6. Introduction to node.js (1/2)

2023학년 2학기 웹응용프로그래밍

권 동 현

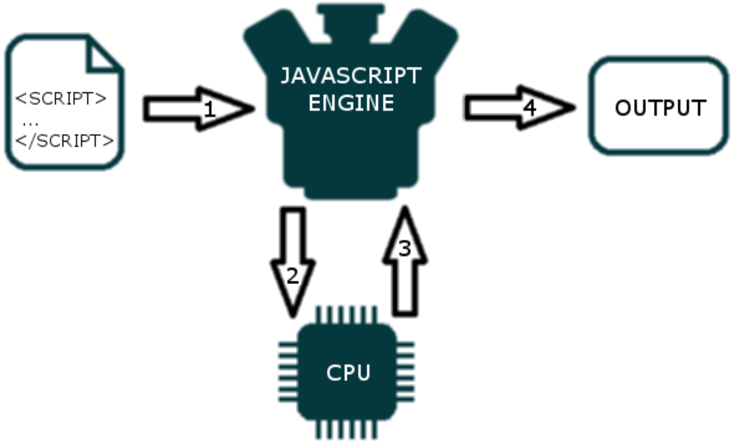
Contents

- What is node.js?
- Features of node.js
- Uses of node.js
- Install node.js
- How to use node.js
- Node.js built-in objects
- Node.js core modules

What is node.js?

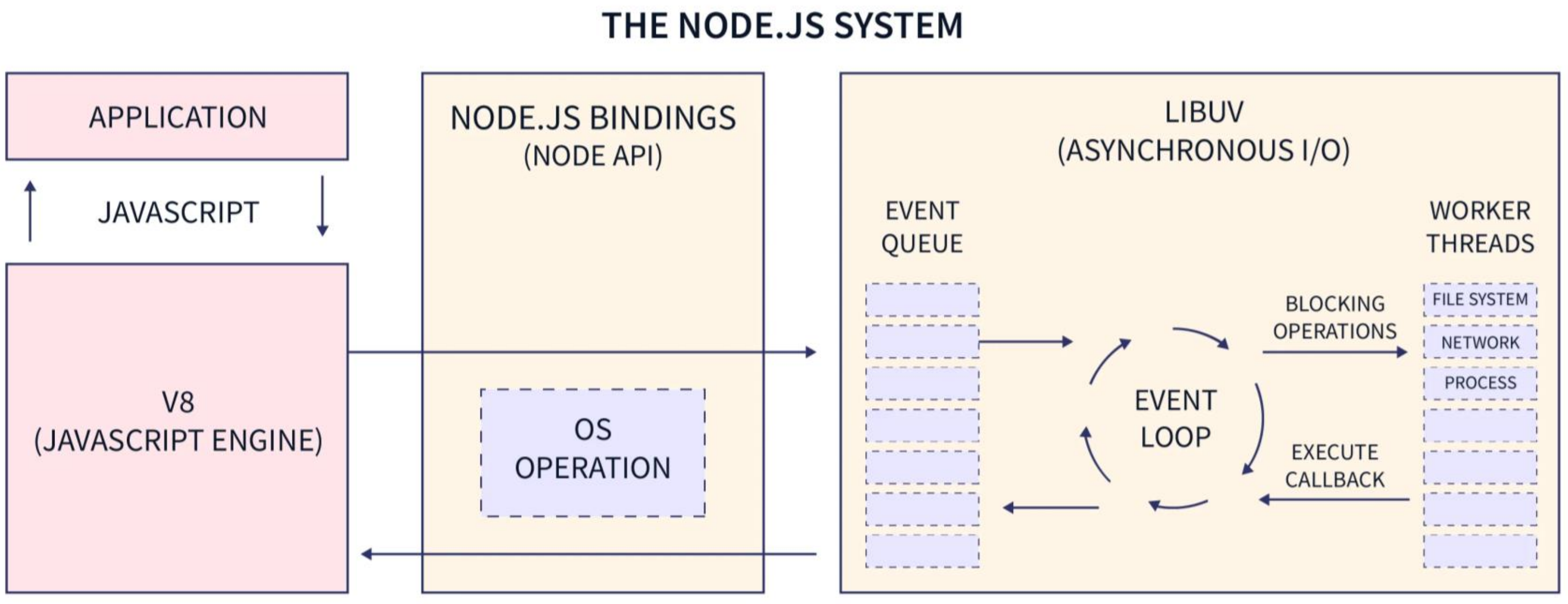
What is node.js?

- An open-source, cross-platform JavaScript runtime environment
  - According to official site
- Open-source
  - You can access the source code of node.js
  - https://github.com/nodejs/node
- Cross-platform
  - can run on Windows, Linux, Unix, macOS, and more.
- JavaScript runtime
  - runs the V8 JavaScript engine, Google's open-source JavaScript engine



Node Architecture

- node.js bindings
- V8
  - JavaScript runtime
- libuv
  - multi-platform support library with a focus on asynchronous I/O



https://www.scaler.com/topics/nodejs/node-js-architecture/

Features of node.js

CSLAB

Features of node.js

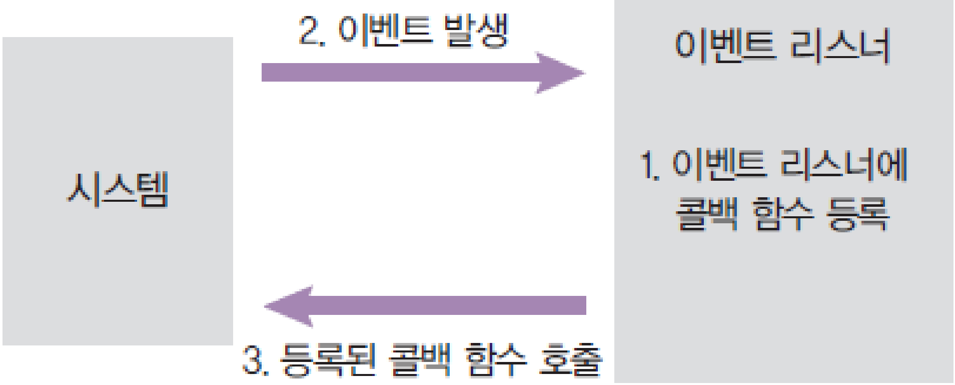
- Event-driven
- Non-blocking I/O
- Single threaded

Event-driven

- Executing pre-defined tasks when an event occurs
  - Examples of events: click, network request, timer, etc.
  - Event listener: a function that registers events.
  - Callback function: a function that is executed when an event occurs.

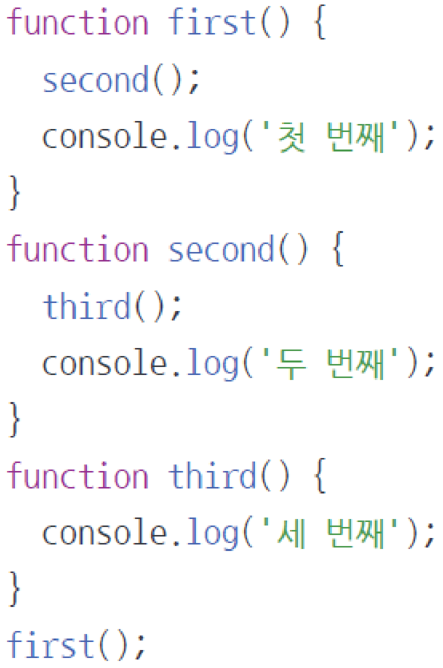
- If there are no events or all events have been handled, it waits until the next event occurs.

그림 1–4 이벤트 기반



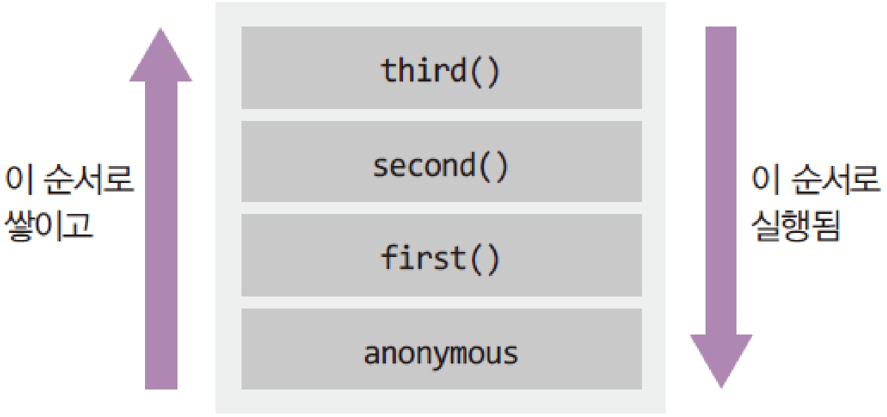
Event-driven

- Predicting the order of the code
  - 세 번째 -> 두 번째 -> 첫 번째
- An easy way to understand: Draw the **call stack**.
  - Anonymous represents a virtual global context (it's always good to assume it's present).
  - Functions stack up in the order of their invocation and are executed in reverse order.
  - Once a function execution is complete, it is removed from the stack.
  - It's called a stack due to its Last-In-First-Out (LIFO) structure.



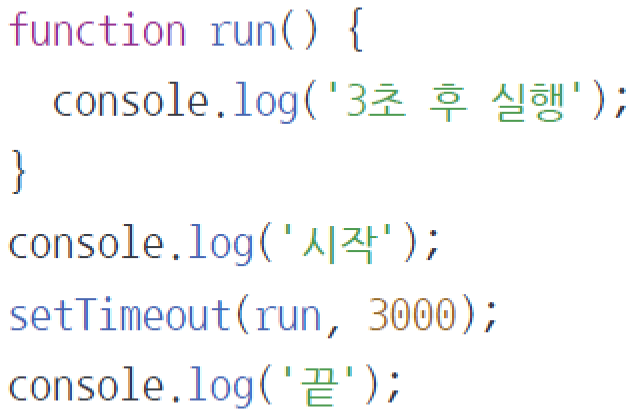
```
function first() {
  second();
  console.log('첫 번째');
}
function second() {
  third();
  console.log('두 번째');
}
function third() {
  console.log('세 번째');
}
first();
```

그림 1-5 호출 스택



Event-driven

- Predicting the order of the code
  - Start -> End -> Execute after 3 seconds
  - Explanation can't be done with call stack alone (run wasn't called?)
  - Explanation can be done with call stack + event loop.

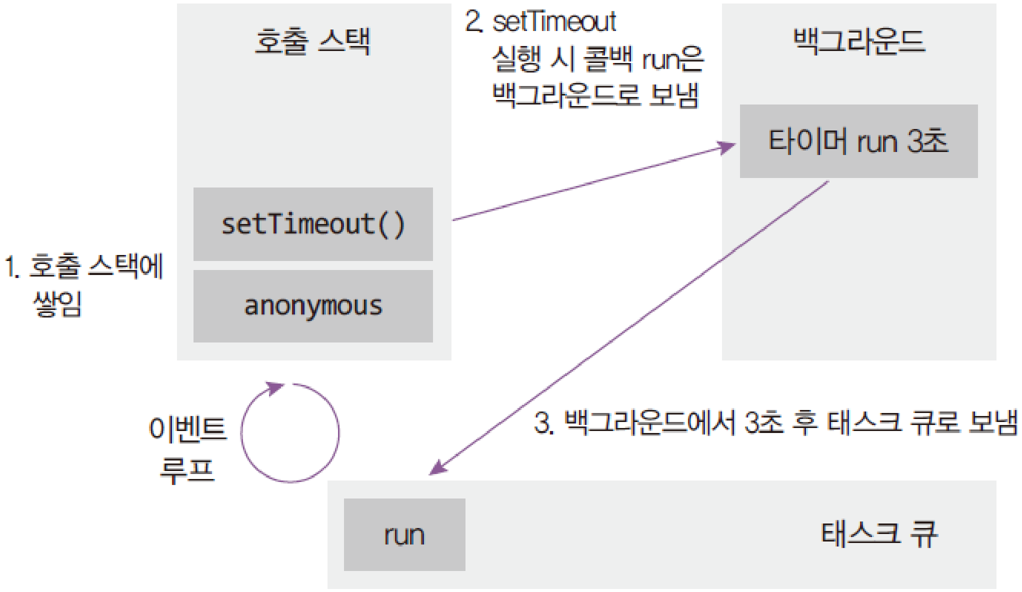


```
function run() {
  console.log('3초 후 실행');
}
console.log('시작');
setTimeout(run, 3000);
console.log('끝');
```

Event-Driven

- Event Loop
  - manages callback functions (such as "run" in the example) to be invoked when events occur (e.g., setTimeout), determining the order of their execution.
- Task Queue
  - the space where callback functions, waiting to be called after events, are queued in a specific order
- Background
  - the space where timers, I/O operation callbacks, and event listeners wait. Multiple tasks can run concurrently in the background.

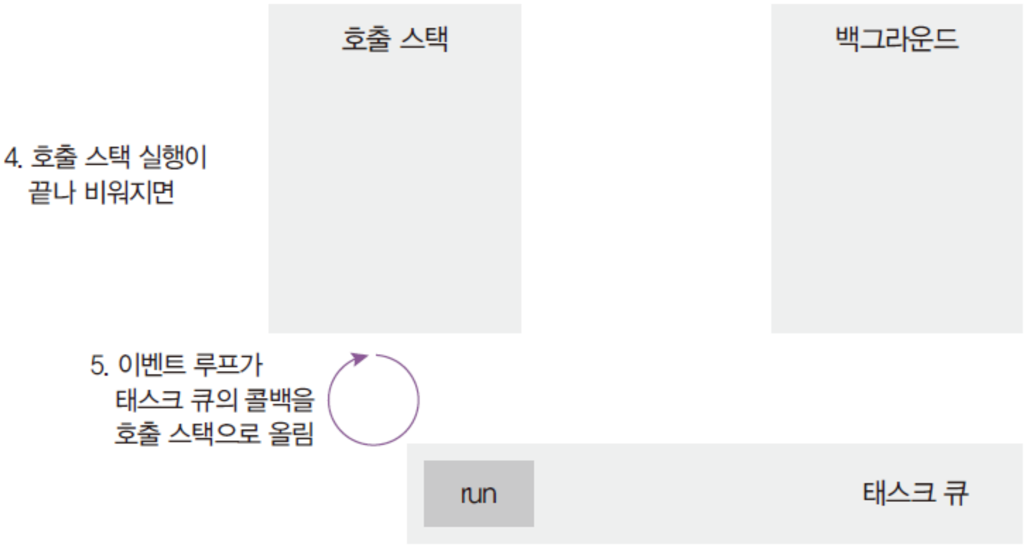
그림 1-6 이벤트 루프 1



Event-driven

- In the example code, when setTimeout is called, the callback function "run" goes to the background.
  - In the background, a timer for 3 seconds is initiated.
  - After 3 seconds, the task is moved from the background to the task queue.
- Once setTimeout and the anonymous function have completed, and the call stack is completely empty, the event loop moves the callback from the task queue to the call stack.

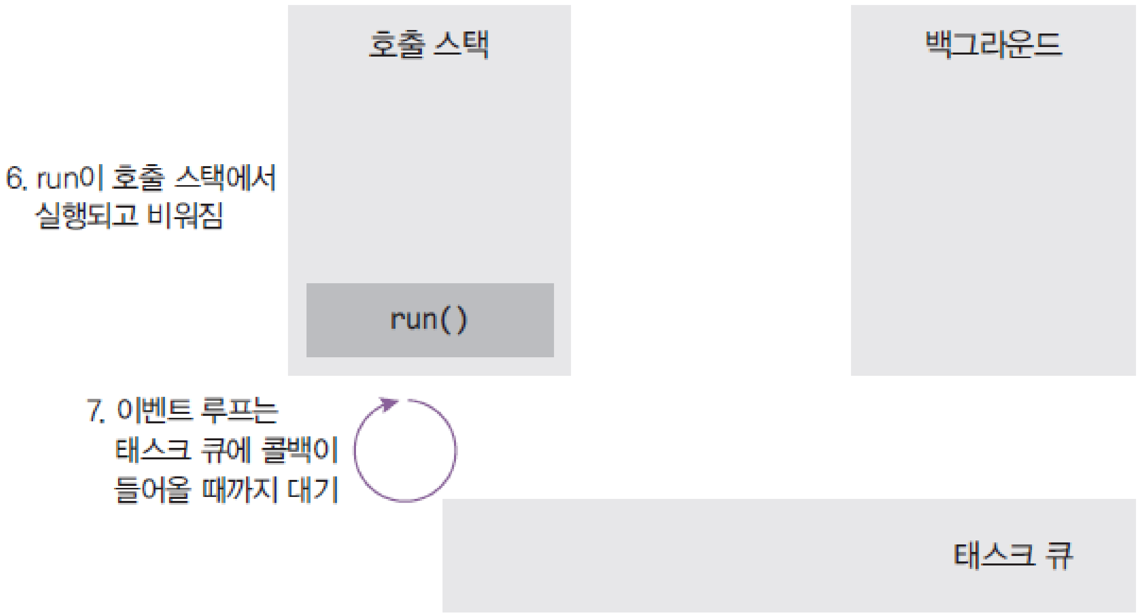
▼ 그림 1-7 이벤트 루프 2


호출 스택
백그라운드
4. 호출 스택 실행이 끝나 비워지면
5. 이벤트 루프가 태스크 큐의 콜백을 호출 스택으로 올림
run
태스크 큐

Event-driven

- when the "run" function is invoked, it enters the call stack, executes, and then exits the call stack.
  - event loop continuously waits until the next function arrives in the task queue.
  - There can be multiple task queues, and the event loop determines the order between task queues and their associated functions.

그림 1-8 이벤트 루프 3



6. run이 호출 스택에서 실행되고 비워짐

7. 이벤트 루프는 태스크 큐에 콜백이 들어올 때까지 대기

Non-blocking I/O

- Non-blocking: Sending time-consuming functions to the background to allow the execution of the following code and later executing the time-consuming function.
  - In a non-blocking approach, some code can run in parallel in the background.
  - Examples of such code include I/O operations (file system access, network requests), compression, encryption, etc.
  - The remaining code operates in a blocking manner.
  - Therefore, when there are many I/O operations, Node.js utilization is maximized, enhancing its efficiency.

그림 1-9 블로킹과 논블로킹

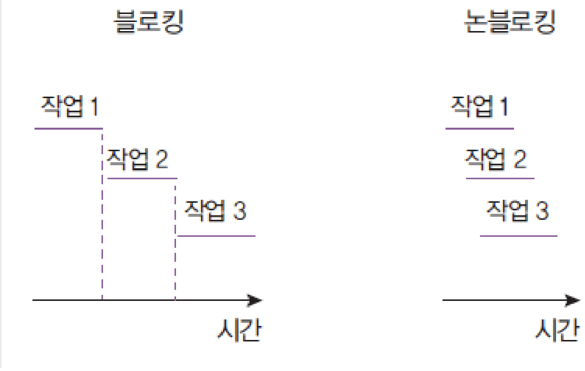
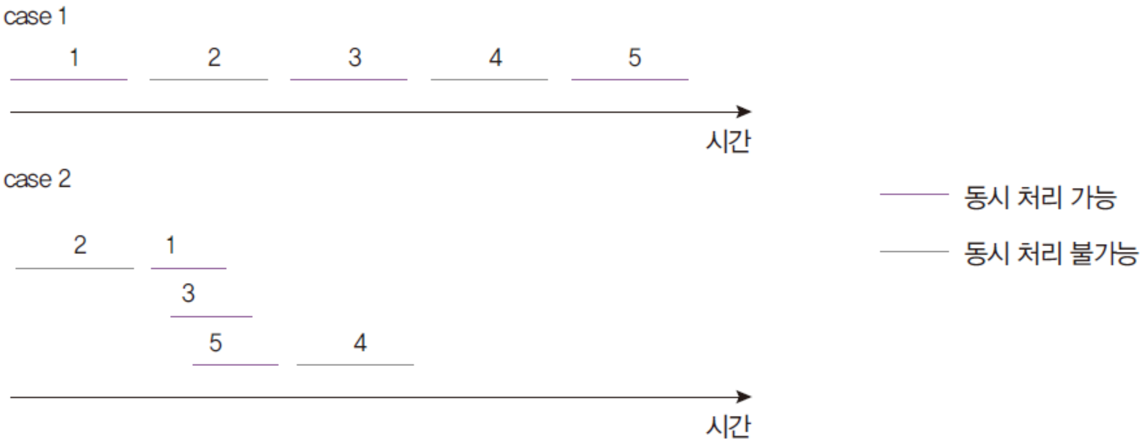


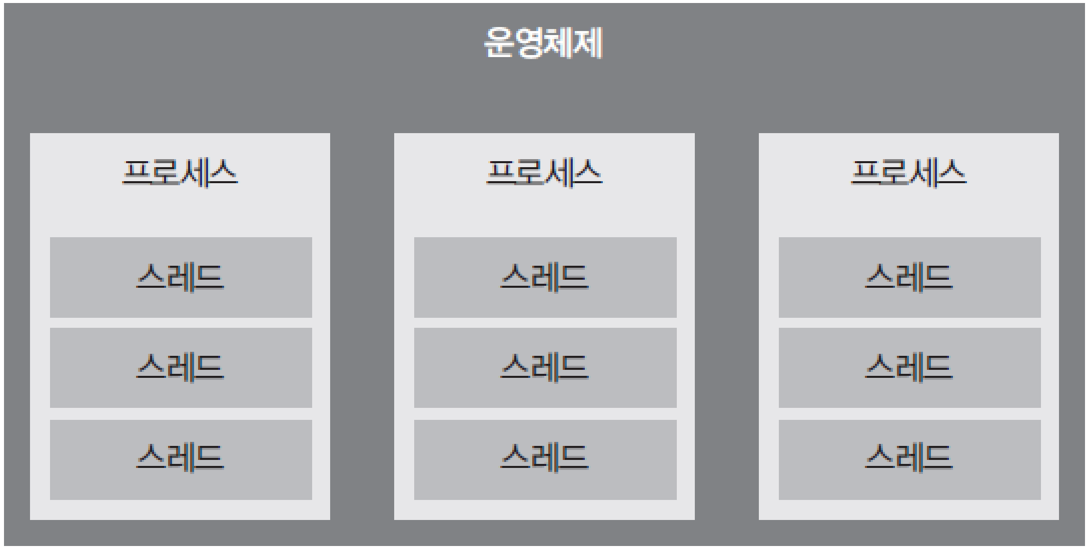
그림 1-10 동시 처리로 얻는 시간적 이득



Single Thread

- Process and Thread
  - Process: Unit of work allocated by the operating system, with no resource sharing between processes.
  - Thread: Unit of work executed within a process, sharing resources with the parent process.
- The Node.js process is multi-threaded, but it is commonly referred to as single-threaded because it only has one thread that developers can directly manipulate.
- Node.js primarily utilizes multi-processes instead of multi-threads.
  - Starting from version 14, Node.js supports multi-threading.

그림 1-13 스레드와 프로세스

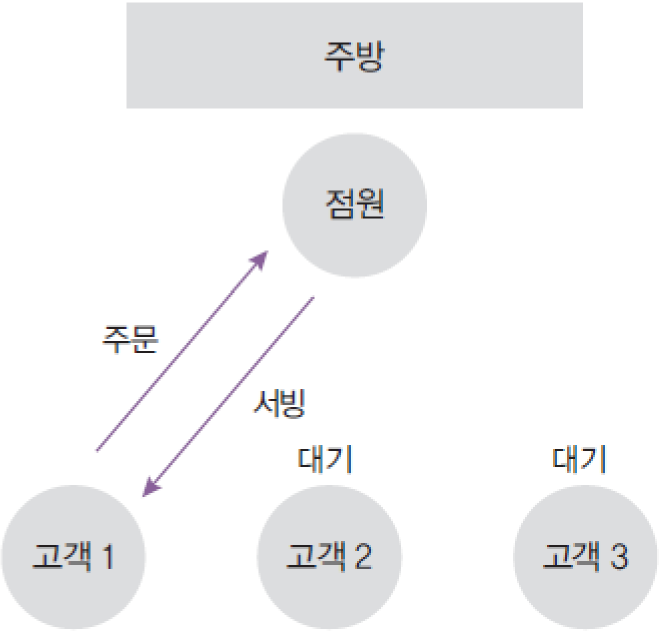


Single Thread

- Being single-threaded means it can handle only one task at a time.
  - In the presence of blocking, all other tasks must wait, leading to inefficiency.

