finally{
  // 무조건 실행됨.
}
```

런타임 에러를 잡자!

```
let json = "{ bad json }";
// JSON.parse(json); // 에러 발생

try{
  console.log("code1");
  JSON.parse(json);
} catch (err){
  console.log("error 처리")
}

// 에러 만들어 던지기
try{
  console.log("code1");
  throw new SyntaxError("새로운 에러 만들기")
} catch(err){
  console.log(err.name);
  console.log(err.message);
}
```

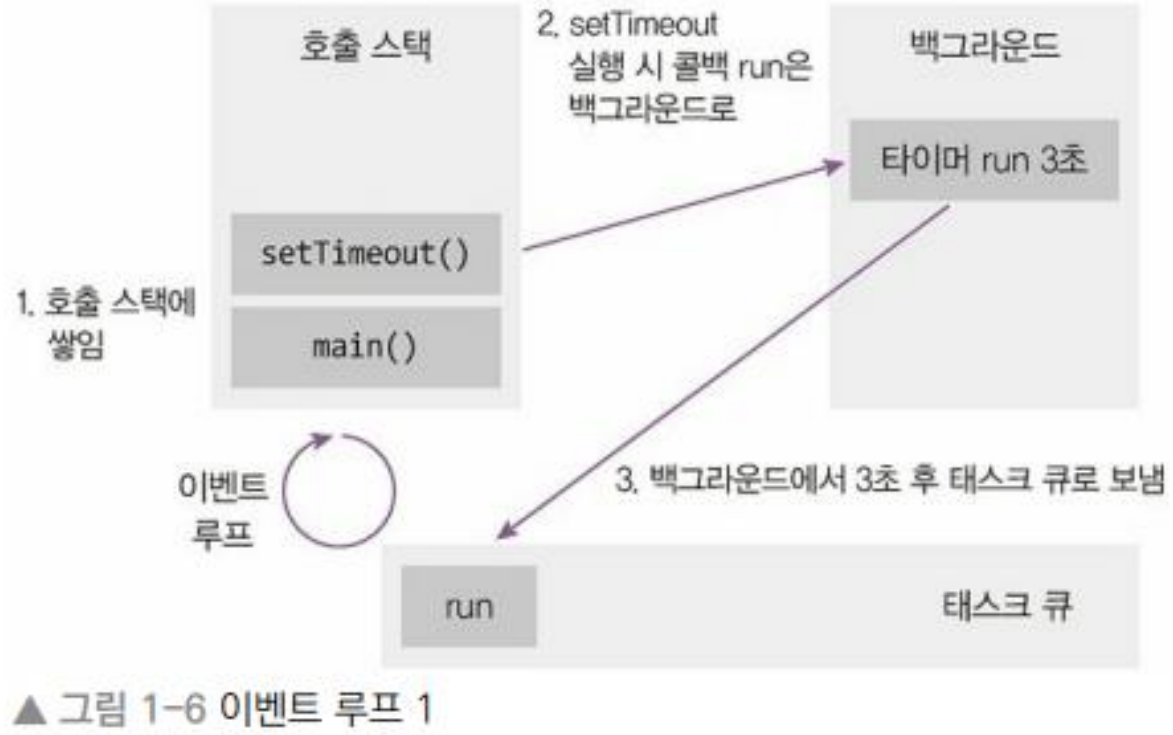
02 JavaScript 기초 문법

Node.js는 대부분 비동기적으로 동작!

```
function run() {
  console.log('3초 후 실행');
}
console.log('시작');
setTimeout(run, 3000);
console.log('끝');
```

콘솔

```
시작
끝
3초 후 실행
```



▲ 그림 1-6 이벤트 루프 1

예상 결과는?

02 JavaScript 기초 문법

Node.js는 대부분 비동기적으로 동작!

```
function run() {
  console.log('3초 후 실행');
}
console.log('시작');
setTimeout(run, 3000);
console.log('끝');
```

비동기적으로 동작하면?

- 해당 비동기 함수가 완료될 경우에 처리해야할(이어지는 Task)를 처리하지 못할 수 있음
- Idea: 비동기 함수가 완료될 경우에 처리한 function을 함께 넣어주자 □ callback function

예상 결과는?

02 JavaScript 기초 문법

Node.js는 대부분 비동기적으로 동작!

```
function run() {
  console.log('3초 후 실행');
}
console.log('시작');
setTimeout(run, 3000);
console.log('끝');
```

비동기적으로 동작하면?

- 해당 비동기 함수가 완료될 경우에 처리해야할(이어지는 Task – 비동기 내에 Blocking 해야할 함수)를 처리하지 못할 수 있음
- Idea: 비동기 함수가 완료될 경우에 처리한 function을 함께 넣어주자 □ callback function

예상 결과는?

02 JavaScript 기초 문법

Node.js는 대부분 비동기적으로 동작!

전통적인 방식 (callback function)□ 비동기적으로 전달하되, 비동기 함수 내에서 블로킹 한 후에 호출할 함수를 함께 전달해주자!

fastFunction □ slowFunction 은 동기적으로 처리해야 함.