- An analogy in a kitchen context (Waiter: Thread, Order: Request, Serving: Response):
  - One waiter (thread) takes one order (request) and serves (responds).
  - While serving, the waiter can't take new orders; others must wait

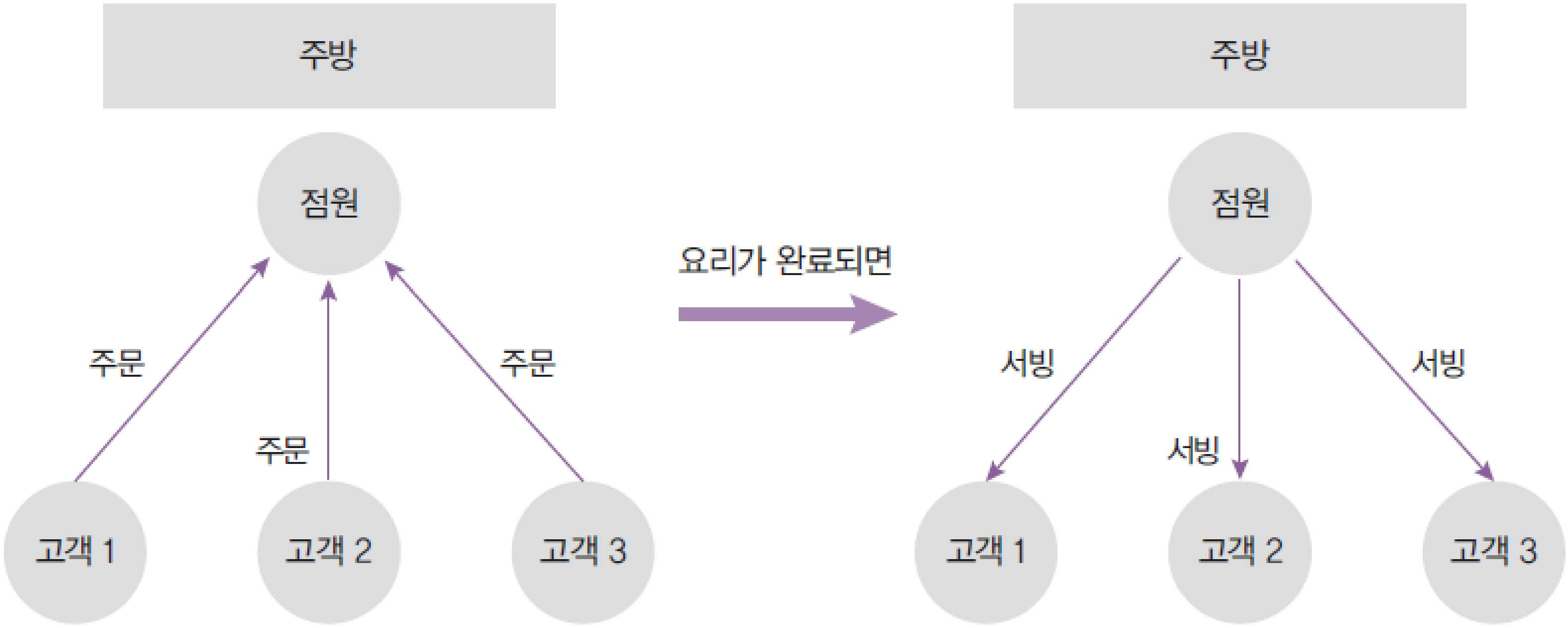
그림 1-10 싱글 스레드, 블로킹 모델



Single Thread

- nstead, adopting a non-blocking model allows certain code (I/O operations) to be executed in the background (in another process).
  - It receives requests first and responds when they are completed.
  - For code not related to I/O operations, it remains single-threaded, resembling a blocking model.

그림 1-11 싱글 스레드, 논블로킹 모델

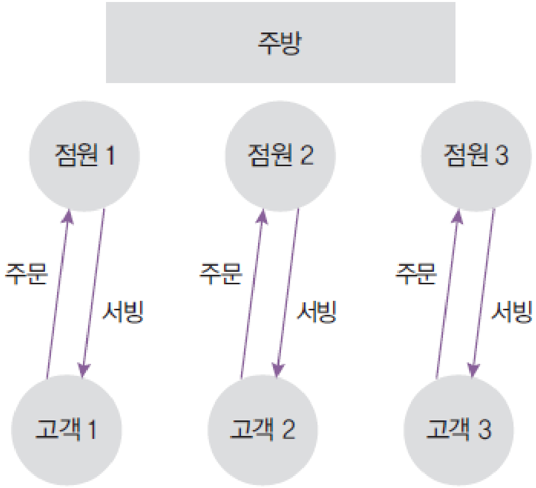


Single Thread

- In a single-threaded model, if an error occurs and is not handled properly, the entire process may come to a halt.
  - This model offers ease in programming and consumes fewer CPU and memory resources.

- On the other hand, the multi-threaded model overcomes errors by creating new threads when needed.
  - However, there are costs associated with creating new threads or managing idle threads.
  - This model presents greater programming complexity and utilizes more resources in proportion to the number of threads.
- Multi-threaded restaurant:
  - Errors can be handled by assigning a new waiter (creating a new thread).
  - However, managing multiple waiters (threads) comes with additional costs in terms of complexity and resource usage.

그림 1-12 멀티 스레드, 블로킹 모델



Single Thread

- In Node.js version 14, the introduction of the worker_threads module allows for the utilization of multi-threading.
  - This module is particularly useful for tasks that heavily involve CPU usage.
  - It addresses the limitation of only being able to use multi-processing in previous versions, providing a more versatile solution.

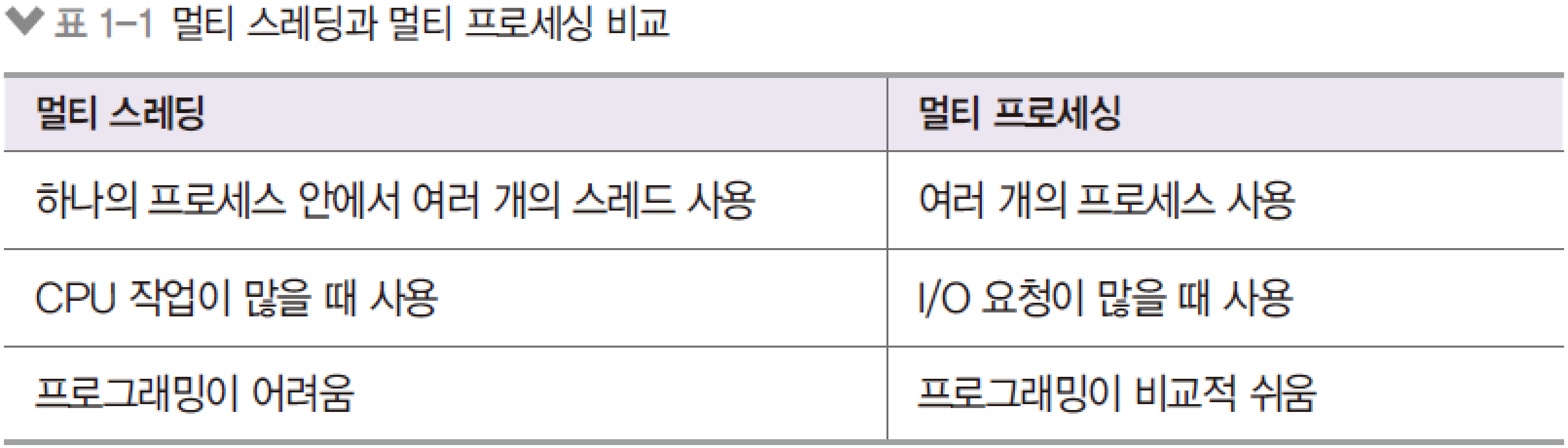


표 1-1 멀티 스레딩과 멀티 프로세싱 비교

멀티 스레딩	멀티 프로세싱
하나의 프로세스 안에서 여러 개의 스레드 사용	여러 개의 프로세스 사용
CPU 작업이 많을 때 사용	I/O 요청이 많을 때 사용
프로그래밍이 어려움	프로그래밍이 비교적 쉬움

Uses of node.js

Uses of node.js

- Node.js is not a server itself; however, it provides modules to facilitate server development.

- Pros and cons of a Node.js server
  - For CPU-intensive tasks, it might be advisable to use separate services like AWS Lambda or Google Cloud Functions.
  - Nevertheless, major companies such as PayPal, Netflix, NASA, Walmart, LinkedIn, and Uber use Node.js as either their main or sub-server.

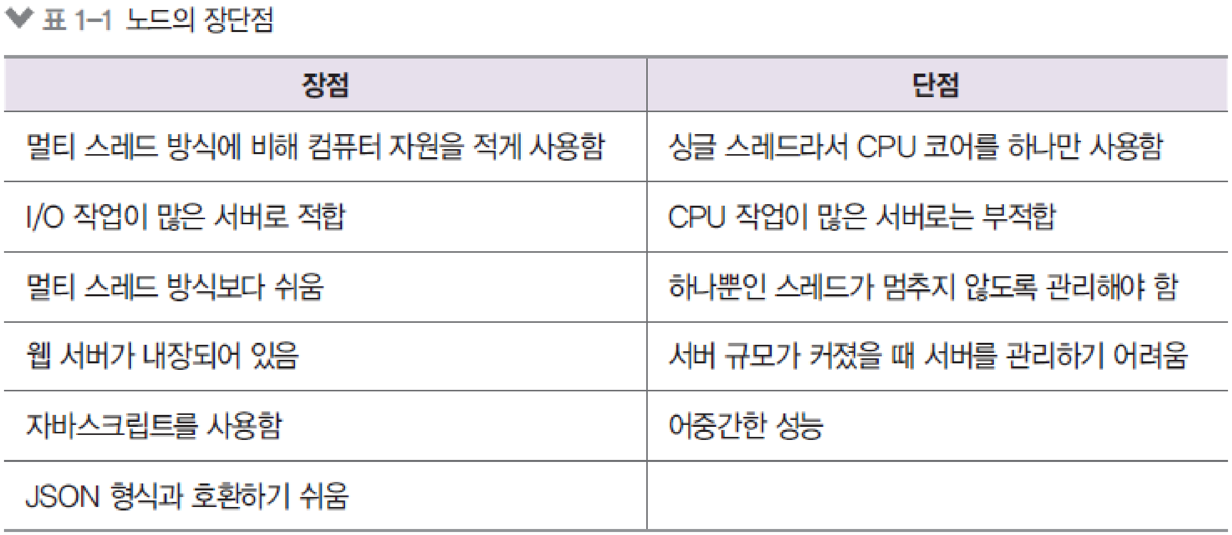
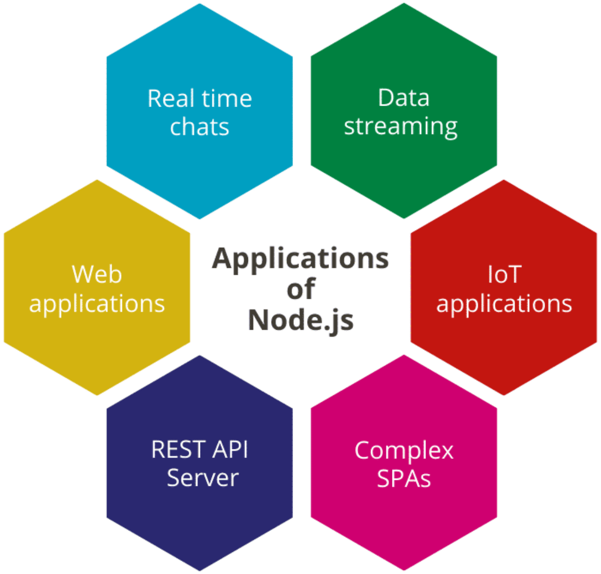


표 1-1 노드의 장단점

장점	단점
멀티 스레드 방식에 비해 컴퓨터 자원을 적게 사용함	싱글 스레드라서 CPU 코어를 하나만 사용함
I/O 작업이 많은 서버로 적합	CPU 작업이 많은 서버로는 부적합
멀티 스레드 방식보다 쉬움	하나뿐인 스레드가 멈추지 않도록 관리해야 함
웹 서버가 내장되어 있음	서버 규모가 커졌을 때 서버를 관리하기 어려움
자바스크립트를 사용함	어중간한 성능
JSON 형식과 호환하기 쉬움	

Uses of node.js

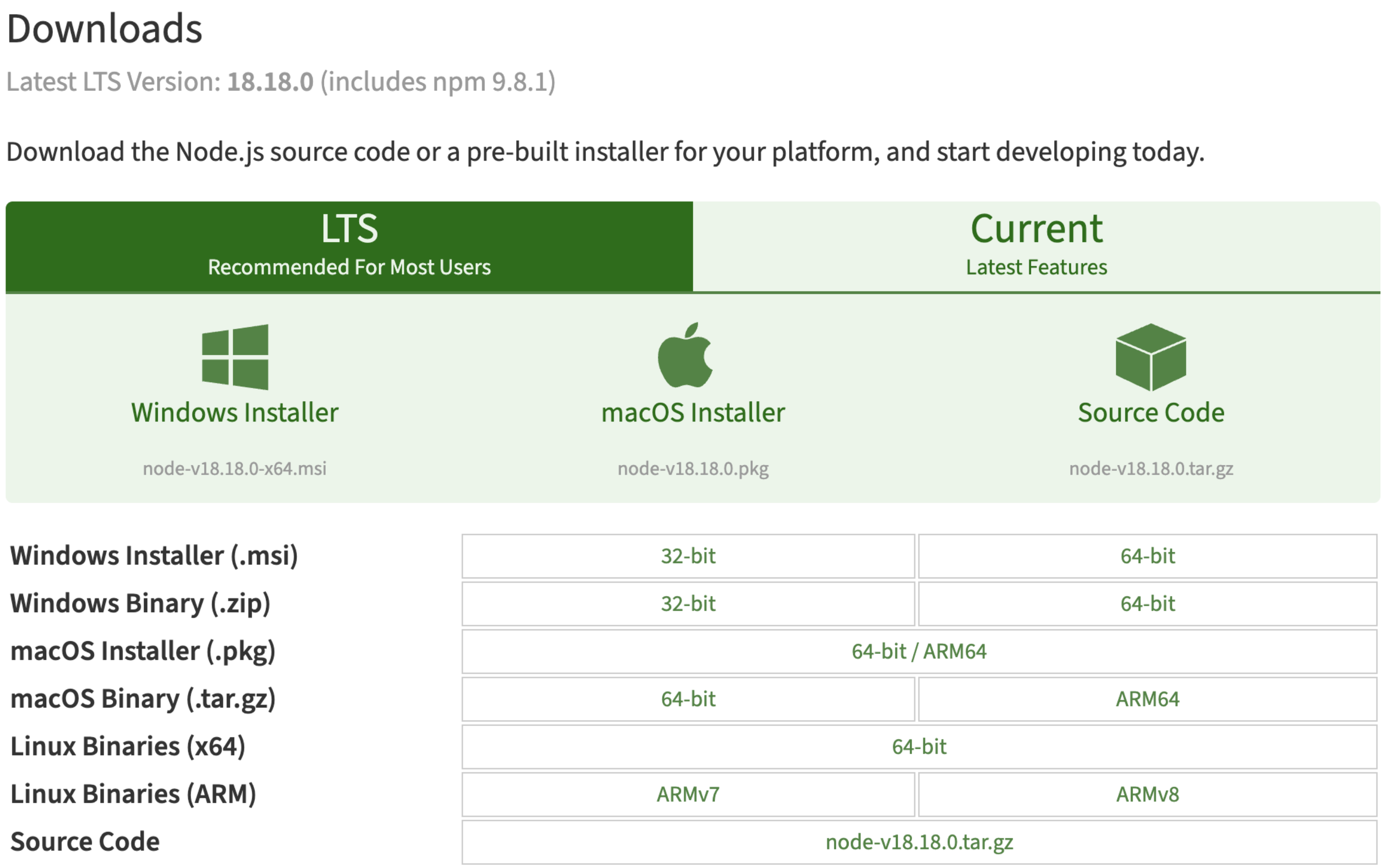
- Because it is a JavaScript runtime, Node.js is not limited to server-side applications. It is also used in web, mobile, and desktop applications.
  - Web Frameworks: Angular, React, Vue, Meteor, and others are popular web frameworks that operate based on Node.js.
  - Mobile App Framework: React Native is a mobile app framework that utilizes Node.js.
  - Desktop Development Tools: Electron, which powers applications like Atom, Slack, VSCode, Discord, and more, is a desktop development tool that is Node.js-based.
- These frameworks and tools leverage the versatility of Node.js, making it suitable for a wide range of applications beyond just server-side development.



Install node.js

Install node.js

- Download site
  - https://nodejs.org/en/download
  - Download LTS (node-v18) according to your OS



Downloads

Latest LTS Version: **18.18.0** (includes npm 9.8.1)

Download the Node.js source code or a pre-built installer for your platform, and start developing today.

LTS Recommended For Most Users	Current Latest Features	
Windows Installer node-v18.18.0-x64.msi	macOS Installer node-v18.18.0.pkg	Source Code node-v18.18.0.tar.gz

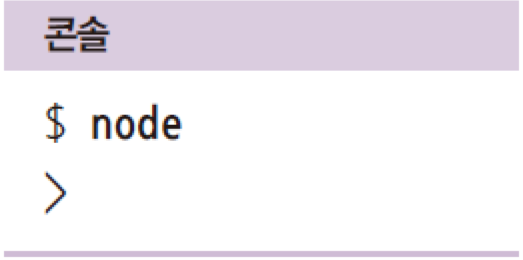
Windows Installer (.msi)	32-bit	64-bit
Windows Binary (.zip)	32-bit	64-bit
macOS Installer (.pkg)	64-bit / ARM64	
macOS Binary (.tar.gz)	64-bit	ARM64
Linux Binaries (x64)	64-bit	
Linux Binaries (ARM)	ARMv7	ARMv8
Source Code	node-v18.18.0.tar.gz	

How to use node.js

REPL

- JavaScript is a scripting language, allowing code to be executed on the fly. It provides a console called REPL (Read, Evaluate, Print, Loop).

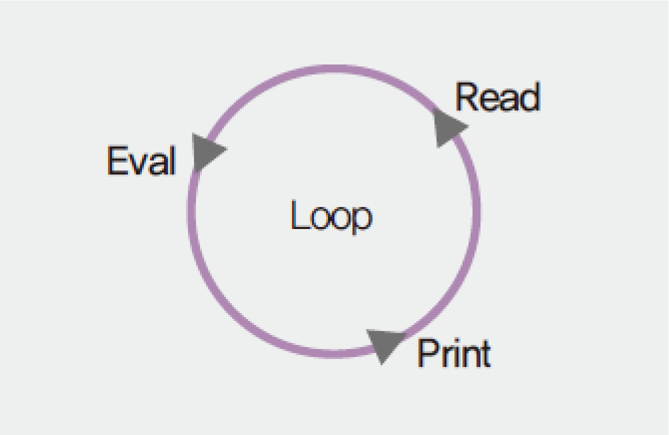
- In Windows, you can use the command prompt, and on macOS or Linux, you can use the terminal by entering 'node'.



콘솔

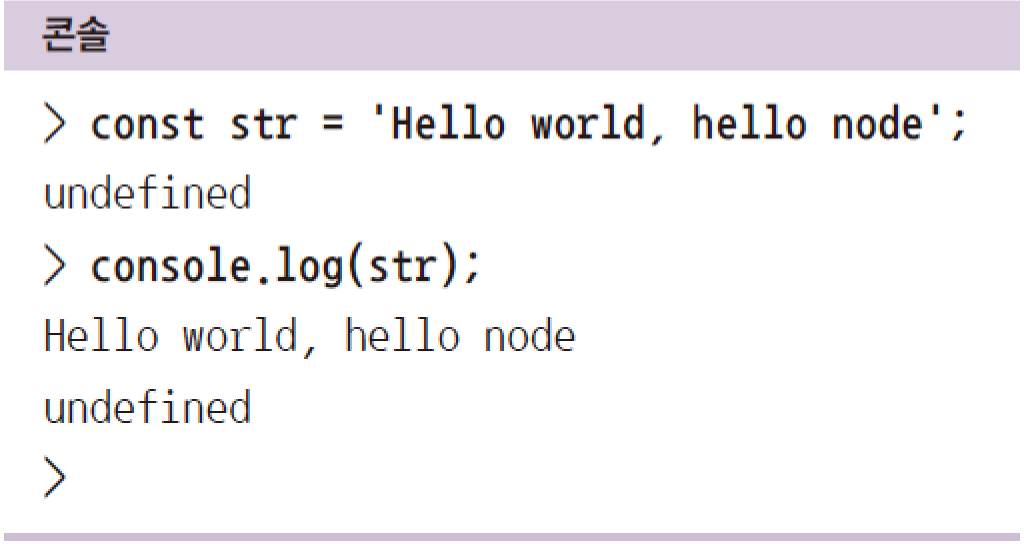
```
$ node
>
```

그림 3-1 REPL



REPL

- When the prompt changes to '>', you can input JavaScript code, and the result of the entered value is immediately displayed.
- It is suitable for testing short snippets of code but is not recommended for entering longer code.

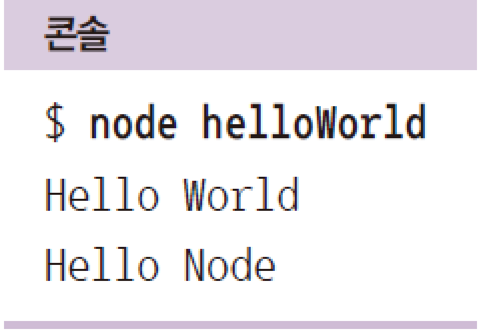


콘솔

```
> const str = 'Hello world, hello node';
undefined
> console.log(str);
Hello world, hello node
undefined
>
```

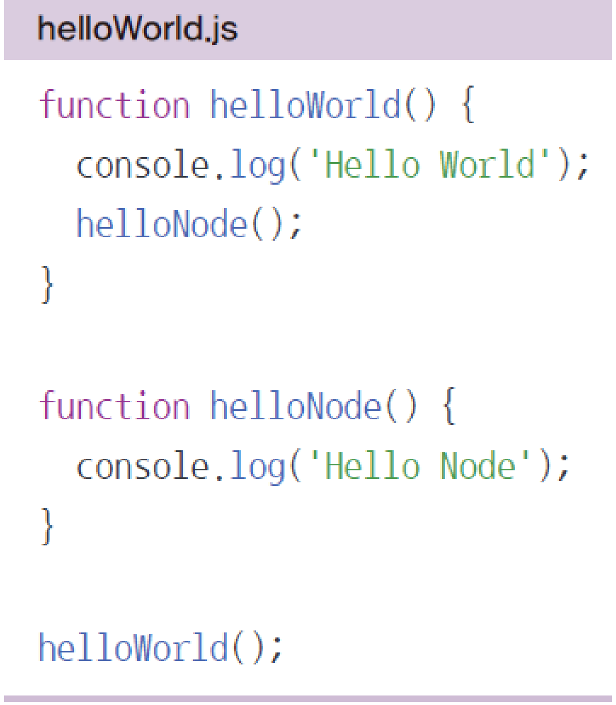
Executing JS file

- Creating a JavaScript file and running the entire code can be done using the following steps:
  - 1. Create a JavaScript file:
    - Open a text editor.
    - Create a file named helloWorld.js.
  - 2. Write the JavaScript code:
    - Inside helloWorld.js, write your JavaScript code.
  - 3. Save the file:
    - Save the helloWorld.js file.
  - 4. Open the terminal or command prompt:
    - Navigate to the directory where helloWorld.js is saved.
  - 5. Run the JavaScript file with Node.js:
    - node helloWorld.js
  - 6. View the output:
    - The output of the executed code, in this case, "Hello, World!", will be displayed in the terminal or command prompt.



콘솔

```
$ node helloWorld
Hello World
Hello Node
```



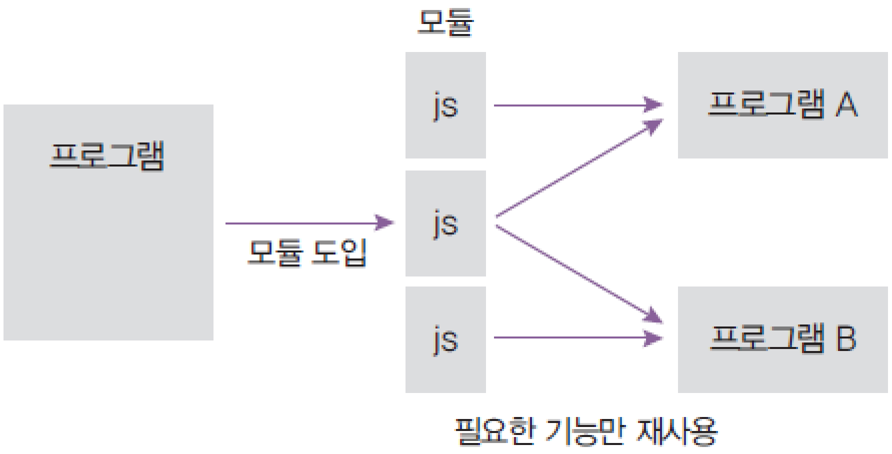
helloWorld.js

```
function helloWorld() {
  console.log('Hello World');
  helloNode();
}

function helloNode() {
  console.log('Hello Node');
}

helloWorld();
```

Module

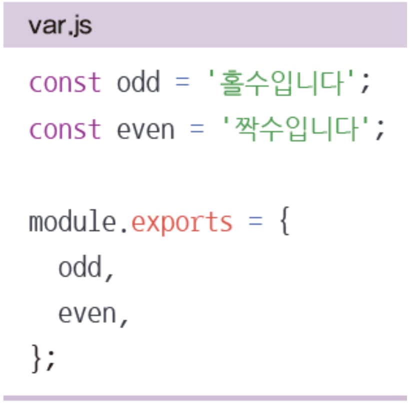
- Node.js allows you to create JavaScript code as modules.
  - A module is a collection of functions or variables that perform a specific set of tasks.
  - By creating code as a module, you enable its reuse across multiple programs.

그림 3-2 모듈과 프로그램



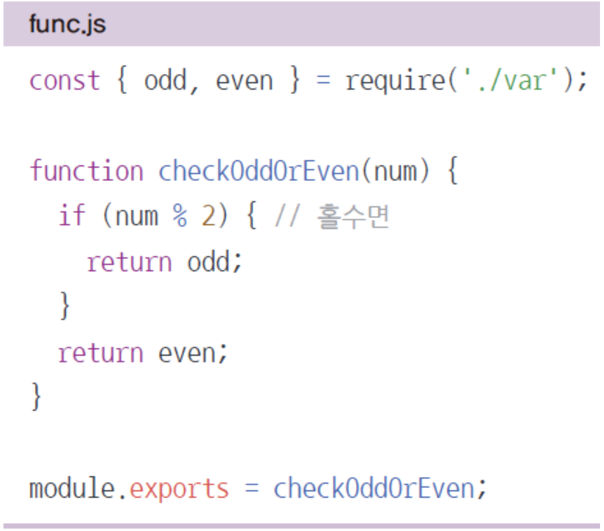
Creating module

- Creating var.js, func.js, and index.js in the same folder:
  - Specify values to be turned into modules using **module.exports** at the end of the file.
  - You can then use **require(file path)** in other files to import the contents of that module.



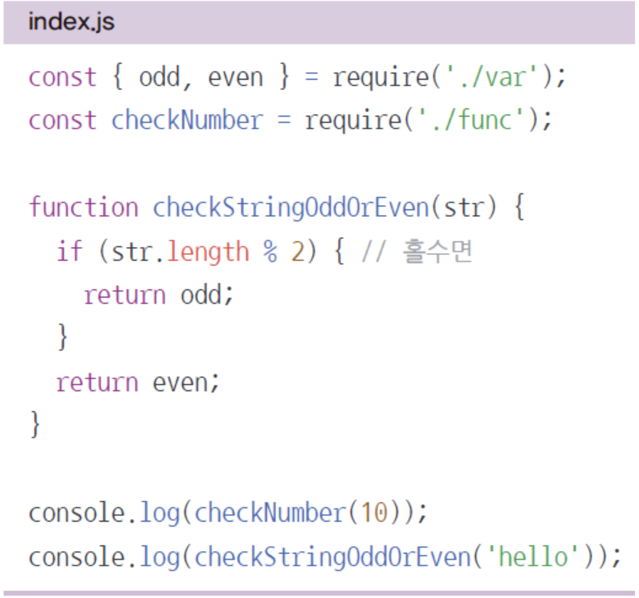
var.js

```
const odd = '홀수입니다';
const even = '짝수입니다';

module.exports = {
  odd,
  even,
};
```



func.js

```
const { odd, even } = require('./var');

function checkOddOrEven(num) {
  if (num % 2) { // 홀수면
    return odd;
  }
  return even;
}

module.exports = checkOddOrEven;
```



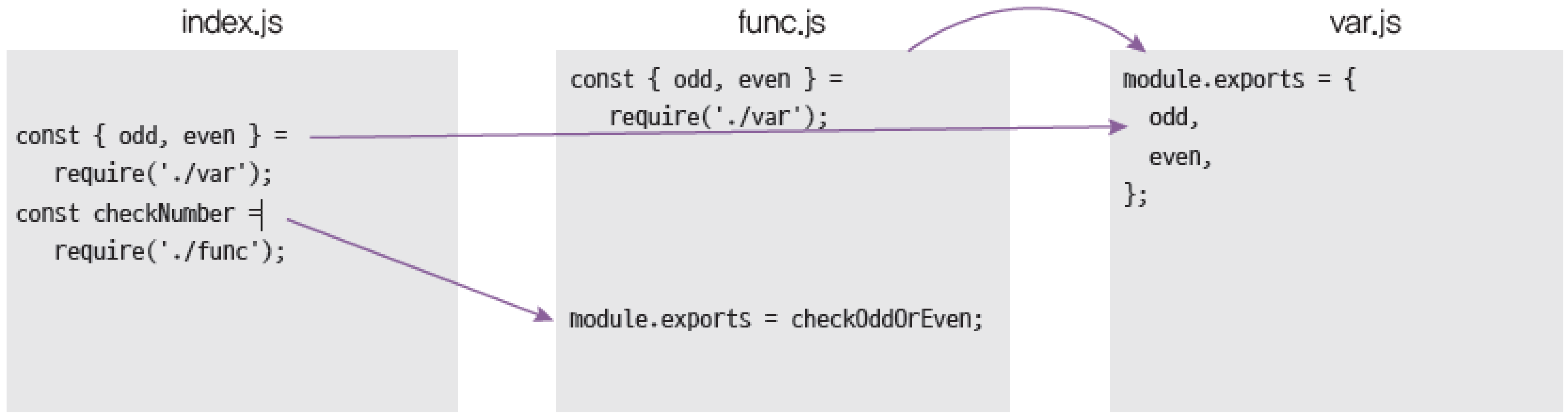
index.js

```
const { odd, even } = require('./var');
const checkNumber = require('./func');

function checkStringOddOrEven(str) {
  if (str.length % 2) { // 홀수면
    return odd;
  }
  return even;
}

console.log(checkNumber(10));
console.log(checkStringOddOrEven('hello'));
```

파일 간의 모듈 관계

- run the code using ‘node index’
  - the given code snippet uses destructuring assignment on module.exports

그림 3-3 require와 module.exports



index.js

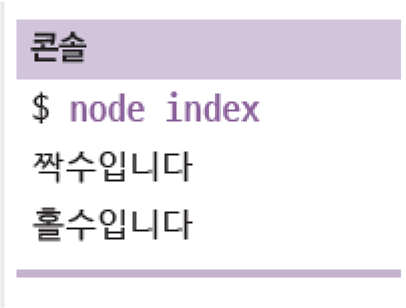
```
const { odd, even } =
   require('./var');
const checkNumber =
   require('./func');
```

func.js

```
const { odd, even } =
   require('./var');

module.exports = checkOddOrEven;
```

var.js

```
module.exports = {
  odd,
  even,
};
```

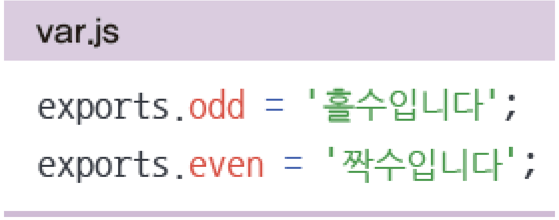


콘솔

```
$ node index
짝수입니다
홀수입니다
```

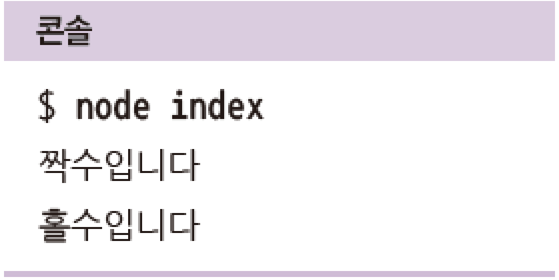
module, exports

- You can create modules using exports in addition to module.exports.
  - After modifying the var.js example in the module sample as follows, it will still work.
  - This is because module.exports and exports refer to the same object.
  - If you assign a value other than an object property to exports, the reference relationship will break.



var.js

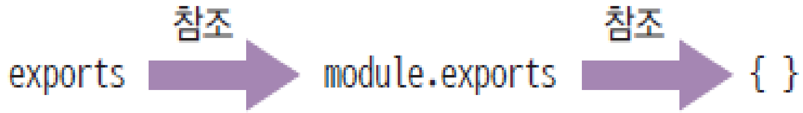
```
exports.odd = '홀수입니다';
exports.even = '짝수입니다';
```



콘솔

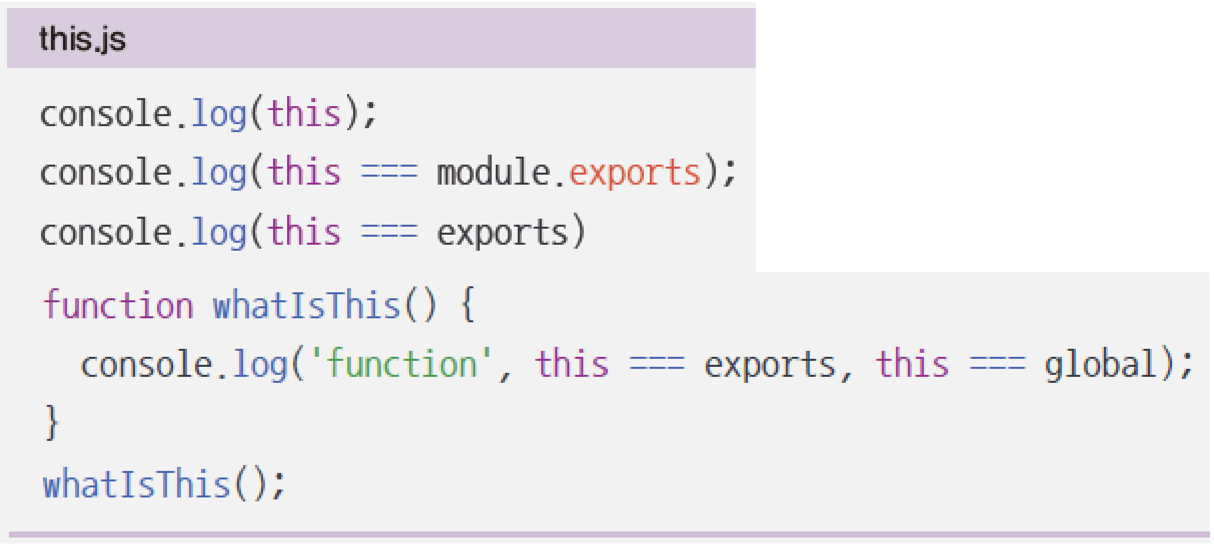
```
$ node index
짝수입니다
홀수입니다
```

▼ 그림 3-5 exports와 module.exports의 관계



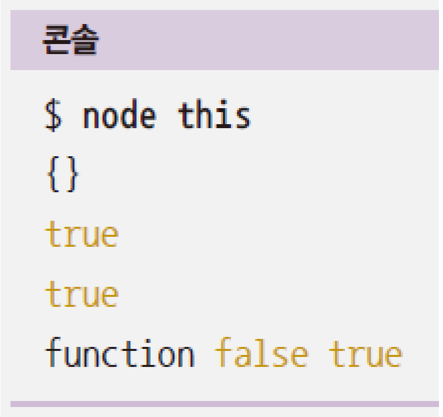
this

- When using this in Node.js, there are some considerations.
  - The this in the top-level scope refers to module.exports.
  - Otherwise, it behaves similarly to JavaScript in the browser.
  - Within the function declaration, this refers to the global object.



this.js

```
console.log(this);
console.log(this === module.exports);
console.log(this === exports)
function whatIsThis() {
  console.log('function', this === exports, this === global);
}
whatIsThis();
```

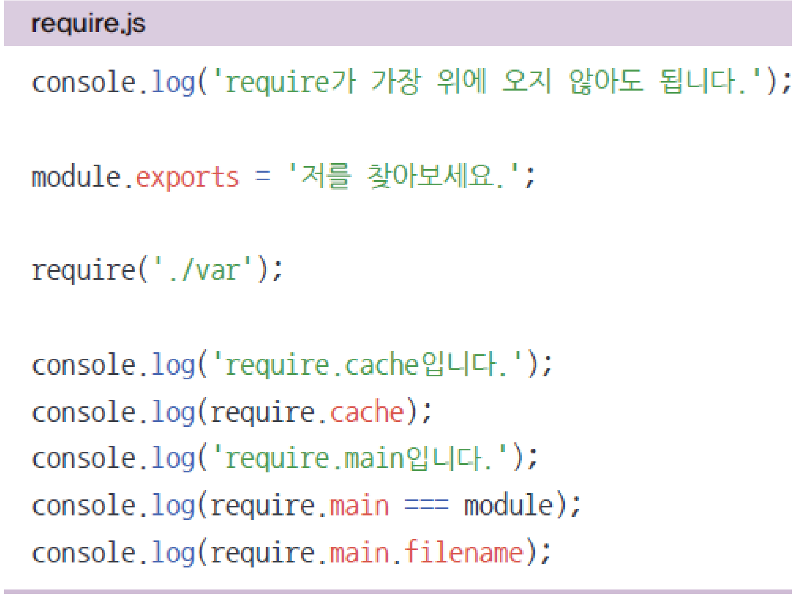


콘솔

```
$ node this
{}
true
true
function false true
```

require

- There are several important properties to be aware of:
  - **require Order:**
    - It's not necessary for require to be at the top of the file. It can be placed anywhere in the code.
  - **require.cache:**
    - The require.cache object contains cached information about modules that have been required.
  - **require.main:**
    - require.main refers to the main module that was initially run by Node.js. This can be used to determine if the script is being run directly or if it is being required as a module into another script.

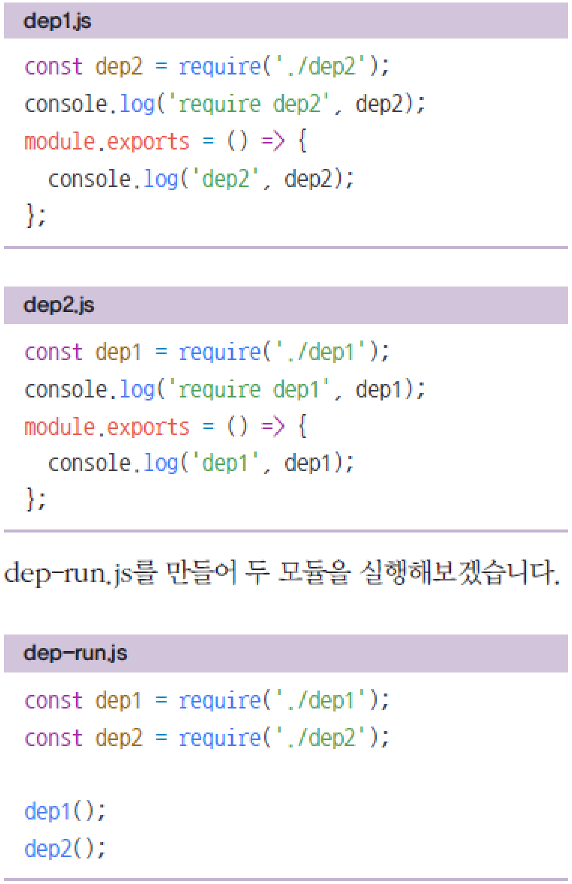


require.js

```
console.log('require가 가장 위에 오지 않아도 됩니다.');

module.exports = '저를 찾아보세요.';

require('./var');

console.log('require.cache입니다.');
console.log(require.cache);
console.log('require.main입니다.');
console.log(require.main === module);
console.log(require.main.filename);
```

circular reference between modules

- If module A requires module B, and module B requires module A in return, a circular reference is created.
  - To prevent an infinite loop, Node.js resolves the circular dependency by assigning an empty object as a placeholder for the circular reference.



dep1.js

```
const dep2 = require('./dep2');
console.log('require dep2', dep2);
module.exports = () => {
  console.log('dep2', dep2);
};
```

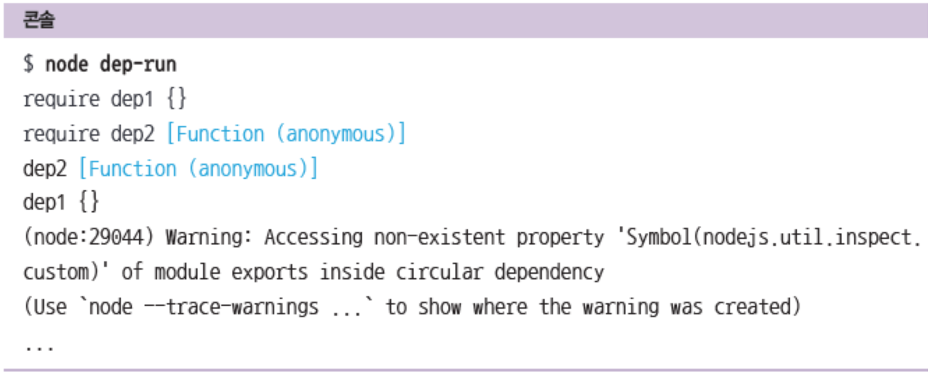
dep2.js

```
const dep1 = require('./dep1');
console.log('require dep1', dep1);
module.exports = () => {
  console.log('dep1', dep1);
};
```

dep-run.js를 만들어 두 모듈을 실행해보겠습니다.

dep-run.js

```
const dep1 = require('./dep1');
const dep2 = require('./dep2');

dep1();
dep2();
```

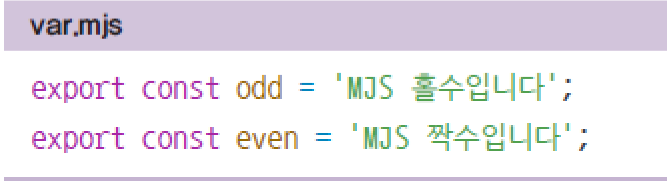


콘솔

```
$ node dep-run
require dep1 {}
require dep2 [Function (anonymous)]
dep2 [Function (anonymous)]
dep1 {}
(node:29044) Warning: Accessing non-existent property 'Symbol(nodejs.util.inspect.
custom)' of module exports inside circular dependency
(Use `node --trace-warnings ...` to show where the warning was created)
...
```

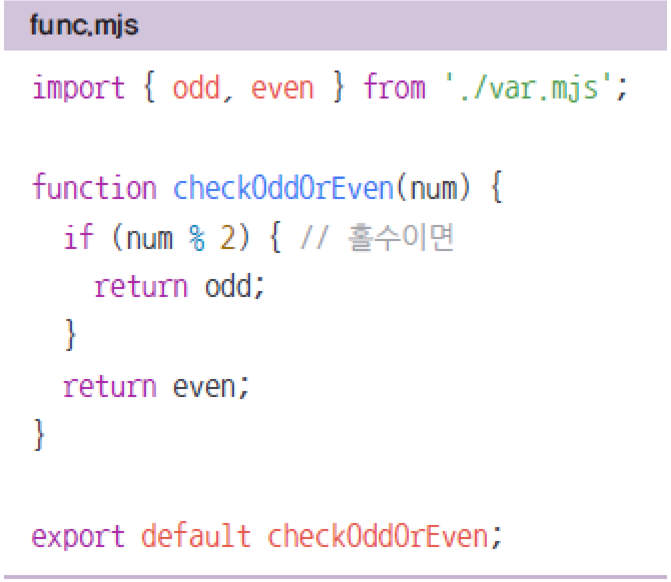
ES module

- avaScript itself introduces a module system syntax.
  - You need to use the .mjs extension or add type: "module" to package.json.
  - The syntax changes include using import instead of require, export default instead of module.exports, and export instead of exports.