```
function fastFunction (err, data, done) {
  setTimeout(function () {
    done(undefined, data*2);
  }, 1000)
}

function slowFunction (err, data, done) {
  setTimeout(function () {
    done(undefined, data + 10);
  }, 3000)
}
```

```
function runTasks (callback) {
  fastFunction(undefined, 10, (err, data) => {
    if (err) return callback(err);
    console.log("fastFunction", data);   // results of a: 10 *2 = 20

    slowFunction(undefined, data, (err, data) => {
      if (err) return callback(err);
      console.log("slowFunction", data); // results of b: 20 + 10 = 30

      // here you can continue running more tasks
    })
  })
}

runTasks(err=>{
  console.error(err);
})
```

02 JavaScript 기초 문법

Node.js는 대부분 비동기적으로 동작!

```
function fastFunction (err, data, done) {
  setTimeout(function () {
    done(undefined, data*2);
  }, 1000)
}

function slowFunction (err, data, done) {
  setTimeout(function () {
    done(undefined, data + 10);
  }, 3000)
}
```

```
function runTasks (callback) {
  fastFunction(undefined, 10, (err, data) => {
    if (err) return callback(err);
    console.log("fastFunction", data);   // results of a: 10 *2 = 20

    slowFunction(undefined, data, (err, data) => {
      if (err) return callback(err);
      console.log("slowFunction", data); // results of b: 20 + 10 = 30

      // here you can continue running more tasks
    })
  })
}

runTasks(err=>{
  console.error(err);
})
```

그런데.. 가독성이 너무 떨어진다.

- 지금은 callback이 하나지만,,
- 이어지는 함수들이 많아지면???..
- 코드의 깊이가 깊어지겠지..

콜백 지옥..

02 JavaScript 기초 문법

Promise 객체!

callback은 가독성이 너무 떨어져.. 비동기를 지원하는 새로운 객체를 만들자.

```
function fastFunction (data) {
  return new Promise((resolve, reject) => {
    setTimeout(function () {
      const result = data * 2;
      console.log('Fast function done', result) // 20
      // reject(new Error("임시 에러")); // catch로 잡음.
      resolve(result); // then으로 받음.
    }, 1000)
  })
}

function slowFunction (data) {
  return new Promise((resolve, reject) => {
    setTimeout(function () {
      const result = data + 10;
      console.log('Slow function done', result) //30
      // reject(new Error("임시 에러")); // catch로 잡음.
      resolve(result); // then으로 받음.
    }, 3000)
  })
}
```

```
function runTasks () {
    return fastFunction(10)
    .then((data)=>{
      return slowFunction(data);
    }).then(data=>{
      console.log("작업완료");
    }).catch(err=>{
      console.error(err);
    })
}

runTasks();
```

02 JavaScript 기초 문법

Promise 객체!

```
function fastFunction (data) {
  return new Promise((resolve, reject) => {
    setTimeout(function () {
      const result = data * 2;
      console.log('Fast function done', result) // 20
      // reject(new Error("임시 에러")); // catch로 잡음.
      resolve(result); // then으로 받음.
    }, 1000)
  })
}

function slowFunction (data) {
  return new Promise((resolve, reject) => {
    setTimeout(function () {
      const result = data + 10;
      console.log('Slow function done', result) //30
      // reject(new Error("임시 에러")); // catch로 잡음.
      resolve(result); // then으로 받음.
    }, 3000)
  })
}
```

```
function runTasks () {
    return fastFunction(10)
    .then((data)=>{
      return slowFunction(data);
    }).then(data=>{
      console.log("작업완료");
    }).catch(err=>{
      console.error(err);
    })
}

runTasks();
```

02 JavaScript 기초 문법

Promise.all()

비동기 함수를 병렬적으로 시작해서 결과를 모을래.