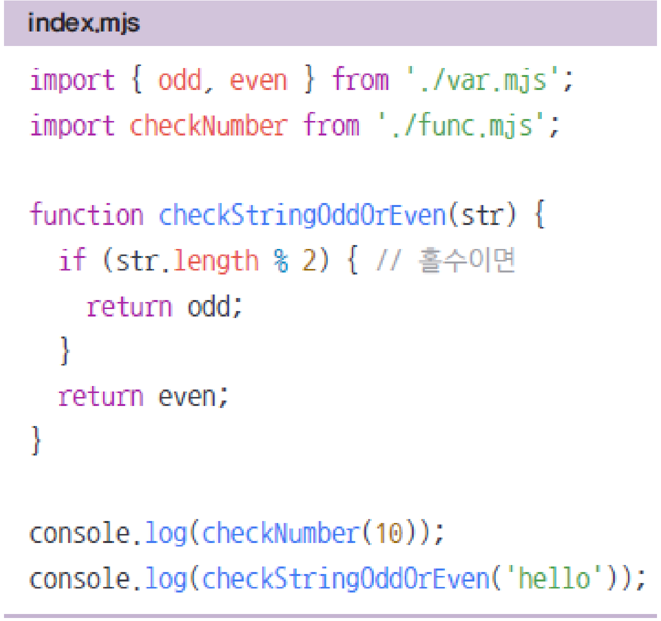
var.mjs

```
export const odd = 'MJS 홀수입니다';
export const even = 'MJS 짝수입니다';
```



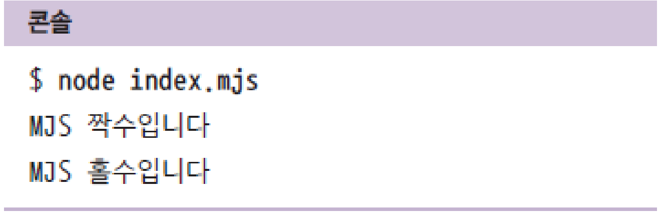
func.mjs

```
import { odd, even } from './var.mjs';

function checkOddOrEven(num) {
  if (num % 2) { // 홀수이면
    return odd;
  }
  return even;
}

export default checkOddOrEven;
```



index.mjs

```
import { odd, even } from './var.mjs';
import checkNumber from './func.mjs';

function checkStringOddOrEven(str) {
  if (str.length % 2) { // 홀수이면
    return odd;
  }
  return even;
}

console.log(checkNumber(10));
console.log(checkStringOddOrEven('hello'));
```



콘솔

```
$ node index.mjs
MJS 짝수입니다
MJS 홀수입니다
```

Differences between CommonJS and ES module

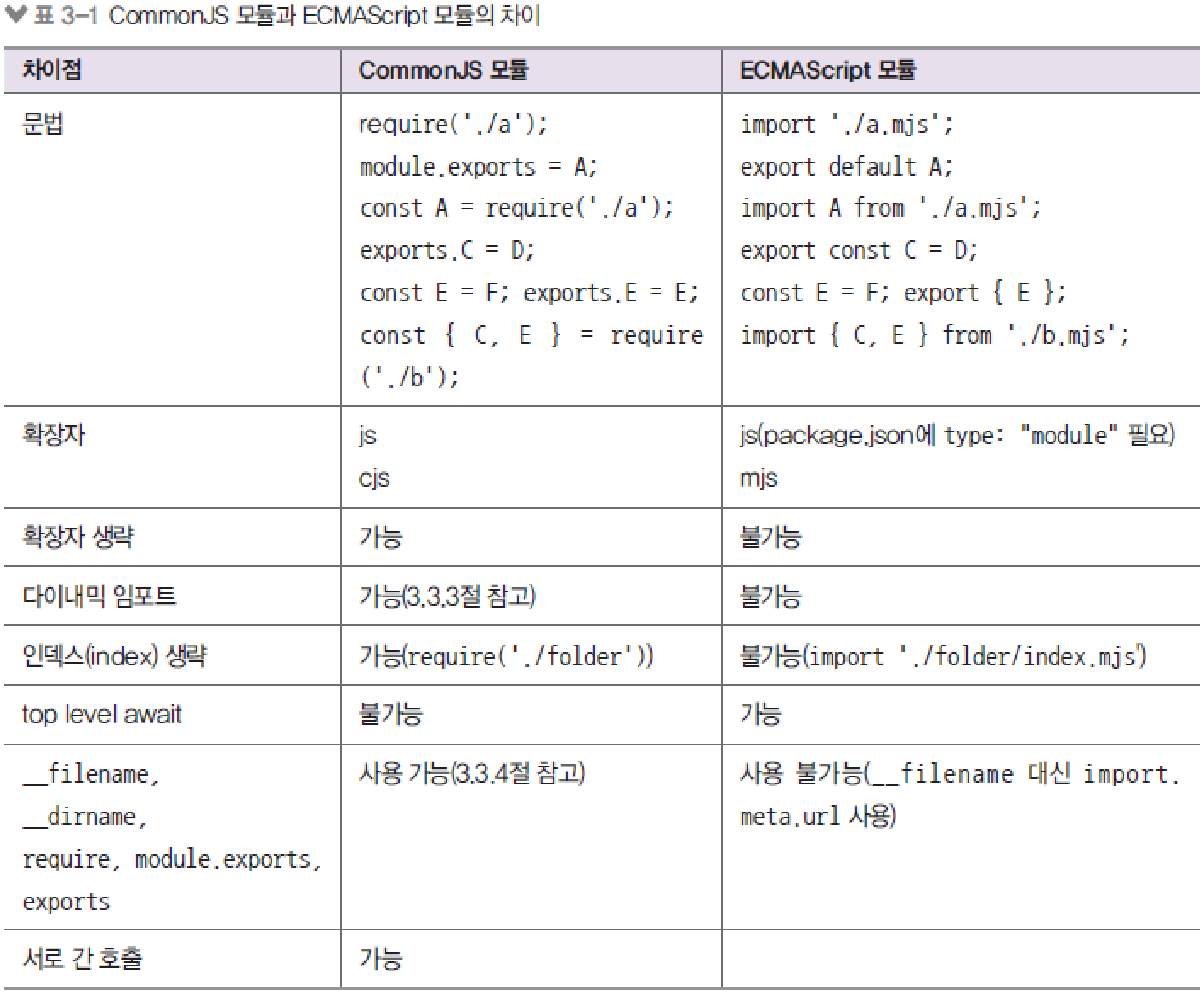


표 3-1 CommonJS 모듈과 ECMAScript 모듈의 차이

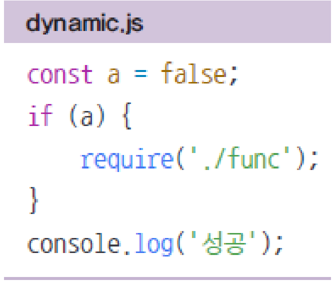
차이점	CommonJS 모듈	ECMAScript 모듈

문법	`require('./a');`
	`module.exports = A;`
	`const A = require('./a');`
	`exports.C = D;`
	`const E = F; exports.E = E;`
	`const { C, E } = require('./b');`
		`import './a.mjs';`
		`export default A;`
		`import A from './a.mjs';`
		`export const C = D;`
		`const E = F; export { E };`
		`import { C, E } from './b.mjs';`

확장자	js cjs	js(package.json에 type: "module" 필요) mjs
확장자 생략	가능	불가능
다이내믹 임포트	가능(3.3.3절 참고)	불가능
인덱스(index) 생략	가능(require('./folder'))	불가능(import './folder/index.mjs')
top level await	불가능	가능
__filename, __dirname, require, module.exports, exports	사용 가능(3.3.4절 참고)	사용 불가능(__filename 대신 import.meta.url 사용)
서로 간 호출	가능	

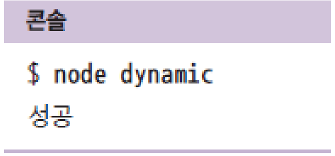
Dynamic Import, top level await

- You can dynamically load in the middle of the code
  - In CommonJS, use require():
  - In ES Modules, use import():
- In mjs files, you can use await without async at the top level:



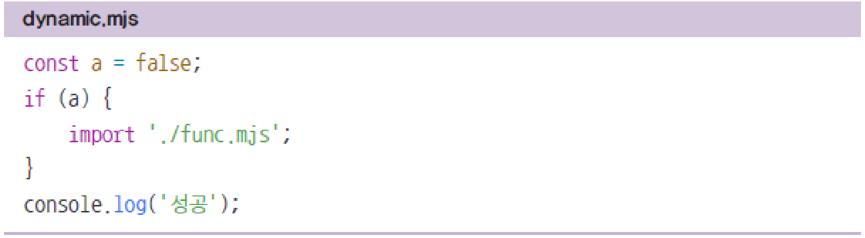
dynamic.js

```
const a = false;
if (a) {
    require('./func');
}
console.log('성공');
```



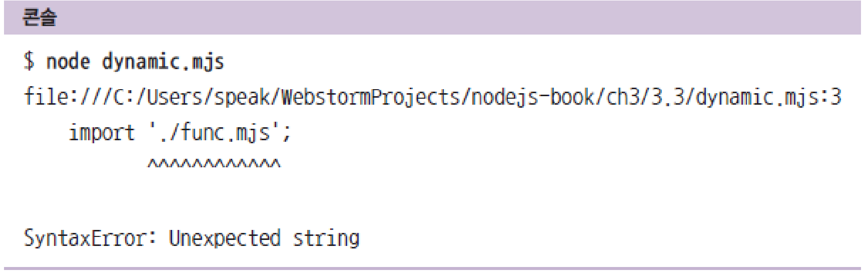
콘솔

```
$ node dynamic
성공
```



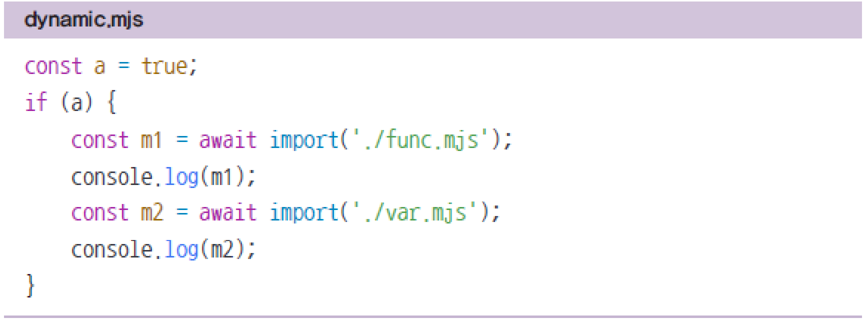
dynamic.mjs

```
const a = false;
if (a) {
    import './func.mjs';
}
console.log('성공');
```



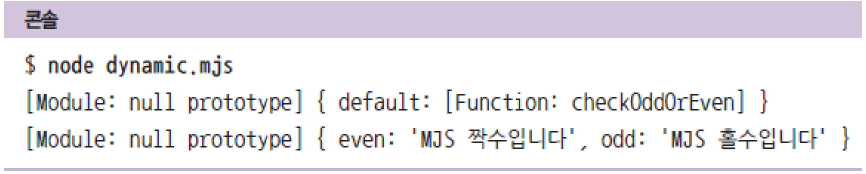
콘솔

```
$ node dynamic.mjs
file:///C:/Users/speak/WebstormProjects/nodejs-book/ch3/3.3/dynamic.mjs:3
    import './func.mjs';
           ^^^^^^^^^^^^


SyntaxError: Unexpected string
```



dynamic.mjs

```
const a = true;
if (a) {
    const m1 = await import('./func.mjs');
    console.log(m1);
    const m2 = await import('./var.mjs');
    console.log(m2);
}
```

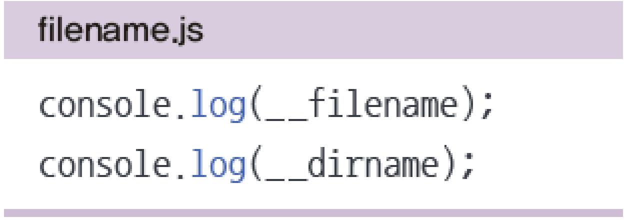


콘솔

```
$ node dynamic.mjs
[Module: null prototype] { default: [Function: checkOddOrEven] }
[Module: null prototype] { even: 'MJS 짝수입니다', odd: 'MJS 홀수입니다' }
```

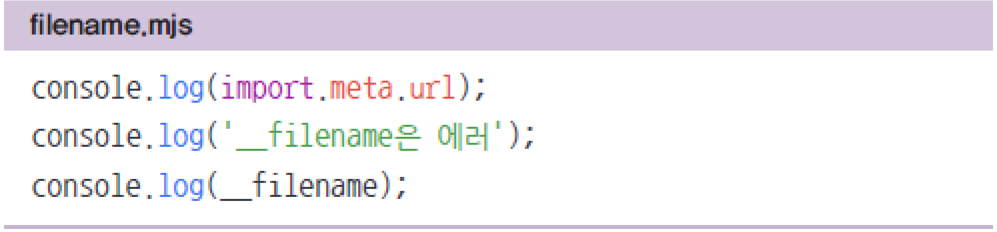
__filename, __dirname

- __filename: Current file path.
- __dirname: Current folder (directory) path.
- In ES Modules, you cannot use the above, and instead, you should use import.meta.url.



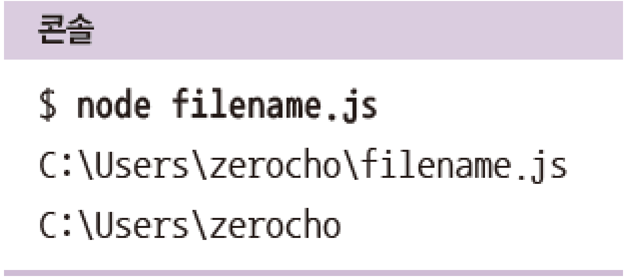
filename.js

```
console.log(__filename);
console.log(__dirname);
```



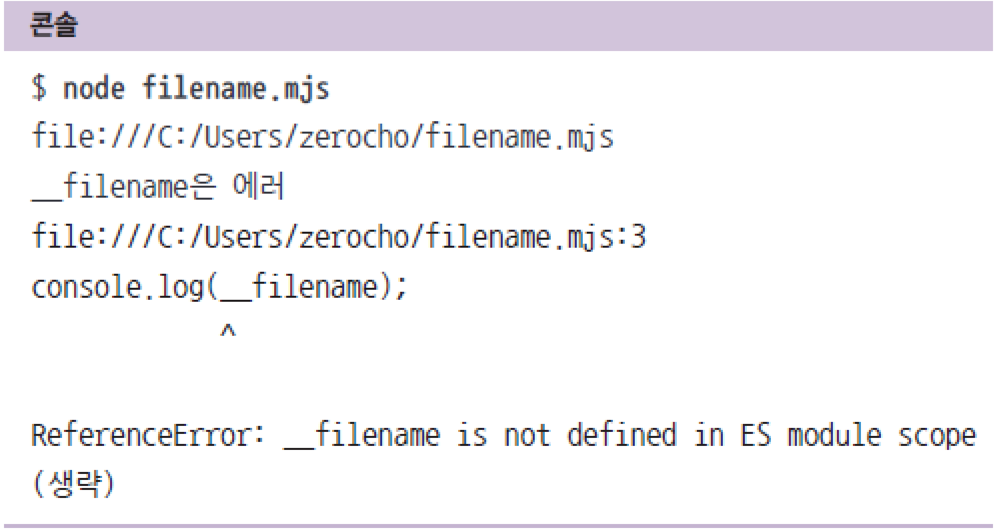
filename.mjs

```
console.log(import.meta.url);
console.log('__filename은 에러');
console.log(__filename);
```



콘솔

```
$ node filename.js
C:\Users\zerocho\filename.js
C:\Users\zerocho
```



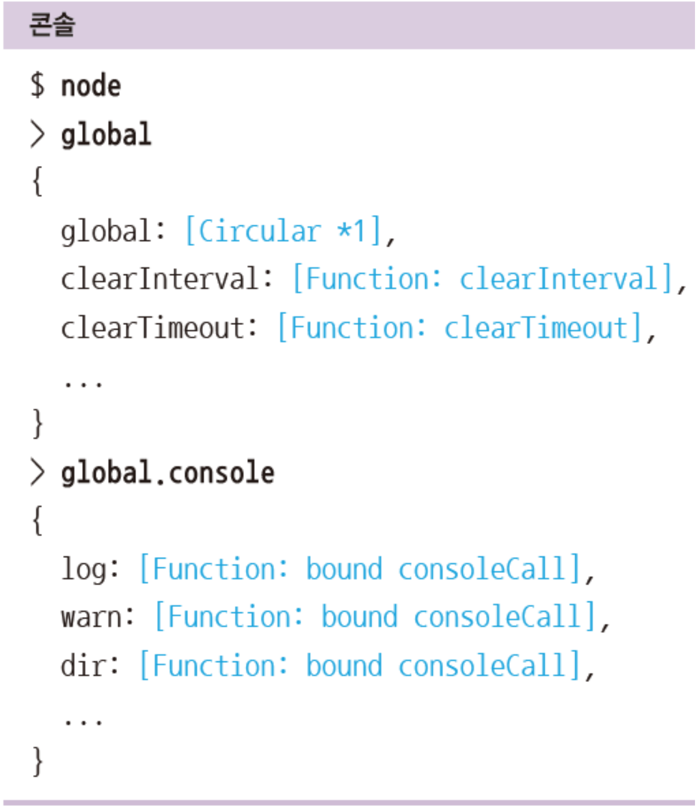
콘솔

```
$ node filename.mjs
file:///C:/Users/zerocho/filename.mjs
__filename은 에러
file:///C:/Users/zerocho/filename.mjs:3
console.log(__filename);
            ^

ReferenceError: __filename is not defined in ES module scope
(생략)
```

Node.js built-in objects

global

- The global object in Node.js plays a role similar to the window object in the browser.
  - It is accessible from every file.
  - Just like in the browser, you can omit global and directly access properties like console and require as they are attributes of the global object.

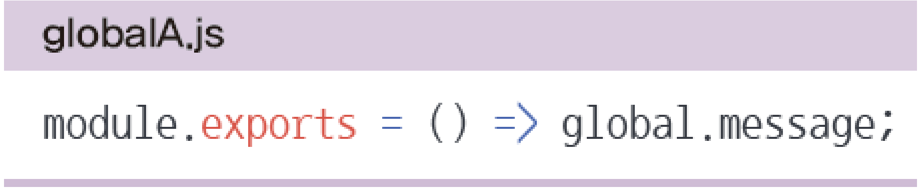


콘솔

```
$ node
> global
{
  global: [Circular *1],
  clearInterval: [Function: clearInterval],
  clearTimeout: [Function: clearTimeout],
  ...
}
> global.console
{
  log: [Function: bound consoleCall],
  warn: [Function: bound consoleCall],
  dir: [Function: bound consoleCall],
  ...
}
```

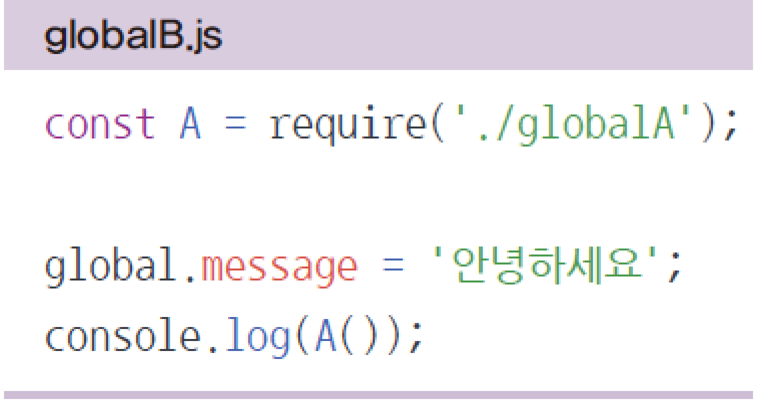
global

- If you assign a value to a property of the global object, it becomes accessible in other files as well.



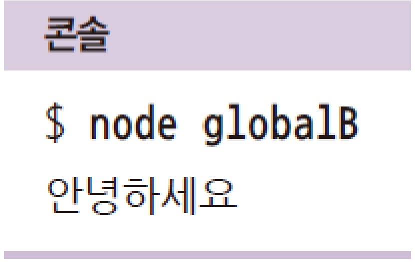
globalA.js

```
module.exports = () => global.message;
```



globalB.js

```
const A = require('./globalA');

global.message = '안녕하세요';
console.log(A());
```

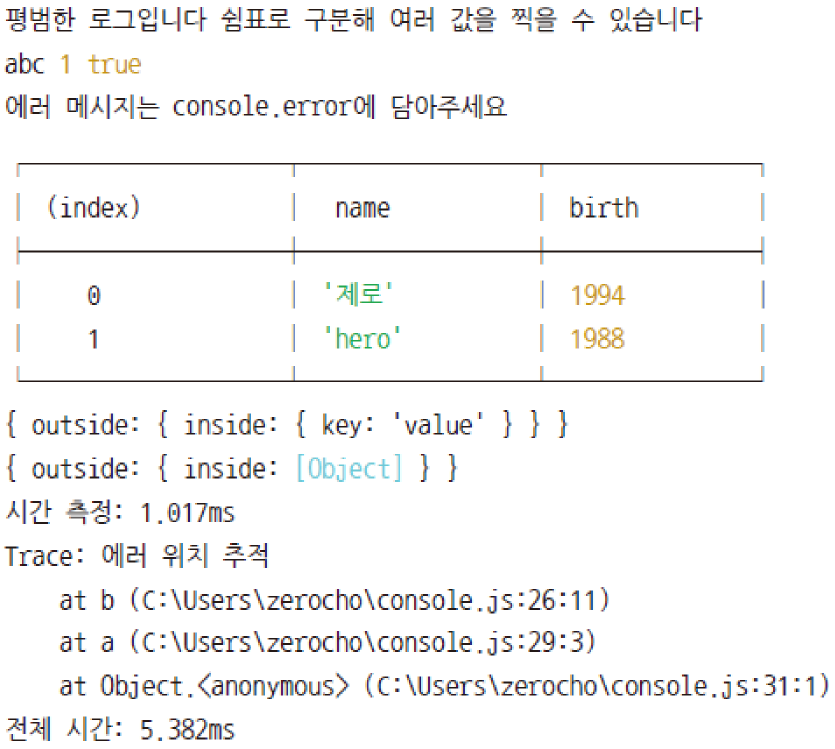


콘솔

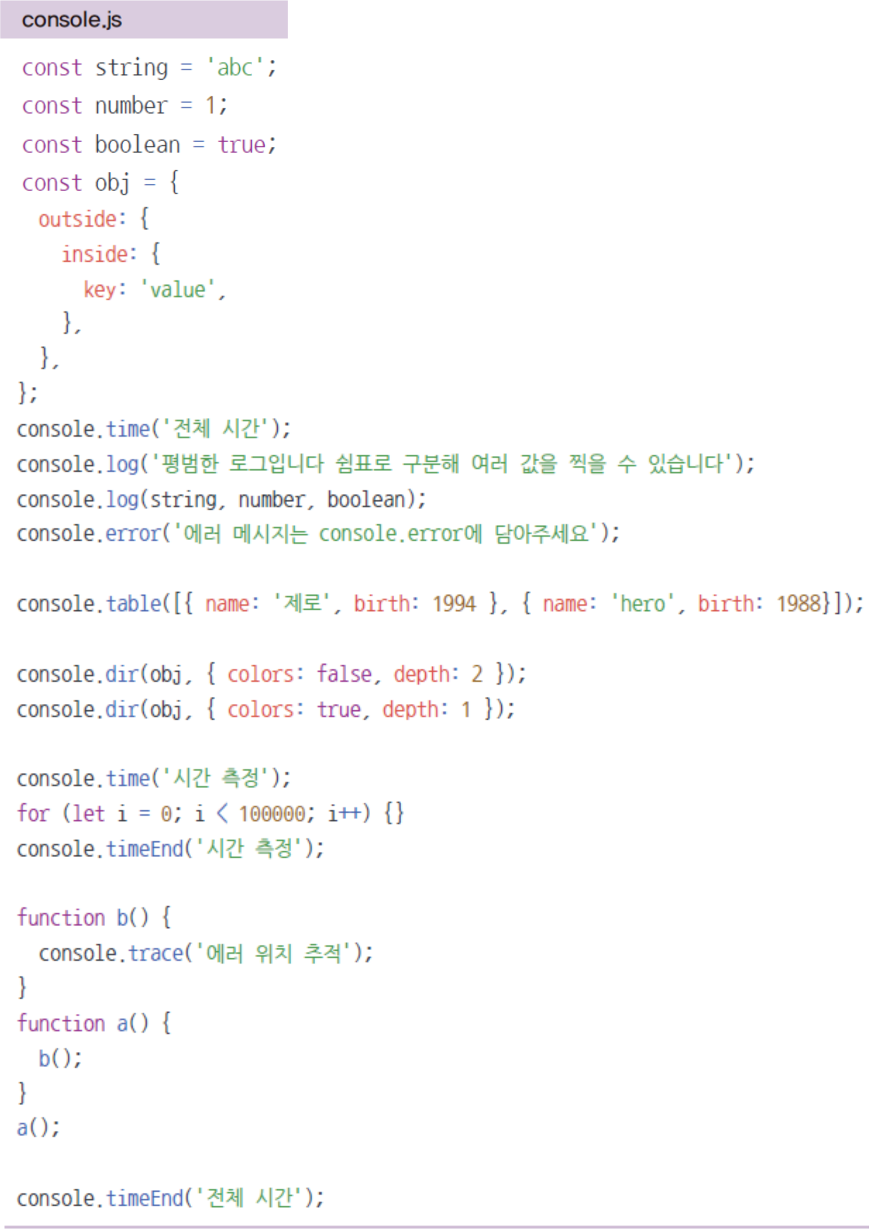
```
$ node globalB
안녕하세요
```

console object

- console object
  - console.time, console.timeEnd: Time logging.
  - console.error: Error logging.
  - console.log: Regular logging.
  - console.dir: Object logging.
  - console.trace: Call stack logging.



```
평범한 로그입니다 쉼표로 구분해 여러 값을 찍을 수 있습니다
abc 1 true
에러 메시지는 console.error에 담아주세요

┌─────────┬────────┬──────┐
│ (index) │  name  │ birth│
├─────────┼────────┼──────┤
│    0    │ '제로' │ 1994 │
│    1    │ 'hero' │ 1988 │
└─────────┴────────┴──────┘
{ outside: { inside: { key: 'value' } } }
{ outside: { inside: [Object] } }
시간 측정: 1.017ms
Trace: 에러 위치 추적
    at b (C:\Users\zerocho\console.js:26:11)
    at a (C:\Users\zerocho\console.js:29:3)
    at Object.<anonymous> (C:\Users\zerocho\console.js:31:1)
전체 시간: 5.382ms
```



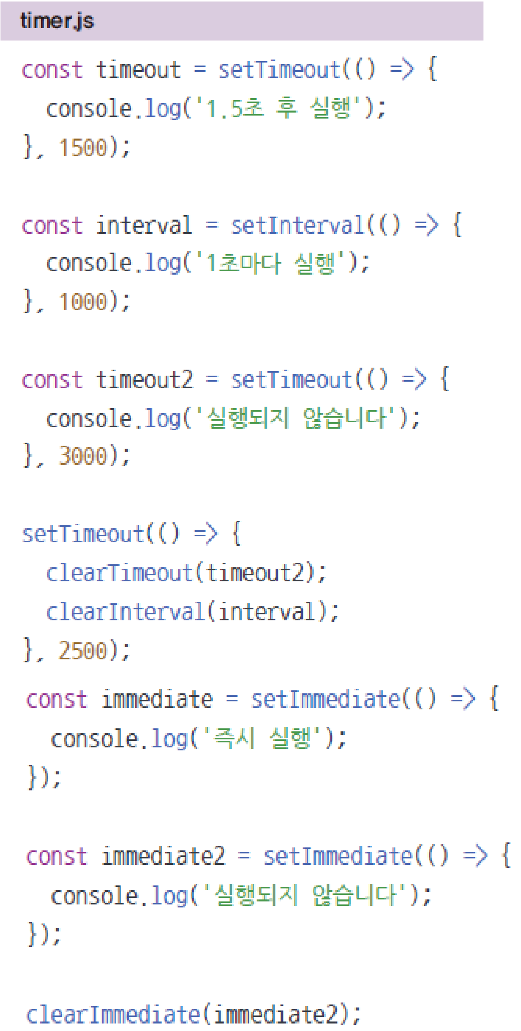
console.js

```
const string = 'abc';
const number = 1;
const boolean = true;
const obj = {
  outside: {
    inside: {
      key: 'value',
    },
  },
};
console.time('전체 시간');
console.log('평범한 로그입니다 쉼표로 구분해 여러 값을 찍을 수 있습니다');
console.log(string, number, boolean);
console.error('에러 메시지는 console.error에 담아주세요');

console.table([{ name: '제로', birth: 1994 }, { name: 'hero', birth: 1988}]);

console.dir(obj, { colors: false, depth: 2 });
console.dir(obj, { colors: true, depth: 1 });

console.time('시간 측정');
for (let i = 0; i < 100000; i++) {}
console.timeEnd('시간 측정');

function b() {
  console.trace('에러 위치 추적');
}
function a() {
  b();
}
a();

console.timeEnd('전체 시간');
```

timer method

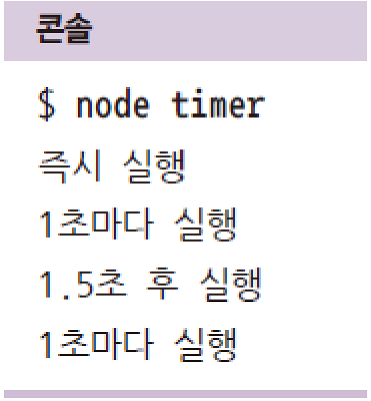
- The clear methods correspond to the set methods:
- For the set method, the return value (ID) is used with the corresponding clear method to cancel it.
  - setTimeout(callback, milliseconds): Executes the callback function after the given number of milliseconds (1/1000 of a second).
  - setInterval(callback, milliseconds): Repeatedly executes the callback function at the specified interval in milliseconds.
  - setImmediate(callback): Executes the callback function immediately.
- The clear methods are used to cancel the scheduled actions:
  - clearTimeout(id): Cancels the scheduled setTimeout.
  - clearInterval(id): Cancels the scheduled setInterval.
  - clearImmediate(id): Cancels the scheduled setImmediate.

timer example



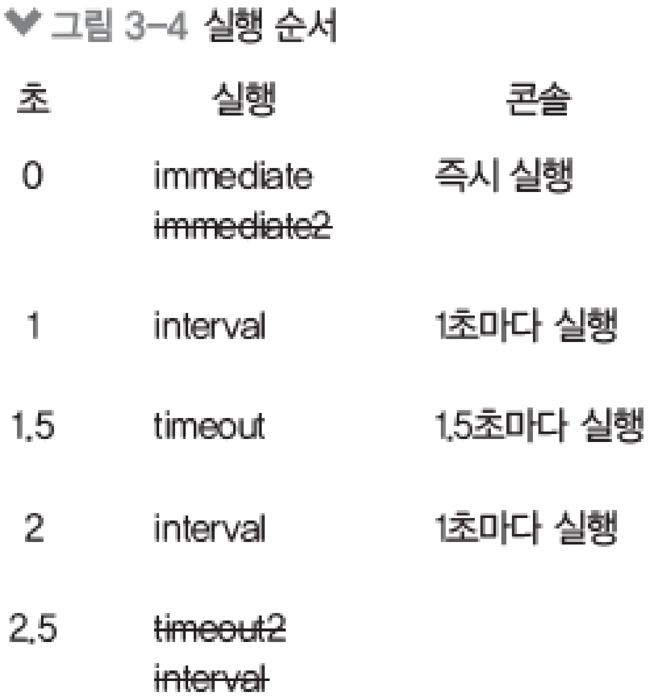
timer.js

```
const timeout = setTimeout(() => {
  console.log('1.5초 후 실행');
}, 1500);

const interval = setInterval(() => {
  console.log('1초마다 실행');
}, 1000);

const timeout2 = setTimeout(() => {
  console.log('실행되지 않습니다');
}, 3000);

setTimeout(() => {
  clearTimeout(timeout2);
  clearInterval(interval);
}, 2500);
const immediate = setImmediate(() => {
  console.log('즉시 실행');
});

const immediate2 = setImmediate(() => {
  console.log('실행되지 않습니다');
});

clearImmediate(immediate2);
```



콘솔

```
$ node timer
즉시 실행
1초마다 실행
1.5초 후 실행
1초마다 실행
```

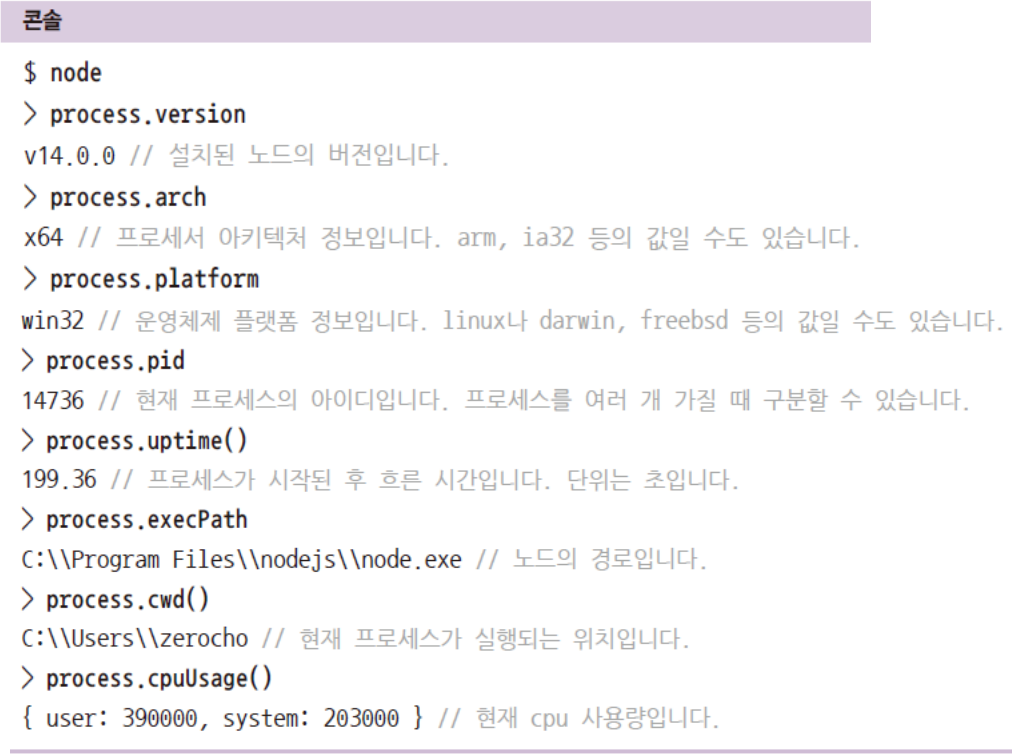


▼ 그림 3-4 실행 순서

초	실행	콘솔
0	immediate ~~immediate2~~	즉시 실행
1	interval	1초마다 실행
1.5	timeout	1.5초마다 실행
2	interval	1초마다 실행
2.5	~~timeout2~~ ~~interval~~	

process

- It contains information about the currently running Node.js process.
  - The output may vary between computers and can differ from what is shown in a presentation (PPT).

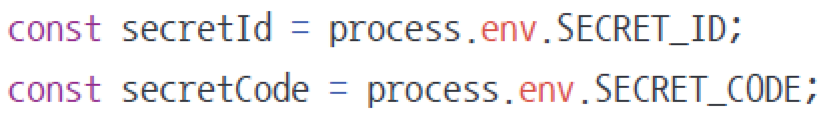


콘솔

```
$ node
> process.version
v14.0.0 // 설치된 노드의 버전입니다.
> process.arch
x64 // 프로세서 아키텍처 정보입니다. arm, ia32 등의 값일 수도 있습니다.
> process.platform
win32 // 운영체제 플랫폼 정보입니다. linux나 darwin, freebsd 등의 값일 수도 있습니다.
> process.pid
14736 // 현재 프로세스의 아이디입니다. 프로세스를 여러 개 가질 때 구분할 수 있습니다.
> process.uptime()
199.36 // 프로세스가 시작된 후 흐른 시간입니다. 단위는 초입니다.
> process.execPath
C:\\Program Files\\nodejs\\node.exe // 노드의 경로입니다.
> process.cwd()
C:\\Users\\zerocho // 현재 프로세스가 실행되는 위치입니다.
> process.cpuUsage()
{ user: 390000, system: 203000 } // 현재 cpu 사용량입니다.
```

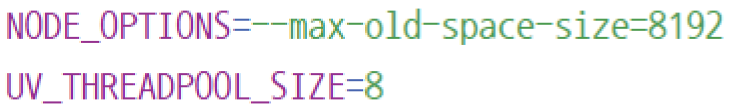
process.env

- It is an object containing system environment variables.
  - It is also used to store sensitive information such as secret keys (database passwords, third-party app keys, etc.).
  - Environment variables can be accessed using process.env.



```
const secretId = process.env.SECRET_ID;
const secretCode = process.env.SECRET_CODE;
```

  - Some environment variables have an impact on the behavior of the Node.js process during execution. For example:
    - NODE_OPTIONS: Node.js execution options.
    - UV_THREADPOOL_SIZE: Specifies the number of threads in the thread pool.
      - max-old-space-size is an option that specifies the amount of memory that Node.js can use.

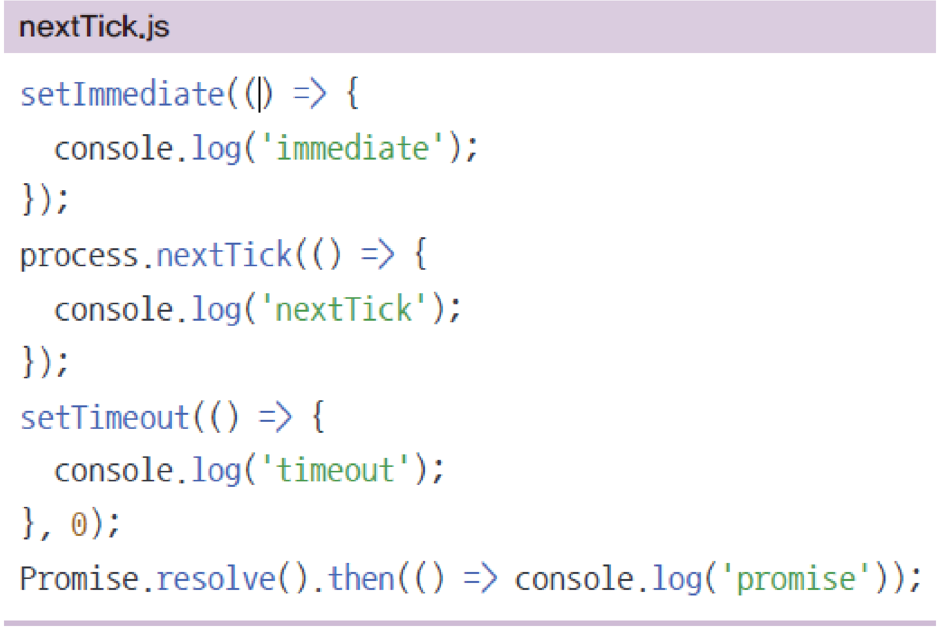


```
NODE_OPTIONS=--max-old-space-size=8192
UV_THREADPOOL_SIZE=8
```

process.nextTick

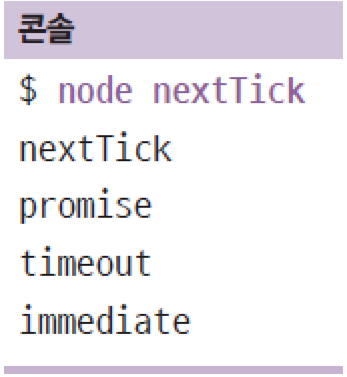
- The event loop prioritizes processing the nextTick callback functions over other callback functions.
  - However, excessive use of nextTick can delay the execution of other callback functions.
  - A similar scenario occurs with promises, which have a higher priority, similar to nextTick.

- In the example below, the promise and nextTick callbacks are executed before setImmediate and setTimeout:



nextTick.js

```
setImmediate(() => {
  console.log('immediate');
});
process.nextTick(() => {
  console.log('nextTick');
});
setTimeout(() => {
  console.log('timeout');
}, 0);
Promise.resolve().then(() => console.log('promise'));
```



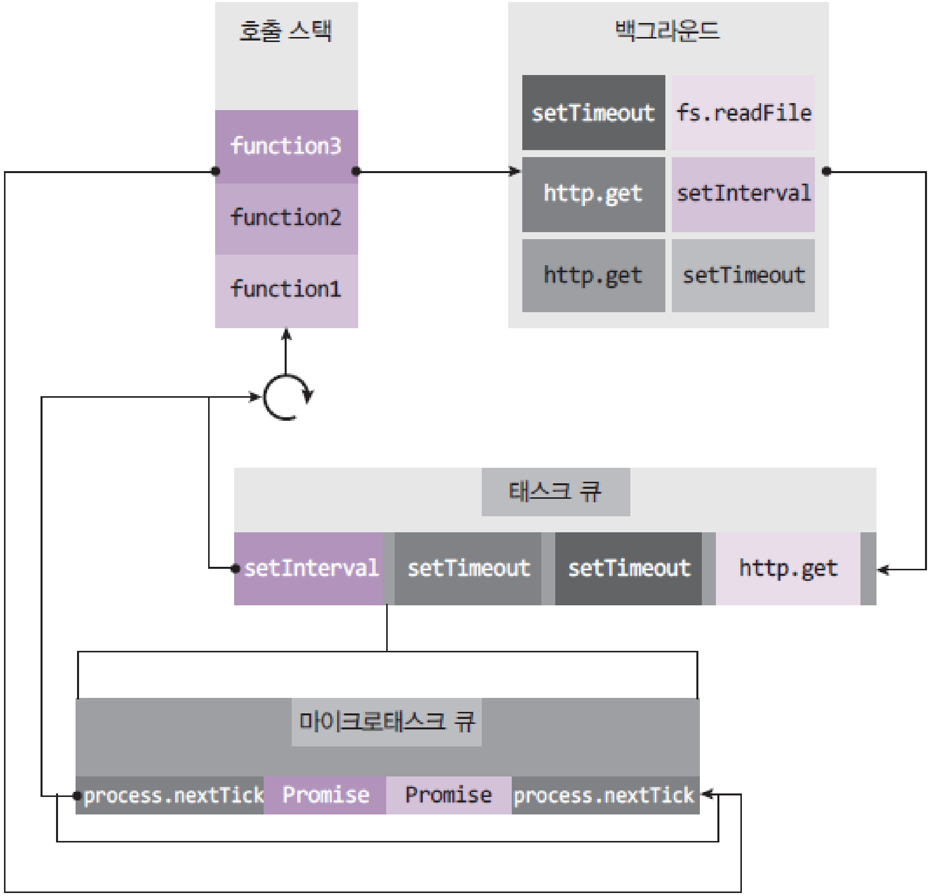
콘솔

```
$ node nextTick
nextTick
promise
timeout
immediate
```

microtask

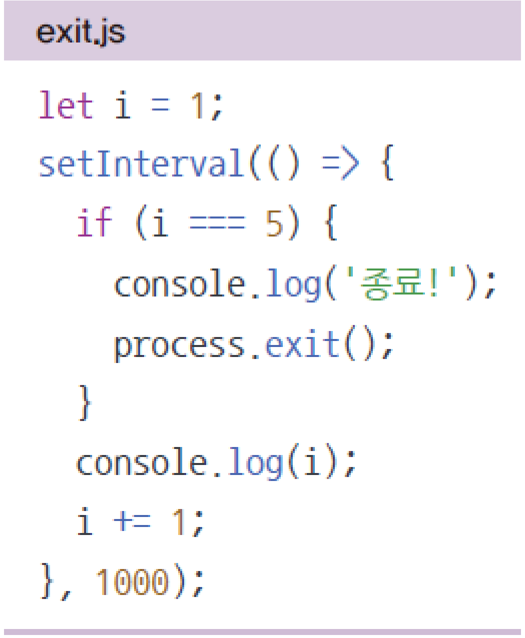
- a type of asynchronous task that is executed in the microtask queue.
  - The concept of microtasks is closely associated with the event loop and is used for handling tasks that need to be executed before the next cycle of the event loop.

▼ 그림 3-6 태스크와 마이크로태스크



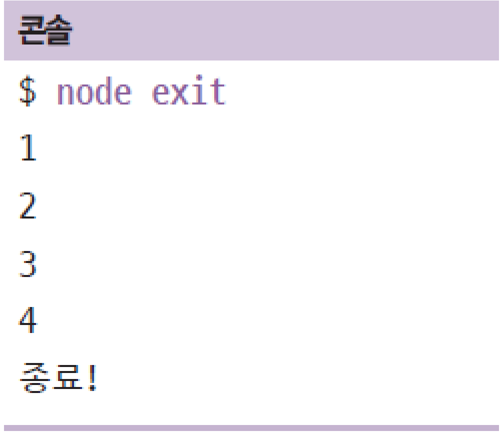
process.exit

- To halt the current process in Node.js, you use the process.exit() method.
  - If the exit code is absent or 0, it signifies a normal termination.
  - Any other exit code indicates an abnormal termination.



exit.js

```
let i = 1;
setInterval(() => {
  if (i === 5) {
    console.log('종료!');
    process.exit();
  }
  console.log(i);
  i += 1;
}, 1000);
```



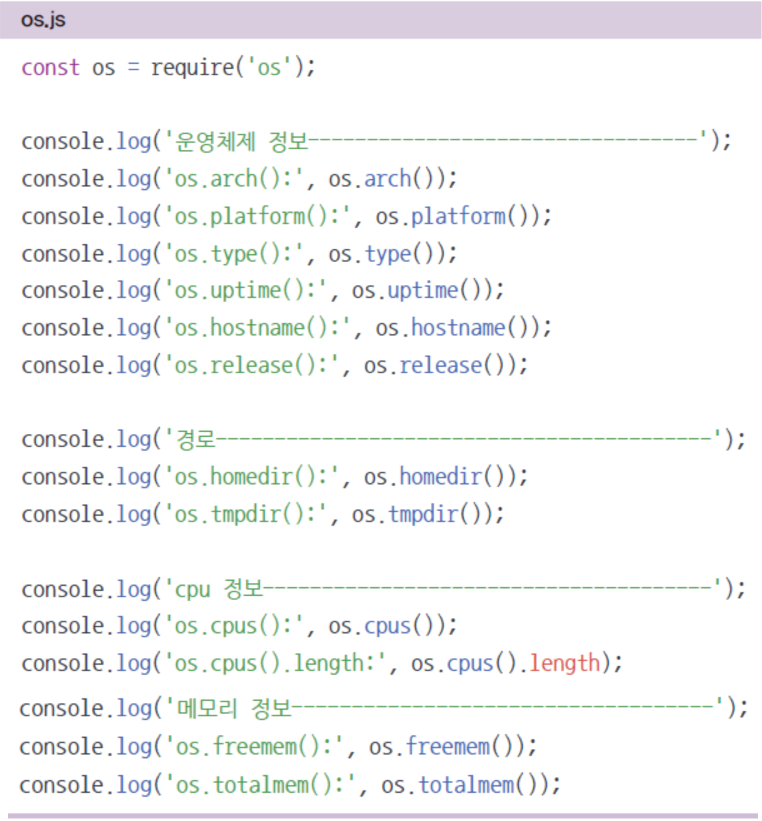
콘솔

```
$ node exit
1
2
3
4
종료!
```

Node.js core modules

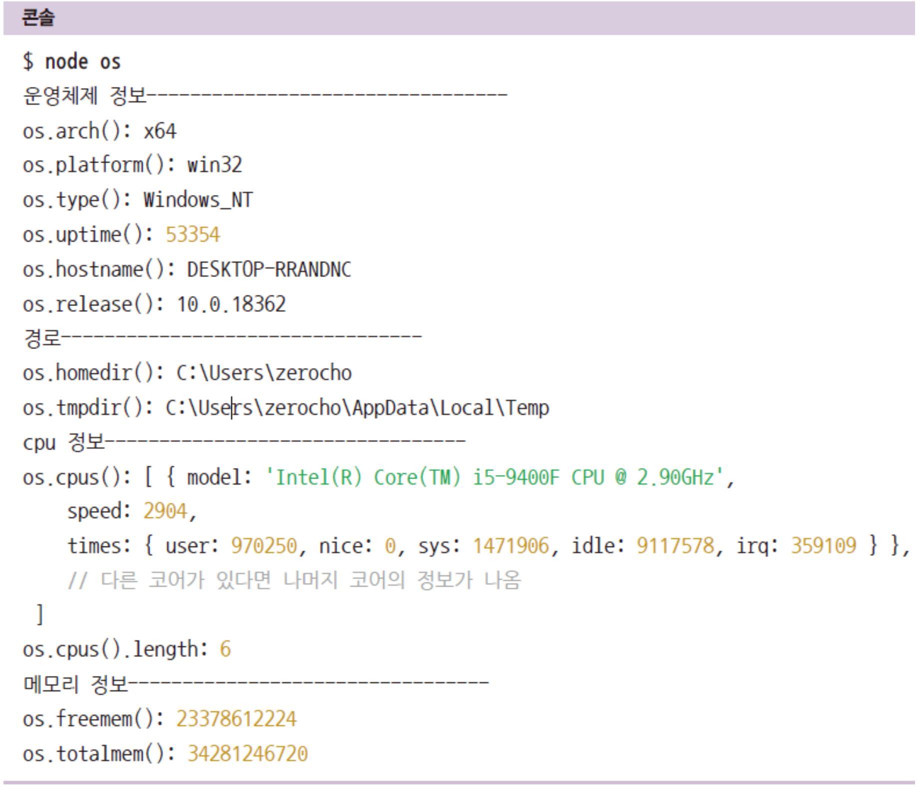
os module

- The information about the operating system is contained in the os module in Node.js.
  - You can obtain this module by using require:



os.js

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

console.log('운영체제 정보----------------------------------');
console.log('os.arch():', os.arch());
console.log('os.platform():', os.platform());
console.log('os.type():', os.type());
console.log('os.uptime():', os.uptime());
console.log('os.hostname():', os.hostname());
console.log('os.release():', os.release());