```
23  function runTasks () {
24      Promise.all([fastFunction(10), slowFunction(5)])
25      .then(([result1, result2])=>{
26        console.log("작업완료", result1, result2);
27      });
28  }
29
30  runTasks();
```

callback보다는 좋아보여. 깊어지진 않으니
☐ 근데 여전히 then, catch로 함수를 전달해주는 것이 마음에 들지 않아..
☐ then지옥?

02 JavaScript 기초 문법

async // await

☐ **함수의 문법(키워드)으로 추가하자.**

☐ 마치 동기적 코드처럼 비동기 함수를 작성하자

```
1   const delay = ms => new Promise(resolve => setTimeout(resolve, ms));
2
3   function fastFunction (data) {
4     return new Promise((resolve, reject) => {
5       setTimeout(function () {
6         const result = data * 2;
7         console.log('Fast function done', result) // 20
8         // reject(new Error("임시 에러")); // catch로 잡음.
9         resolve(result); // then으로 받음.
10      }, 1000);
11    })
12  }
13
14  async function slowFunction (data) {
15    await delay(3000);
16    const result = data + 10;
17    console.log('slow function done', result) // 20
18    return result;
19  }
```

```
21  async function runTasks () {
22      let result = await fastFunction(10);
23      result = await slowFunction(result);
24      console.log("작업완료", result);
25  }
26
27  runTasks();
```

- Promise와 함께 사용할 수 있다.
- setTimeout은 Timeout 객체를 반환하므로 Promise 로 만들어줌.
- async await를 사용하면 마치 동기적 코드처럼 작성할 수 있음

02 JavaScript 기초 문법

Prototype (js의 OOP)

리터럴로 객체 선언

```
1   // object
2   const animal1 = {
3     name: 'lion',
4     run(){
5       console.log(`${this.name} 동물이 달린다.`)
6     },
7     run2: function(){
8       console.log("동물이 달려요.")
9     }
10  };
11  animal1.run();
12  animal1.run2();
```

02 JavaScript 기초 문법

Prototype (js의 OOP)

생성자 함수로 객체 생성

```
1   // 만약 같은 형태의(속성을 가진) 객체를 여러개 만들고 싶으면?
2   // javascript는 생성자 함수.
3   function Animal(name){
4     this.name = name;
5     this.run = function(){
6       console.log(`${this.name} 동물이 달린다.`)
7     }
8   }
9
10  const animal2 = new Animal("사자");
11  console.log(animal2)
12  console.log(animal2.constructor)
13  animal2.run()
```

02 JavaScript 기초 문법

Prototype (js의 상속방법)

prototype과 _ _proto_ _

```
// 만약 같은 형태의(속성을 가진) 객체를 여러개 만들고 싶으면?
// javascript는 생성자 함수.
function Animal(name){
  this.name = name || 'lion';
  this.run = function(){
    console.log(`${this.name} 동물이 달린다.`)
  }
}

const animal2 = new Animal("사자");
console.log(animal2)
console.log(animal2.constructor)
animal2.run()

Animal.prototype.eat = function(){
  console.log(`${this.name}가 먹는다.`);
}
```

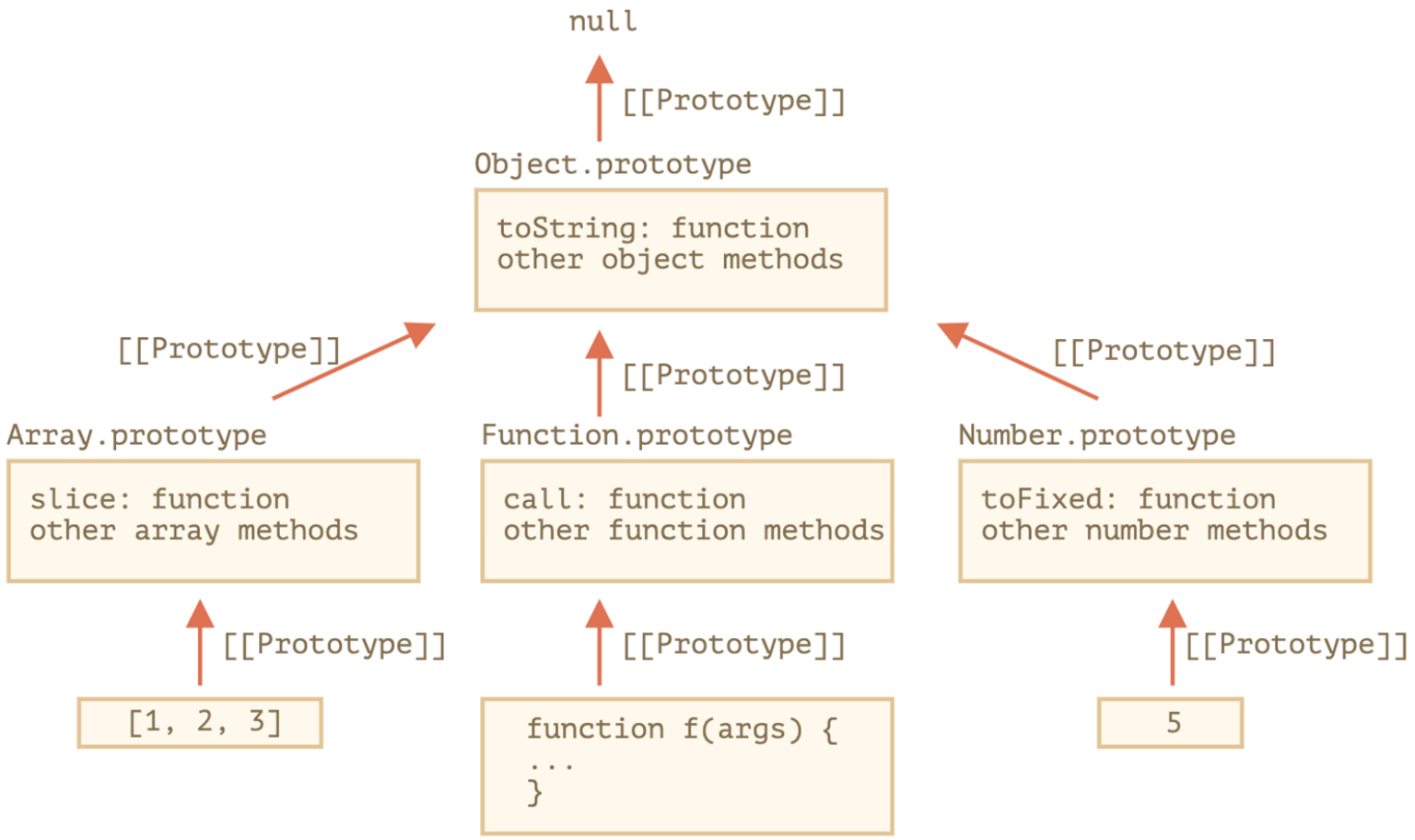
```
Animal.prototype.eat = function(){
  console.log(`${this.name}가 먹는다.`);
}