console.log('경로------------------------------------------');
console.log('os.homedir():', os.homedir());
console.log('os.tmpdir():', os.tmpdir());

console.log('cpu 정보--------------------------------------');
console.log('os.cpus():', os.cpus());
console.log('os.cpus().length:', os.cpus().length);
console.log('메모리 정보-----------------------------------');
console.log('os.freemem():', os.freemem());
console.log('os.totalmem():', os.totalmem());
```



콘솔

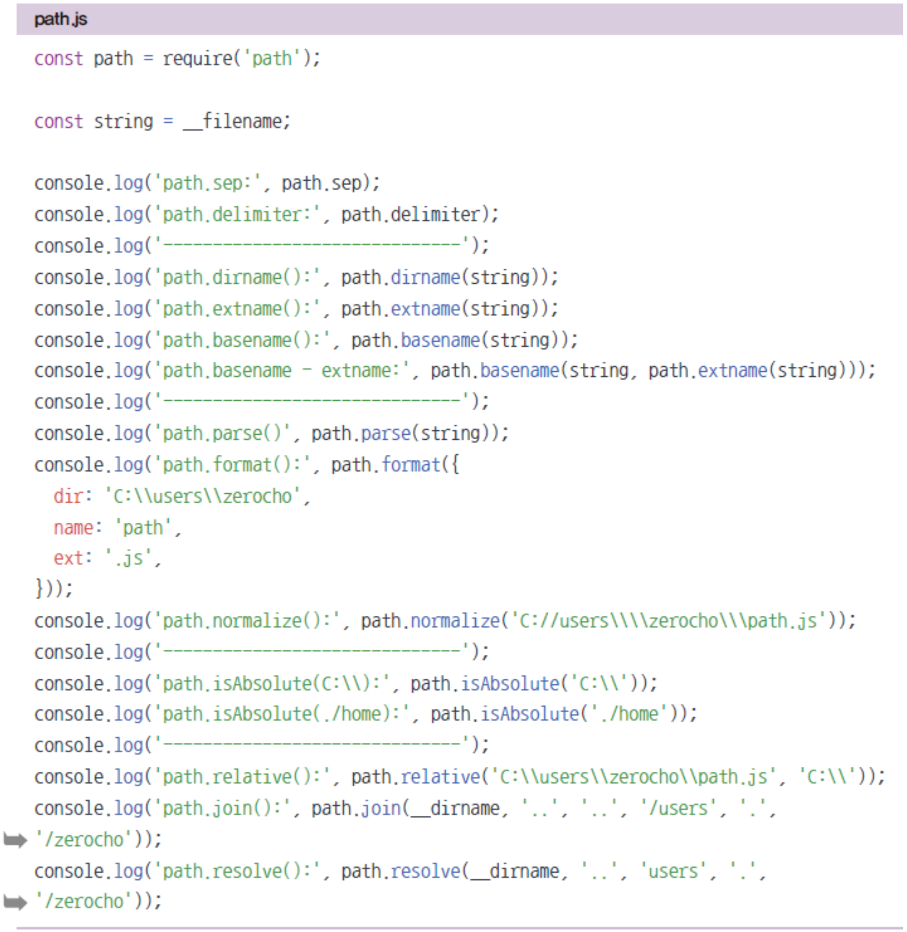
```
$ node os
운영체제 정보----------------------------------
os.arch(): x64
os.platform(): win32
os.type(): Windows_NT
os.uptime(): 53354
os.hostname(): DESKTOP-RRANDNC
os.release(): 10.0.18362
경로------------------------------------------
os.homedir(): C:\Users\zerocho
os.tmpdir(): C:\Users\zerocho\AppData\Local\Temp
cpu 정보--------------------------------------
os.cpus(): [ { model: 'Intel(R) Core(TM) i5-9400F CPU @ 2.90GHz',
    speed: 2904,
    times: { user: 970250, nice: 0, sys: 1471906, idle: 9117578, irq: 359109 } },
    // 다른 코어가 있다면 나머지 코어의 정보가 나옴
 ]
os.cpus().length: 6
메모리 정보-----------------------------------
os.freemem(): 23378612224
os.totalmem(): 34281246720
```

os module method

- os.arch(): Returns the CPU architecture. Equivalent to process.arch.
- os.platform(): Returns the operating system platform. Equivalent to process.platform.
- os.type(): Returns the operating system type.
- os.uptime(): Returns the elapsed time in seconds since the operating system was started.
- os.hostname(): Returns the host name of the operating system.
- os.release(): Returns the release version of the operating system.
- os.homedir(): Returns the home directory path of the current user.
- os.tmpdir(): Returns the temporary file directory path.
- os.cpus(): Returns an array of objects containing information about each CPU/core.
- os.freemem(): Returns the amount of free system memory (RAM).
- os.totalmem(): Returns the total amount of system memory.

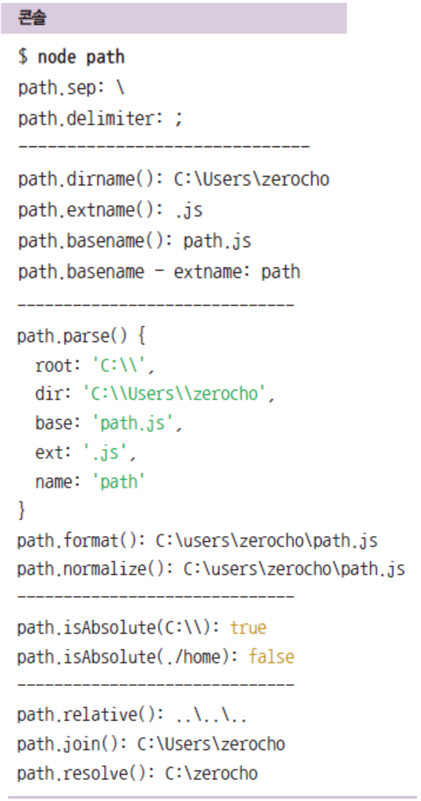
path module

- The path module provides methods for working with file and directory paths and helps in manipulating them.
  - It's particularly useful because it takes care of differences in path conventions between different operating systems (Windows: '₩', POSIX: '/')



path.js

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

const string = __filename;

console.log('path.sep:', path.sep);
console.log('path.delimiter:', path.delimiter);
console.log('------------------------------');
console.log('path.dirname():', path.dirname(string));
console.log('path.extname():', path.extname(string));
console.log('path.basename():', path.basename(string));
console.log('path.basename - extname:', path.basename(string, path.extname(string)));
console.log('------------------------------');
console.log('path.parse()', path.parse(string));
console.log('path.format():', path.format({
  dir: 'C:\\users\\zerocho',
  name: 'path',
  ext: '.js',
}));
console.log('path.normalize():', path.normalize('C://users\\\\zerocho\\\path.js'));
console.log('------------------------------');
console.log('path.isAbsolute(C:\\):', path.isAbsolute('C:\\'));
console.log('path.isAbsolute(./home):', path.isAbsolute('./home'));
console.log('------------------------------');
console.log('path.relative():', path.relative('C:\\users\\zerocho\\path.js', 'C:\\'));
console.log('path.join():', path.join(__dirname, '..', '..', '/users', '.',
➡ '/zerocho'));
console.log('path.resolve():', path.resolve(__dirname, '..', 'users', '.',
➡ '/zerocho'));
```



콘솔

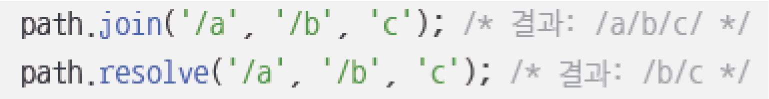
```
$ node path
path.sep: \
path.delimiter: ;
------------------------------
path.dirname(): C:\Users\zerocho
path.extname(): .js
path.basename(): path.js
path.basename - extname: path
------------------------------
path.parse() {
  root: 'C:\\',
  dir: 'C:\\Users\\zerocho',
  base: 'path.js',
  ext: '.js',
  name: 'path'
}
path.format(): C:\users\zerocho\path.js
path.normalize(): C:\users\zerocho\path.js
------------------------------
path.isAbsolute(C:\\): true
path.isAbsolute(./home): false
------------------------------
path.relative(): ..\..\..
path.join(): C:\Users\zerocho
path.resolve(): C:\zerocho
```

path module method

- path.sep: Path segment separator. For Windows, it is '', and for POSIX systems, it is '/'.
- path.delimiter: Path segment delimiter in environment variables. For Windows, it is ';', and for POSIX systems, it is ':'.
- path.dirname(path): Returns the directory name of a path.
- path.extname(path): Returns the extension of a file path.
- path.basename(path, ext): Returns the file name (with extension). If ext is provided, it removes that extension.
- path.parse(path): Parses a file path into an object with root, dir, base, ext, and name properties.
- path.format(pathObject): Joins path segments in a path object into a path string.
- path.normalize(path): Normalizes a path by resolving '..' and '.' segments.
- path.isAbsolute(path): Determines whether a path is absolute or relative.
- path.relative(from, to): Returns the relative path from one path to another.
- path.join(...paths): Joins path segments into a single path, handling '..' and '.'.
- path.resolve(...paths): Similar to path.join() but with some differences, discussed in the next slide.

path module

- differences between path.join and path.resolve
  - path.join(...paths): Joins path segments into a single path. Treats arguments as a sequence of path segments to concatenate. It's used for handling relative paths.
  - path.resolve(...paths): Resolves an absolute path. Treats arguments as a sequence of path segments and resolves them from right to left, with each subsequent segment being resolved against (appended to) the previous.



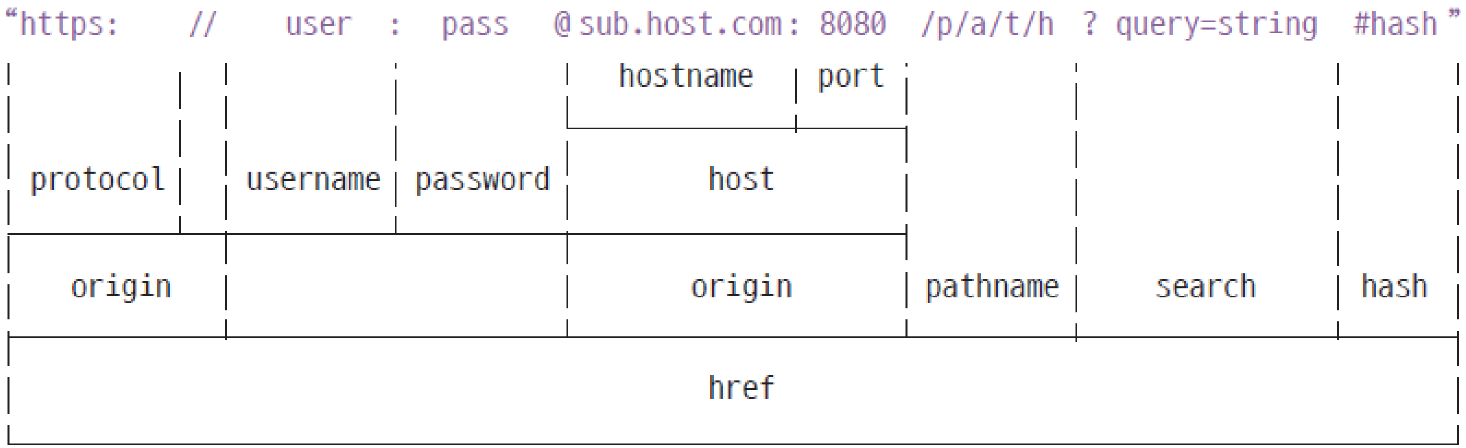
```
path.join('/a', '/b', 'c'); /* 결과: /a/b/c/ */
path.resolve('/a', '/b', 'c'); /* 결과: /b/c */
```

- For the difference between '₩₩' and '₩':
  - '₩₩' is the escape sequence for a single backslash in JavaScript strings.
  - '₩', as is, is used as a path separator in Windows.

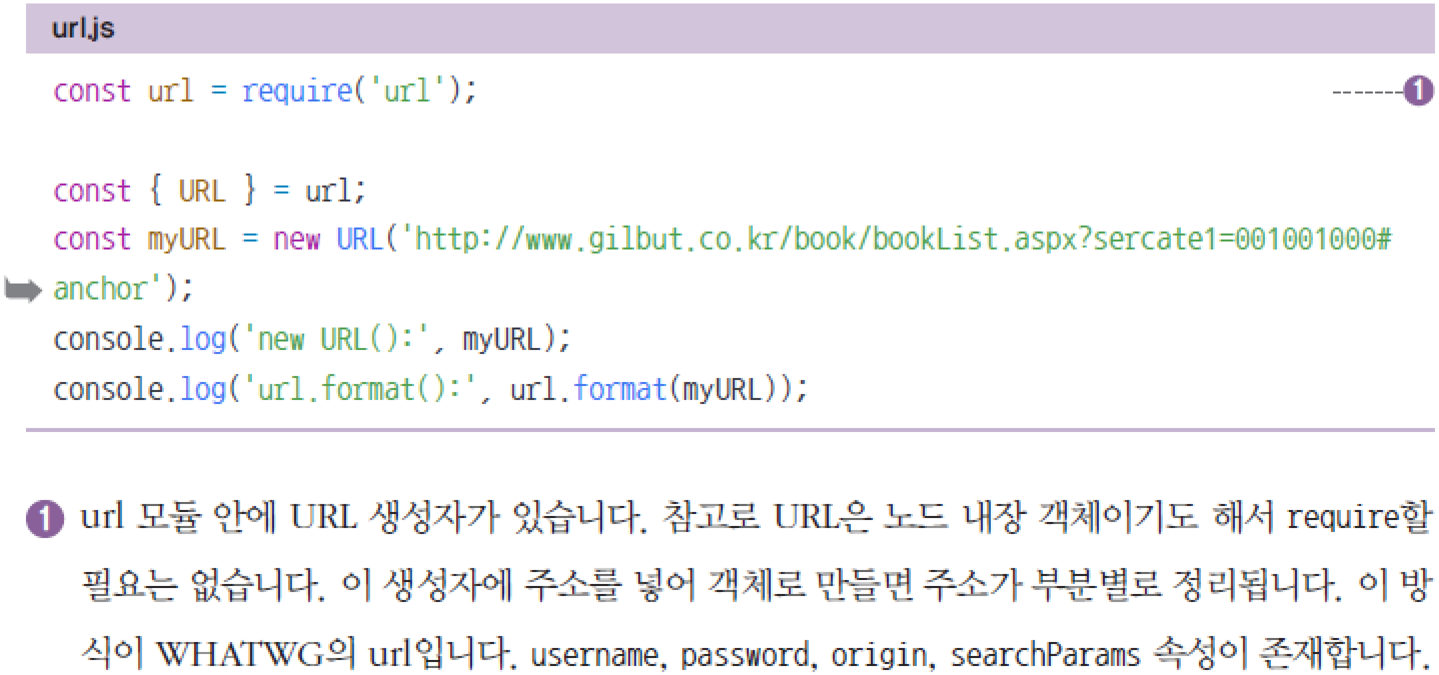
- Regarding the use of POSIX or Windows paths:
  - To work with POSIX-style paths on Windows, you can use path.posix methods.
  - To work with Windows-style paths on POSIX systems, you can use path.win32 methods.

url module

- Module that facilitates easy manipulation of internet addresses (URLs).
  - There are two main approaches to handling URLs: the traditional Node.js approach and the WHATWG (Web Hypertext Application Technology Working Group) approach.
  - Presently, the industry predominantly uses the WHATWG approach.
  - The WHATWG URL structure is as follows:


"https: // user : pass @ sub.host.com : 8080 /p/a/t/h ? query=string #hash"
hostname
port
protocol
username
password
host
origin
origin
pathname
search
hash
href

url module example

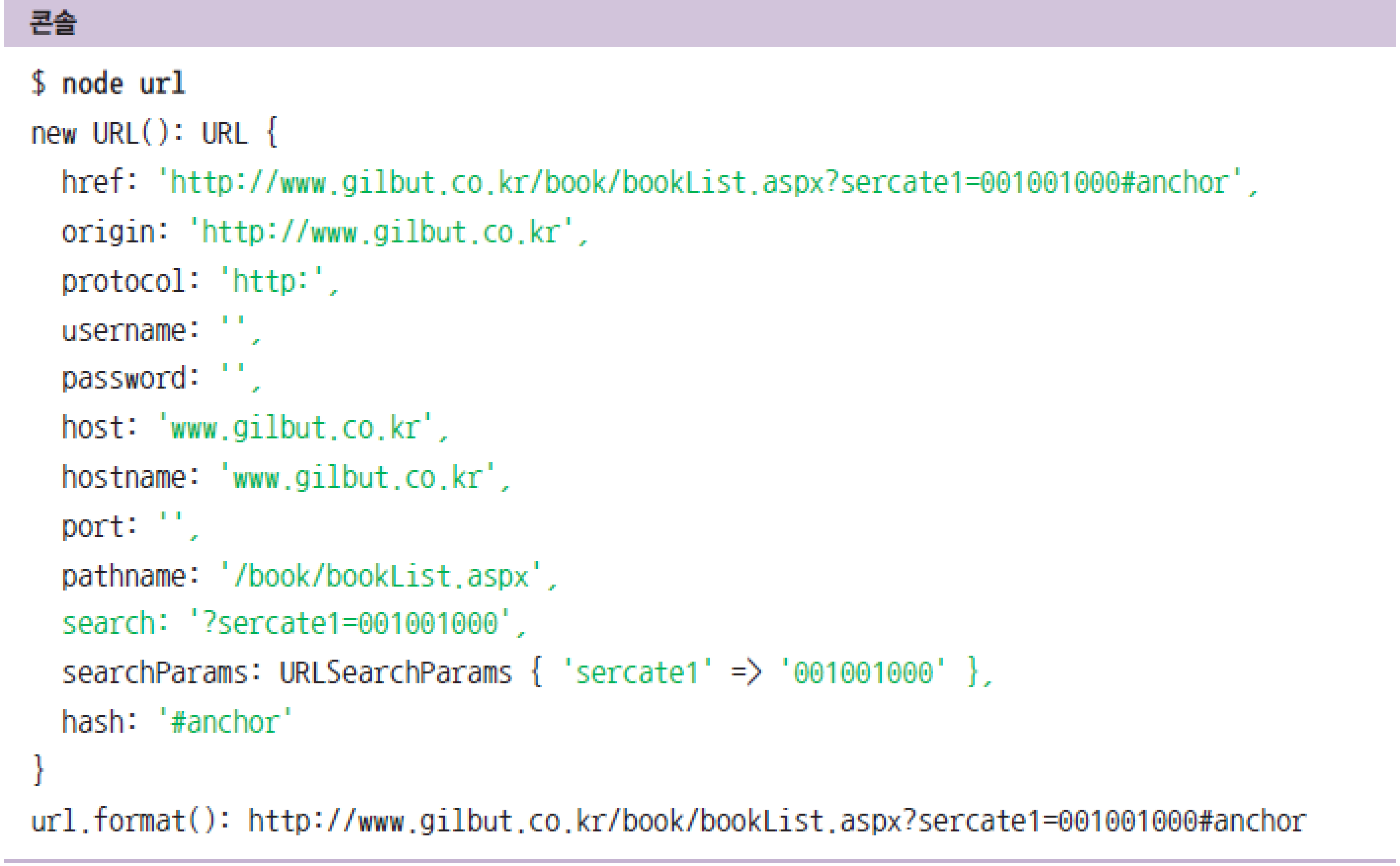


url.js

```
const url = require('url');                                          ❶

const { URL } = url;
const myURL = new URL('http://www.gilbut.co.kr/book/bookList.aspx?sercate1=001001000#anchor');
console.log('new URL():', myURL);
console.log('url.format():', url.format(myURL));
```

❶ url 모듈 안에 URL 생성자가 있습니다. 참고로 URL은 노드 내장 객체이기도 해서 require할 필요는 없습니다. 이 생성자에 주소를 넣어 객체로 만들면 주소가 부분별로 정리됩니다. 이 방식이 WHATWG의 url입니다. username, password, origin, searchParams 속성이 존재합니다.

url module example

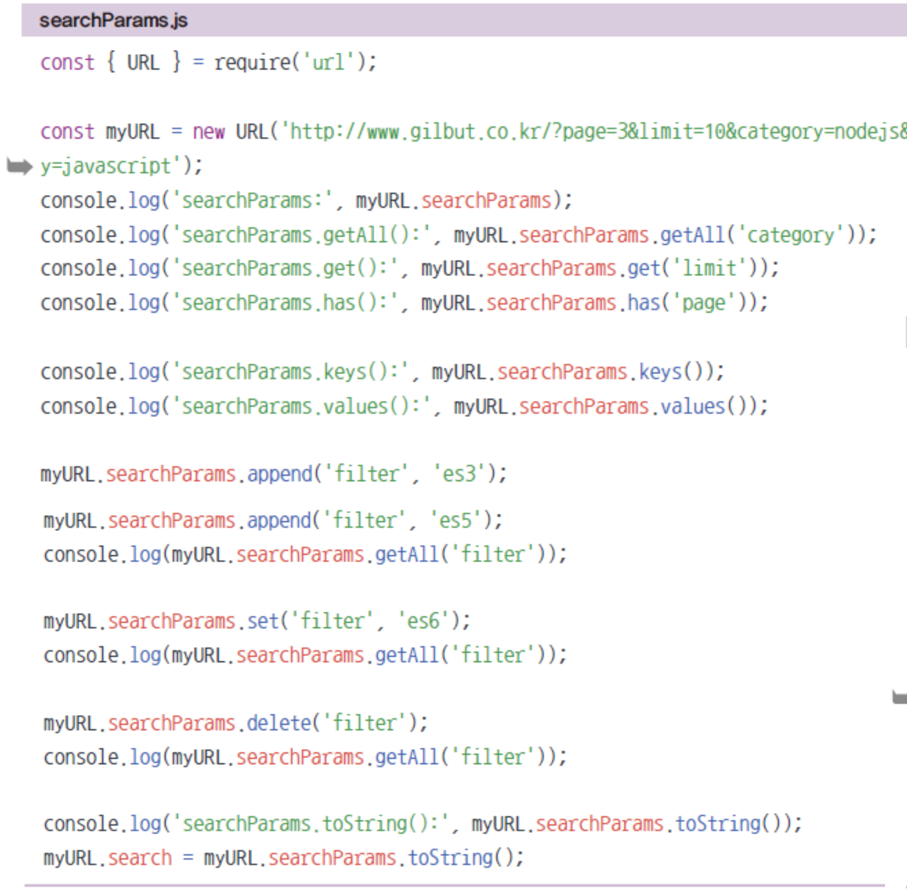


콘솔

```
$ node url
new URL(): URL {
  href: 'http://www.gilbut.co.kr/book/bookList.aspx?sercate1=001001000#anchor',
  origin: 'http://www.gilbut.co.kr',
  protocol: 'http:',
  username: '',
  password: '',
  host: 'www.gilbut.co.kr',
  hostname: 'www.gilbut.co.kr',
  port: '',
  pathname: '/book/bookList.aspx',
  search: '?sercate1=001001000',
  searchParams: URLSearchParams { 'sercate1' => '001001000' },
  hash: '#anchor'
}
url.format(): http://www.gilbut.co.kr/book/bookList.aspx?sercate1=001001000#anchor
```

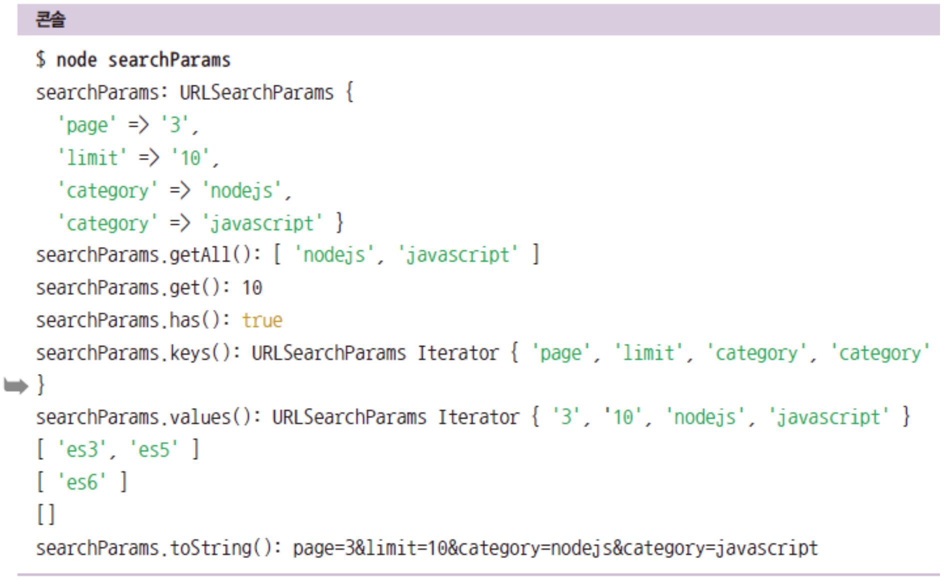
searchParams

- Object that assists in handling the query string (search) part in the WHATWG approach.
  - For example, in the URL ?page=3&limit=10&category=nodejs&category=javascript, this object helps with the manipulation of the query string portion.



searchParams.js

```
const { URL } = require('url');

const myURL = new URL('http://www.gilbut.co.kr/?page=3&limit=10&category=nodejs&category=javascript');
console.log('searchParams:', myURL.searchParams);
console.log('searchParams.getAll():', myURL.searchParams.getAll('category'));
console.log('searchParams.get():', myURL.searchParams.get('limit'));
console.log('searchParams.has():', myURL.searchParams.has('page'));

console.log('searchParams.keys():', myURL.searchParams.keys());
console.log('searchParams.values():', myURL.searchParams.values());

myURL.searchParams.append('filter', 'es3');
myURL.searchParams.append('filter', 'es5');
console.log(myURL.searchParams.getAll('filter'));

myURL.searchParams.set('filter', 'es6');
console.log(myURL.searchParams.getAll('filter'));

myURL.searchParams.delete('filter');
console.log(myURL.searchParams.getAll('filter'));

console.log('searchParams.toString():', myURL.searchParams.toString());
myURL.search = myURL.searchParams.toString();
```



콘솔

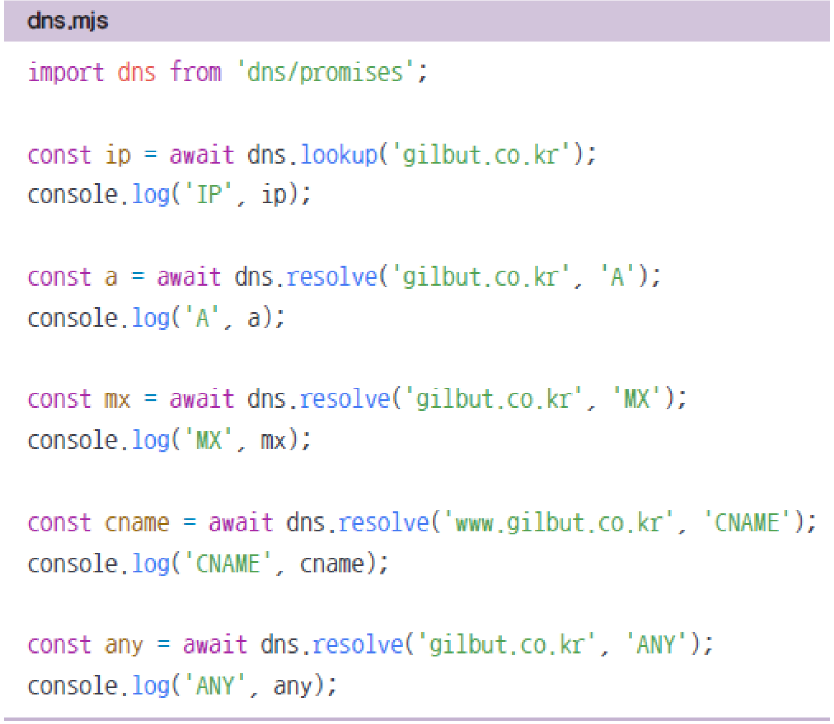
```
$ node searchParams
searchParams: URLSearchParams {
  'page' => '3',
  'limit' => '10',
  'category' => 'nodejs',
  'category' => 'javascript' }
searchParams.getAll(): [ 'nodejs', 'javascript' ]
searchParams.get(): 10
searchParams.has(): true
searchParams.keys(): URLSearchParams Iterator { 'page', 'limit', 'category', 'category' }
searchParams.values(): URLSearchParams Iterator { '3', '10', 'nodejs', 'javascript' }
[ 'es3', 'es5' ]
[ 'es6' ]
[]
searchParams.toString(): page=3&limit=10&category=nodejs&category=javascript
```

searchParams example

- getAll(key): Retrieves all values associated with the key. In the case of the "category" key, it contains two values, namely "nodejs" and "javascript."
- get(key): Retrieves the first value associated with the key.
- has(key): Checks whether the key exists.
- keys(): Retrieves all keys of the searchParams as an iterator object.
- values(): Retrieves all values of the searchParams as an iterator object.
- append(key, value): Adds the specified key. If there are existing values for the key, it keeps them and adds one more.
- set(key, value): Similar to append, but removes all existing values for the key and adds the new one.
- delete(key): Removes the specified key.
- toString(): Converts the manipulated searchParams object back to a string. Assigning this string to the search property will reflect the changes in the address object.

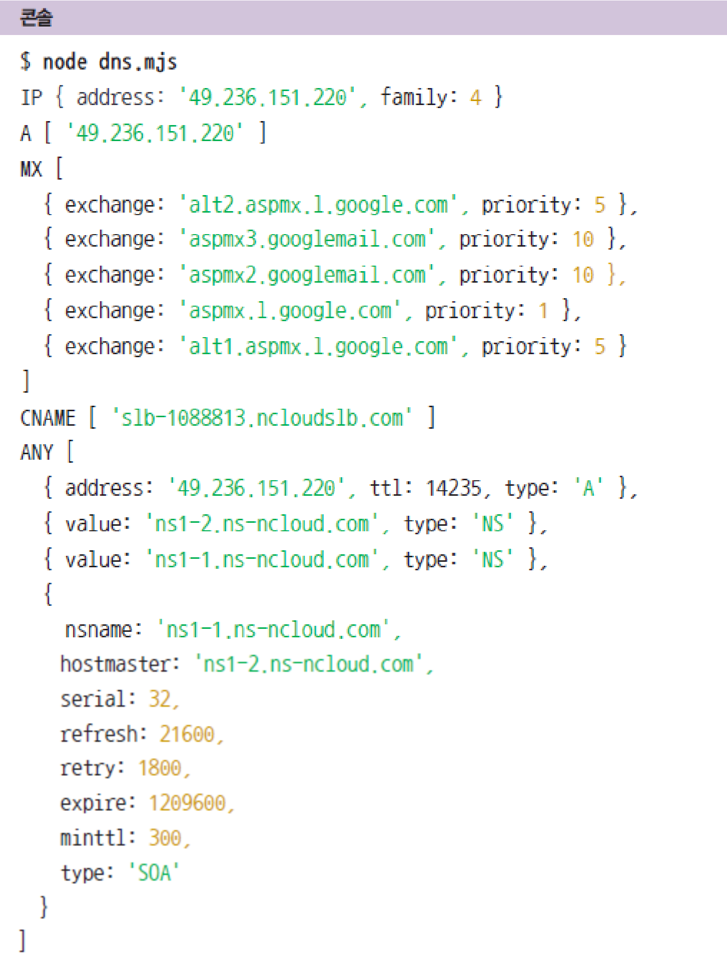
dns

- A module used for handling DNS (Domain Name System) operations.
  - It is utilized when obtaining IP addresses or DNS records through domain names.



dns.mjs

```
import dns from 'dns/promises';

const ip = await dns.lookup('gilbut.co.kr');
console.log('IP', ip);

const a = await dns.resolve('gilbut.co.kr', 'A');
console.log('A', a);

const mx = await dns.resolve('gilbut.co.kr', 'MX');
console.log('MX', mx);

const cname = await dns.resolve('www.gilbut.co.kr', 'CNAME');
console.log('CNAME', cname);

const any = await dns.resolve('gilbut.co.kr', 'ANY');
console.log('ANY', any);
```



콘솔

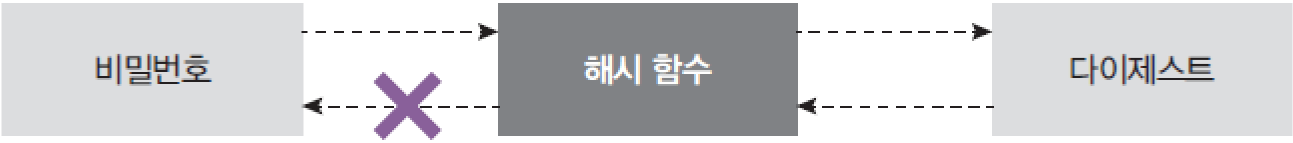
```
$ node dns.mjs
IP { address: '49.236.151.220', family: 4 }
A [ '49.236.151.220' ]
MX [
  { exchange: 'alt2.aspmx.l.google.com', priority: 5 },
  { exchange: 'aspmx3.googlemail.com', priority: 10 },
  { exchange: 'aspmx2.googlemail.com', priority: 10 },
  { exchange: 'aspmx.l.google.com', priority: 1 },
  { exchange: 'alt1.aspmx.l.google.com', priority: 5 }
]
CNAME [ 'slb-1088813.ncloudslb.com' ]
ANY [
  { address: '49.236.151.220', ttl: 14235, type: 'A' },
  { value: 'ns1-2.ns-ncloud.com', type: 'NS' },
  { value: 'ns1-1.ns-ncloud.com', type: 'NS' },
  {
    nsname: 'ns1-1.ns-ncloud.com',
    hostmaster: 'ns1-2.ns-ncloud.com',
    serial: 32,
    refresh: 21600,
    retry: 1800,
    expire: 1209600,
    minttl: 300,
    type: 'SOA'
  }
]
```

one-way encryption (crypto module)

- Encryption is possible, but decryption is not.
  - Encryption: Converting plaintext into ciphertext.
  - Decryption: Decoding ciphertext into plaintext.

- The flagship approach for one-way encryption is the hash technique.
  - It involves transforming a string into a fixed-length, different string.
  - For example, the string "abcdefgh" may be transformed into "qvew".

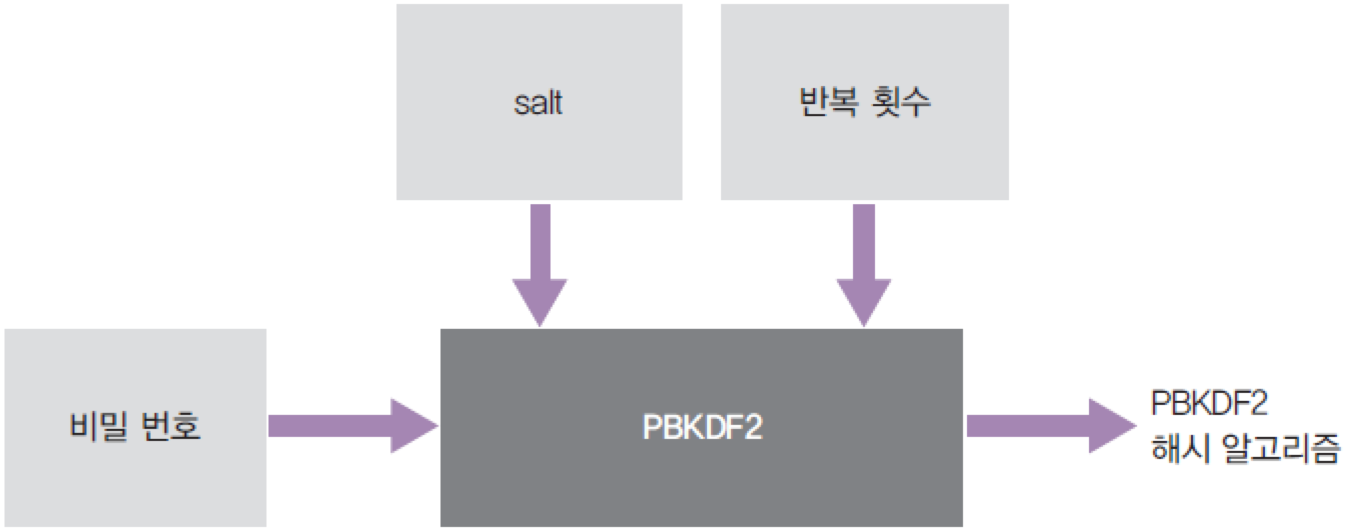
그림 3-8 해시 함수



pbkdf2

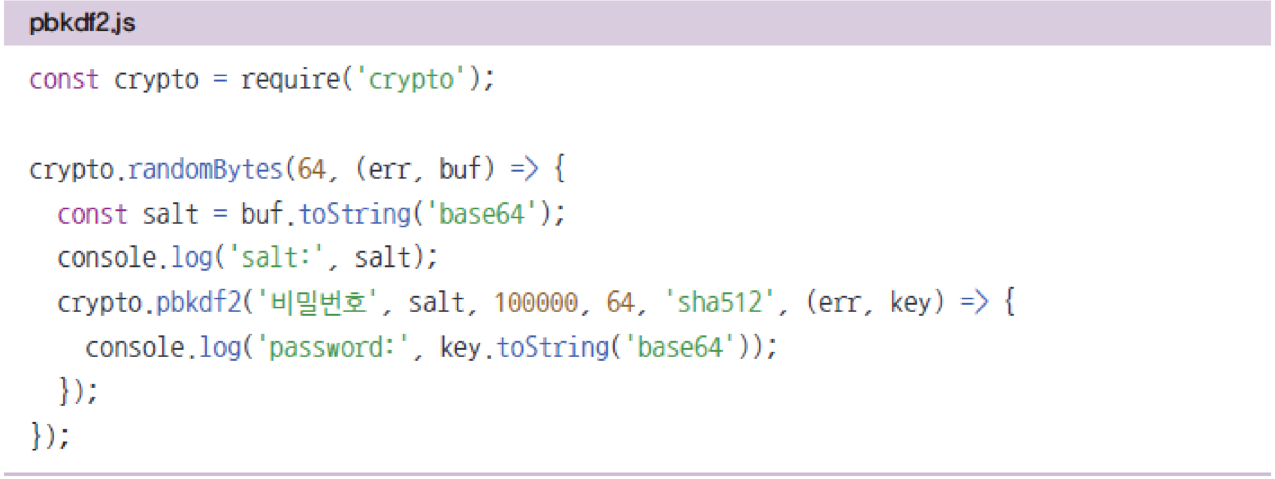
- With the advancement of computers, traditional encryption algorithms are under threat.
  - If SHA-512 becomes vulnerable, transitioning to SHA-3 may be necessary.
  - Currently, password encryption is often done using algorithms like PBKDF2, bcrypt, or scrypt.
  - Node.js supports PBKDF2 and scrypt algorithms.

그림 3-9 pbkdf2



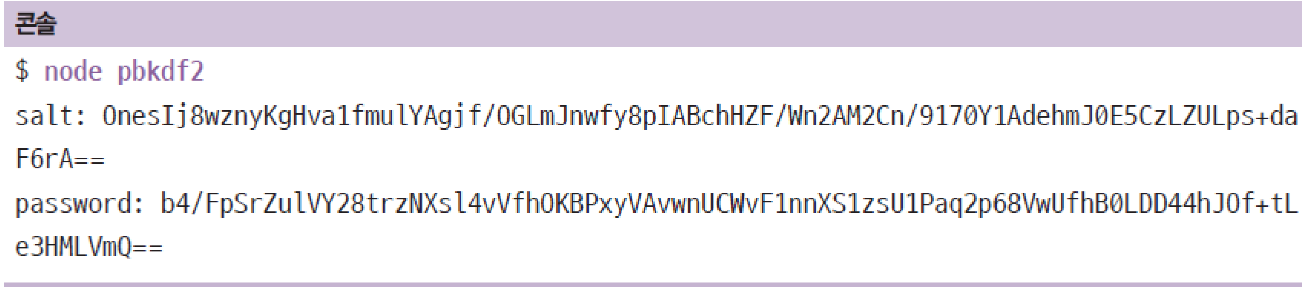
pbkdf2 example

- Generating a 64-byte string using crypto.randomBytes serves as a salt.
- For the pbkdf2 function, the arguments include the password, salt, iteration count, output byte length, and algorithm, in that order.
- It is recommended to adjust the iteration count to make the encryption take around 1 second.



pbkdf2.js

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

crypto.randomBytes(64, (err, buf) => {
  const salt = buf.toString('base64');
  console.log('salt:', salt);
  crypto.pbkdf2('비밀번호', salt, 100000, 64, 'sha512', (err, key) => {
    console.log('password:', key.toString('base64'));
  });
});
```

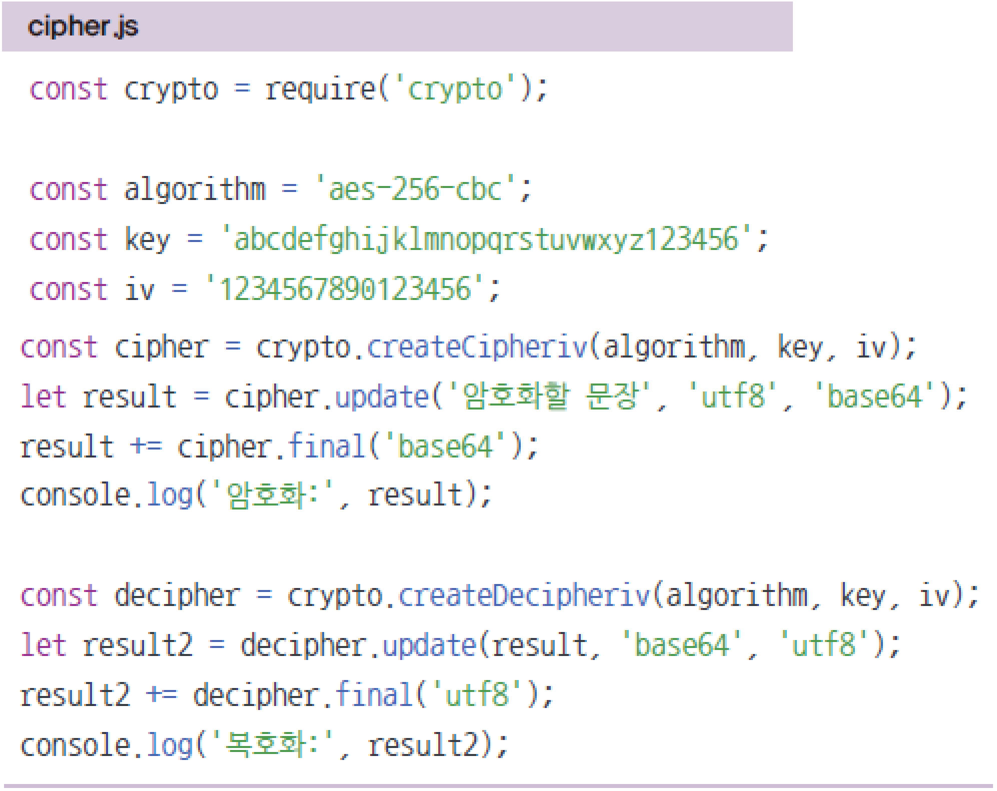


콘솔

```
$ node pbkdf2
salt: OnesIj8wznyKgHva1fmulYAgjf/OGLmJnwfy8pIABchHZF/Wn2AM2Cn/9170Y1AdehmJ0E5CzLZULps+da
F6rA==
password: b4/FpSrZulVY28trzNXsl4vVfhOKBPxyVAvwnUCWvF1nnXS1zsU1Paq2p68VwUfhB0LDD44hJOf+tL
e3HMLVmQ==
```

two-way encryption

- Symmetric encryption involves using a key for both encryption and decryption.
    - The same key is used for both processes, ensuring that the encryption and decryption are symmetrical.

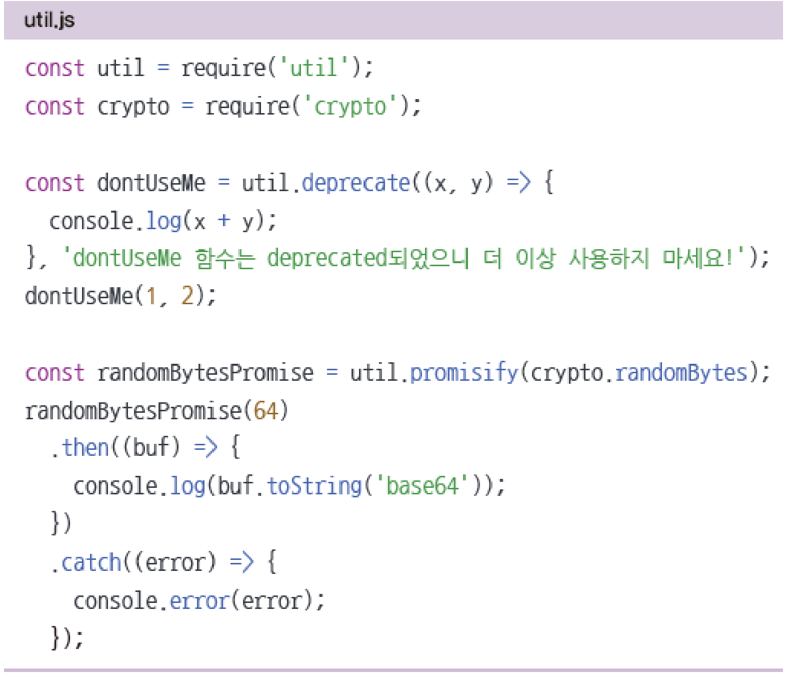


cipher.js

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

const algorithm = 'aes-256-cbc';
const key = 'abcdefghijklmnopqrstuvwxyz123456';
const iv = '1234567890123456';
const cipher = crypto.createCipheriv(algorithm, key, iv);
let result = cipher.update('암호화할 문장', 'utf8', 'base64');
result += cipher.final('base64');
console.log('암호화:', result);

const decipher = crypto.createDecipheriv(algorithm, key, iv);
let result2 = decipher.update(result, 'base64', 'utf8');
result2 += decipher.final('utf8');
console.log('복호화:', result2);
```

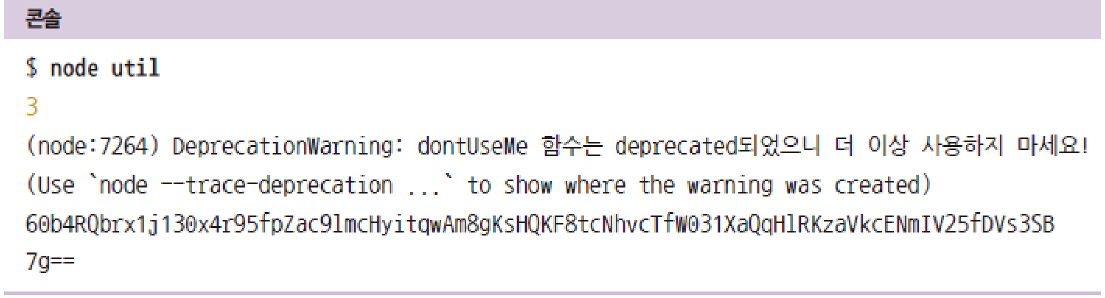
util

- util.deprecate: Indicates that the function has been deprecated.
- util.promisify: Converts callback-based pattern to promise-based pattern.



util.js

```
const util = require('util');
const crypto = require('crypto');

const dontUseMe = util.deprecate((x, y) => {
  console.log(x + y);
}, 'dontUseMe 함수는 deprecated되었으니 더 이상 사용하지 마세요!');
dontUseMe(1, 2);

const randomBytesPromise = util.promisify(crypto.randomBytes);
randomBytesPromise(64)
  .then((buf) => {
    console.log(buf.toString('base64'));
  })
  .catch((error) => {
    console.error(error);
  });
```