animal2.eat();
console.log(animal2);
console.log(animal2.__proto__);
```

- constructor는 생성자 함수 그 자체
- prototype은 생성자 함수에 정의한 모든 객체가 공유할 원형
- __proto__는 생성자 함수를 new로 호출할 때, 정의해두었던 prototype을 참조한 객체
- prototype은 생성자 함수에 사용자가 직접 넣는 거고, __proto__는 new를 호출할 때 prototype을 참조하여 자동으로

02 JavaScript 기초 문법

Prototype (js의 상속방법)



02 JavaScript 기초 문법

Prototype (js의 상속방법)

```
12  // javascript는 상속을 prototype으로 한다.
13  function Rabbit(name, color){
14    // arguments: 함수라면 가지고 있는 특수한 변수, 인자를 나타내는 유사배열;
15    Animal.apply(this, arguments); // Animal함수를 적용하되 context를 this(Rabiit)으로 한다.
16    this.color = color;
17  }
18
19  // Object.create함수는 객체는 만들되 생성자는 실행하지 않음
20  Rabbit.prototype = Object.create(Animal.prototype);
21  Rabbit.prototype.constructor = Rabbit; // 생성자의 클래스의 생성자는 자기자신.
22
23  const rabbit1 = new Rabbit("토끼", "white");
24
25  rabbit1.run();
26  rabbit1.eat();
27  console.log(rabbit1);
28  console.log(rabbit1.__proto__);
```

02 JavaScript 기초 문법

class 문법 (무늬만 클래스, 실제로는 prototype 기반으로 동작)

```
1   class Animal {
2     constructor(name='lion'){
3       this.name = name;
4     }
5     run(){
6       console.log(`${this.name} 동물이 달린다.`)
7     }
8     eat(){
9       console.log(`${this.name}가 먹는다.`);
10    }
11  }
```

```
13  class Rabbit extends Animal{
14    constructor(name, color){
15      super(name);
16      this.color = color;
17    }
18    newFunction(){
19      console.log(`${this.name} 가 새로운 기능을 가졌다!`);
20    }
21  }
22  const rabbit1 = new Rabbit("토끼", "white");
23  console.log(rabbit1);
24  rabbit1.run();
25  rabbit1.newFunction();
```

02 브라우저에서만 사용되는 객체

BOM(Browser Object Model)

- window(브라우저 객체의 최상위 객체. (global 선언))
  - open("url 경로", "창 이름", "옵션 설정") - url 열기
  - alert - 경고 창
  - confirm("질의 내용") - 확인/취소 창을 띄움.(true/false)

- screen(사용자의 모니터 정보를 제공)
  - width/heigh

- location(페이지 위치)
  - href - 주소 영역에 참조 주소를 설정하거나 URL 반환.

- history(사용자 위치 history)
  - back() - 뒤로가기
  - forward() - 앞으로 가기

DOM(Document Object Model)

- 선택자
  - document.getElementBy<something>
  - <선택자>.[parentNode, childNodes, nextSibling, previousSibling]

- 이벤트 핸들러
  - onclick(function)
  - onmouseover(function)
  - onchange(function)
  - onsubmit(function)
  - onresize(function)
  - ...