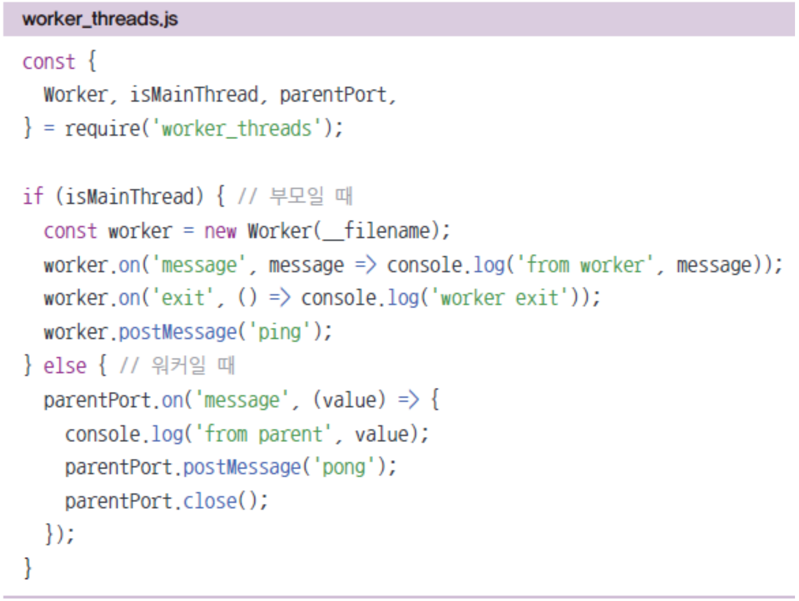


콘솔

```
$ node util
3
(node:7264) DeprecationWarning: dontUseMe 함수는 deprecated되었으니 더 이상 사용하지 마세요!
(Use `node --trace-deprecation ...` to show where the warning was created)
60b4RQbrx1j130x4r95fpZac9lmcHyitqwAm8gKsHQKF8tcNhvcTfW031XaQqHlRKzaVkcENmIV25fDVs3SB
7g==
```

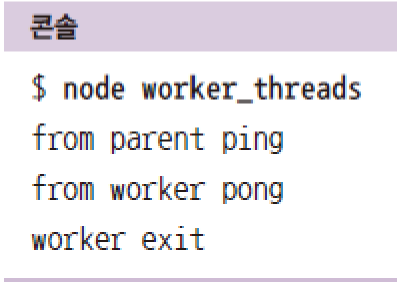
worker_threads

- In Node.js, you can work with a multi-threaded approach.
  - isMainThread: It distinguishes whether the current code is running in the main thread or a worker thread.
  - In the main thread, you can execute the current file (__filename) in a worker thread using new Worker. You can send data from the parent to the worker using worker.postMessage.
  - To receive data from the parent, use parentPort.on('message'), and to send data, use postMessage.



worker_threads.js

```
const {
  Worker, isMainThread, parentPort,
} = require('worker_threads');

if (isMainThread) { // 부모일 때
  const worker = new Worker(__filename);
  worker.on('message', message => console.log('from worker', message));
  worker.on('exit', () => console.log('worker exit'));
  worker.postMessage('ping');
} else { // 워커일 때
  parentPort.on('message', (value) => {
    console.log('from parent', value);
    parentPort.postMessage('pong');
    parentPort.close();
  });
}
```

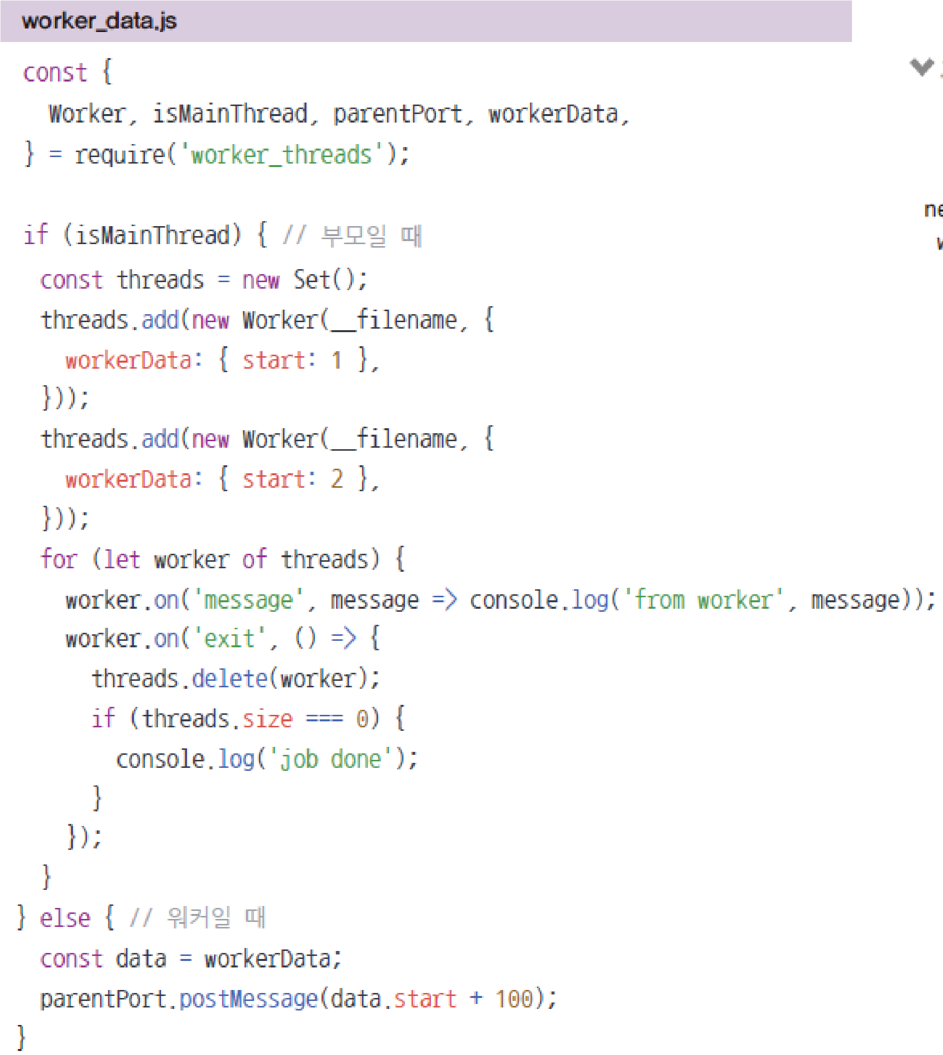


콘솔

```
$ node worker_threads
from parent ping
from worker pong
worker exit
```

using multiple workers

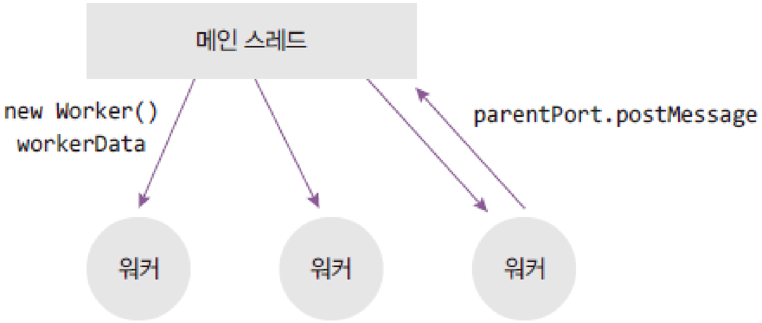
- "As many times as new Worker is invoked, a corresponding worker thread is created."

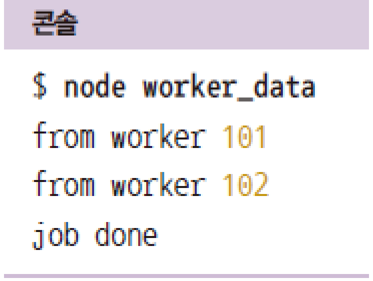


worker_data.js

```
const {
  Worker, isMainThread, parentPort, workerData,
} = require('worker_threads');

if (isMainThread) { // 부모일 때
  const threads = new Set();
  threads.add(new Worker(__filename, {
    workerData: { start: 1 },
  }));
  threads.add(new Worker(__filename, {
    workerData: { start: 2 },
  }));
  for (let worker of threads) {
    worker.on('message', message => console.log('from worker', message));
    worker.on('exit', () => {
      threads.delete(worker);
      if (threads.size === 0) {
        console.log('job done');
      }
    });
  }
} else { // 워커일 때
  const data = workerData;
  parentPort.postMessage(data.start + 100);
}
```

그림 3-10 메인 스레드와 워커의 통신



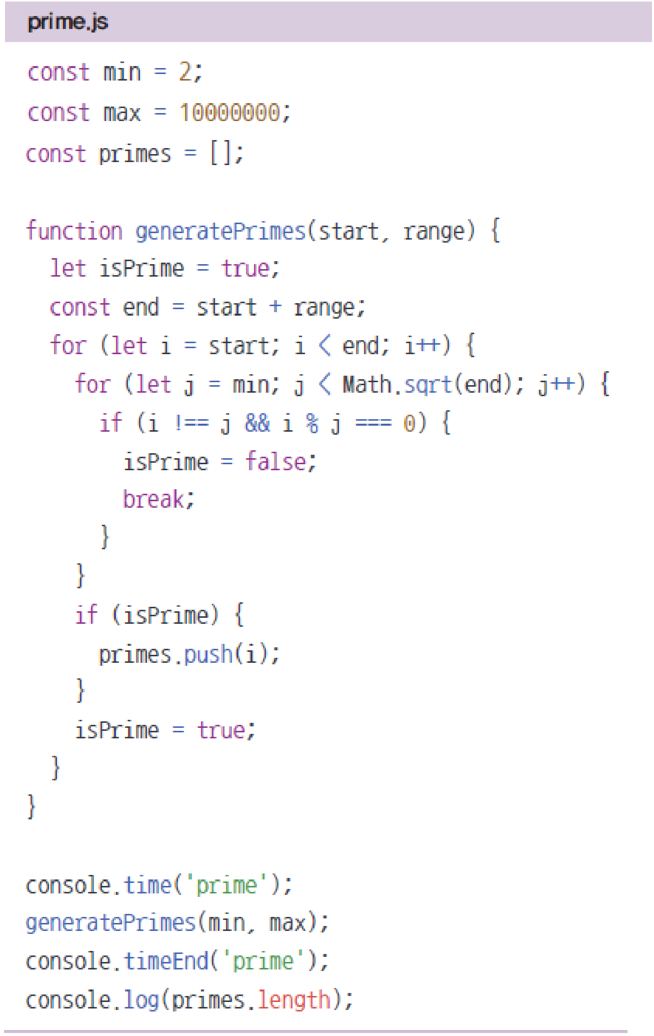


콘솔

```
$ node worker_data
from worker 101
from worker 102
job done
```

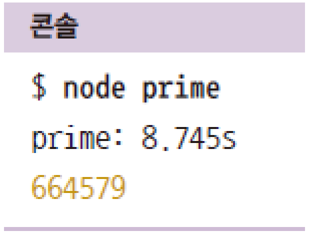
worker_threads example

- When not using worker threads (in a single-threaded environment)



prime.js

```
const min = 2;
const max = 10000000;
const primes = [];

function generatePrimes(start, range) {
  let isPrime = true;
  const end = start + range;
  for (let i = start; i < end; i++) {
    for (let j = min; j < Math.sqrt(end); j++) {
      if (i !== j && i % j === 0) {
        isPrime = false;
        break;
      }
    }
    if (isPrime) {
      primes.push(i);
    }
    isPrime = true;
  }
}

console.time('prime');
generatePrimes(min, max);
console.timeEnd('prime');
console.log(primes.length);
```

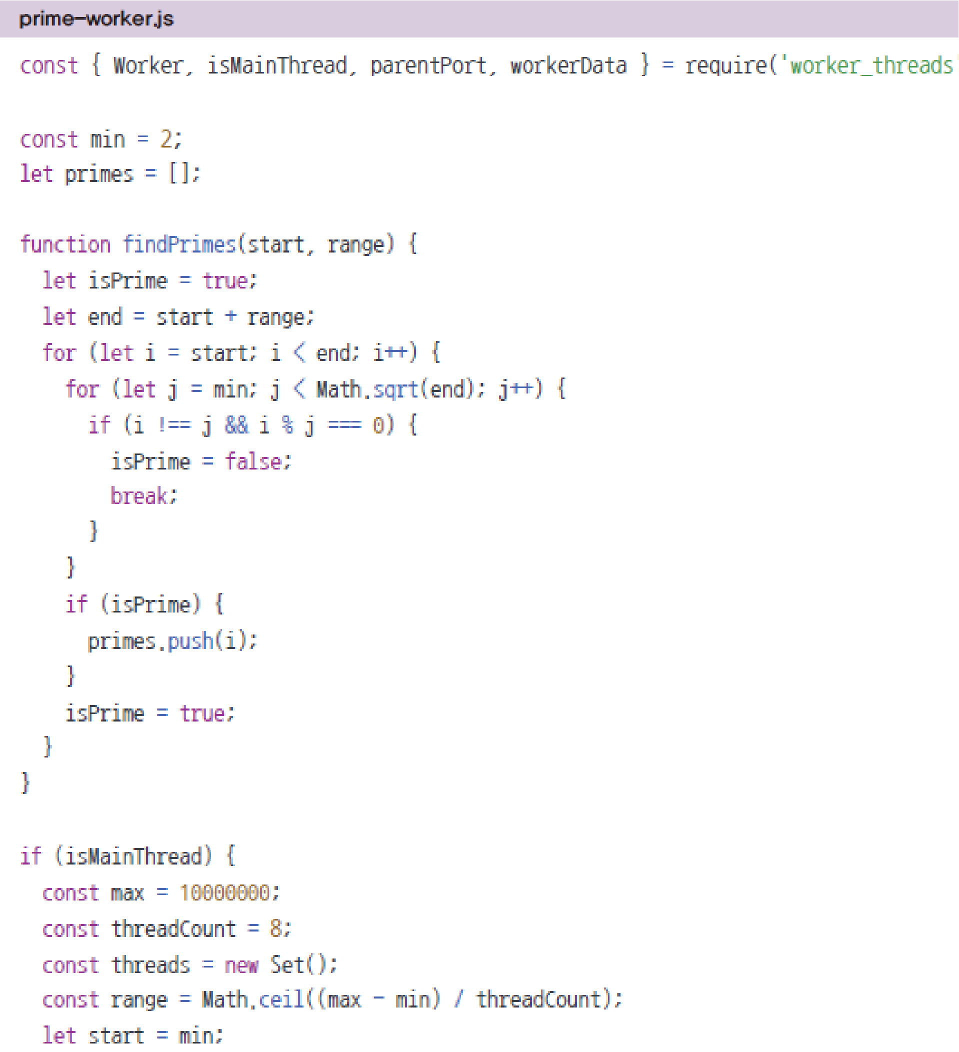


콘솔

```
$ node prime
prime: 8.745s
664579
```

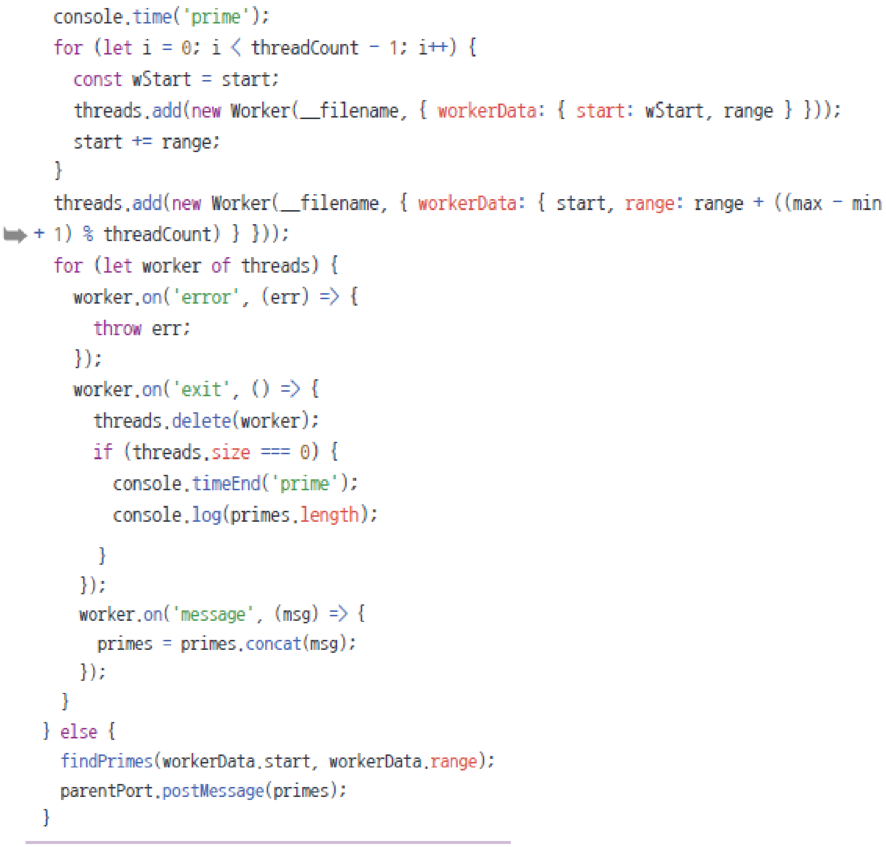
worker_threads example

- When using worker threads

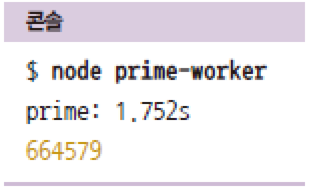


prime-worker.js

```
const { Worker, isMainThread, parentPort, workerData } = require('worker_threads

const min = 2;
let primes = [];

function findPrimes(start, range) {
  let isPrime = true;
  let end = start + range;
  for (let i = start; i < end; i++) {
    for (let j = min; j < Math.sqrt(end); j++) {
      if (i !== j && i % j === 0) {
        isPrime = false;
        break;
      }
    }
    if (isPrime) {
      primes.push(i);
    }
    isPrime = true;
  }
}

if (isMainThread) {
  const max = 10000000;
  const threadCount = 8;
  const threads = new Set();
  const range = Math.ceil((max - min) / threadCount);
  let start = min;
```



```
  console.time('prime');
  for (let i = 0; i < threadCount - 1; i++) {
    const wStart = start;
    threads.add(new Worker(__filename, { workerData: { start: wStart, range } }));
    start += range;
  }
  threads.add(new Worker(__filename, { workerData: { start, range: range + ((max - min
+ 1) % threadCount) } }));
  for (let worker of threads) {
    worker.on('error', (err) => {
      throw err;
    });
    worker.on('exit', () => {
      threads.delete(worker);
      if (threads.size === 0) {
        console.timeEnd('prime');
        console.log(primes.length);
      }
    });
    worker.on('message', (msg) => {
      primes = primes.concat(msg);
    });
  }
} else {
  findPrimes(workerData.start, workerData.range);
  parentPort.postMessage(primes);
}
```

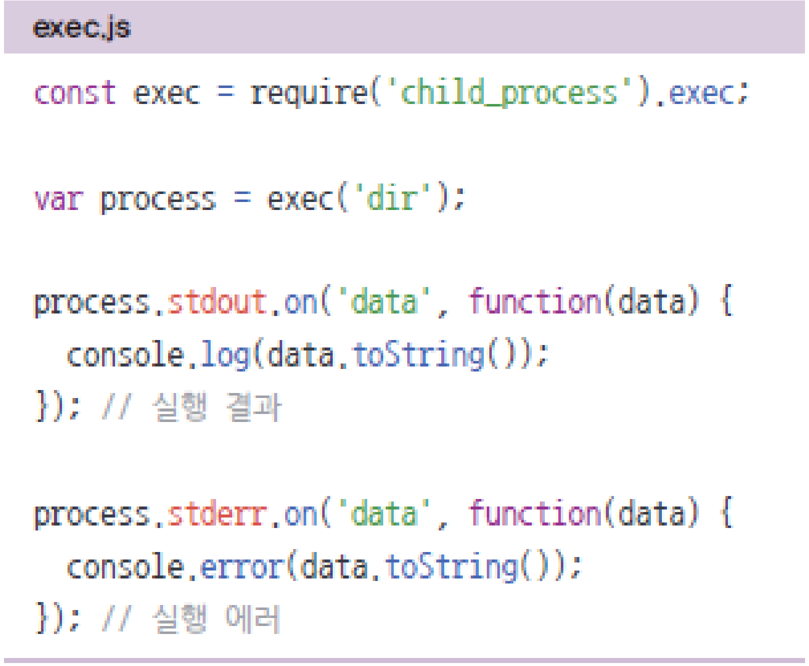


콘솔

```
$ node prime-worker
prime: 1.752s
664579
```

child_process

- In Node.js, you can use the child_process module to execute other programs or perform command-line operations.
    - It spawns a new process outside the current Node.js process to execute the specified command.
- Here's an example of running the command dir (or ls on Linux) using Node.js:

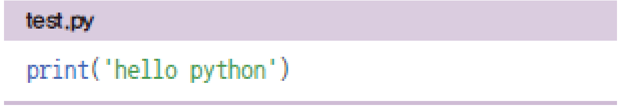


exec.js

```
const exec = require('child_process').exec;

var process = exec('dir');

process.stdout.on('data', function(data) {
  console.log(data.toString());
}); // 실행 결과

process.stderr.on('data', function(data) {
  console.error(data.toString());
}); // 실행 에러
```

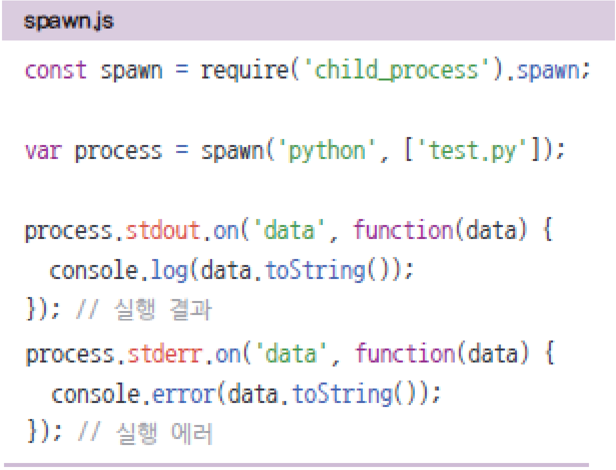
child_process example

- In Node.js, you can use the child_process module to execute Python scripts. Here's an example:
  - This example assumes that Python 3 is installed on your system.



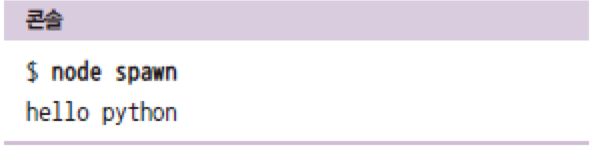
test.py

```python
print('hello python')
```



spawn.js

```javascript
const spawn = require('child_process').spawn;

var process = spawn('python', ['test.py']);

process.stdout.on('data', function(data) {
  console.log(data.toString());
}); // 실행 결과
process.stderr.on('data', function(data) {
  console.error(data.toString());
}); // 실행 에러
```



콘솔

```
$ node spawn
hello python
```

other modules

- **async_hooks:** An experimental module that allows tracking the flow of asynchronous code.
- **dgram:** Used for operations related to UDP (User Datagram Protocol).
- **net:** Utilized when working with lower-level TCP (Transmission Control Protocol) or IPC (Inter-Process Communication) communication, more raw than HTTP.
- **perf_hooks:** Used for more precise performance measurements than console.time when measuring performance.
- **querystring:** A module used for handling query strings before the introduction of URLSearchParams.
- **string_decoder:** Used for converting buffer data to strings.
- **tls:** Employed for tasks related to TLS (Transport Layer Security) and SSL (Secure Sockets Layer).
- **tty:** Used for operations related to the terminal.
- **v8:** Used when direct access to the V8 engine is required.
- **vm:** Utilized for direct access to the virtual machine.
- **wasi:** An experimental module used when executing WebAssembly.