

The Node.js logo, featuring a stylized 'N' composed of yellow, red, and blue squares, with a black crosshair.

Node.js

본 강의교안을 무단 배포 및 게재를 할 경우 법적인 처벌을 받을 수 있음을 알려드립니다.



JavaScript

- Overview
 - What is JavaScript
 - Client-side JavaScript
 - Advantages of JavaScript
 - Limitations of JavaScript
 - Where is JavaScript Today?
- Hello World!
 - Internal Scripts
 - External Scripts
- Code structure
 - statements
 - expressions
 - semicolons
 - comments
- Strict mode
 - The “use strict” Directive
 - Declaring Strict Mode
 - Syntax errors



Overview

■ What is Node.js?

- 오픈소스, 구글에서 개발, C++로 작성됨
- Node.js는 확장성 있는 네트워크 애플리케이션(특히 서버 사이드) 개발에 사용되는 소프트웨어 플랫폼
- 작성언어는 Javascript
- non-blocking I/O와 single thread 이벤트 루프를 통한 높은 처리 성능
- 내장 http 서버 라이브러리를 포함하고 있어 웹 서버에서 아파치 등의 별도의 소프트웨어 없이 동작하는 것이 가능
- V8(자바스크립트 엔진)으로 빌드 된 이벤트 기반 자바스크립트 런타임



Overview

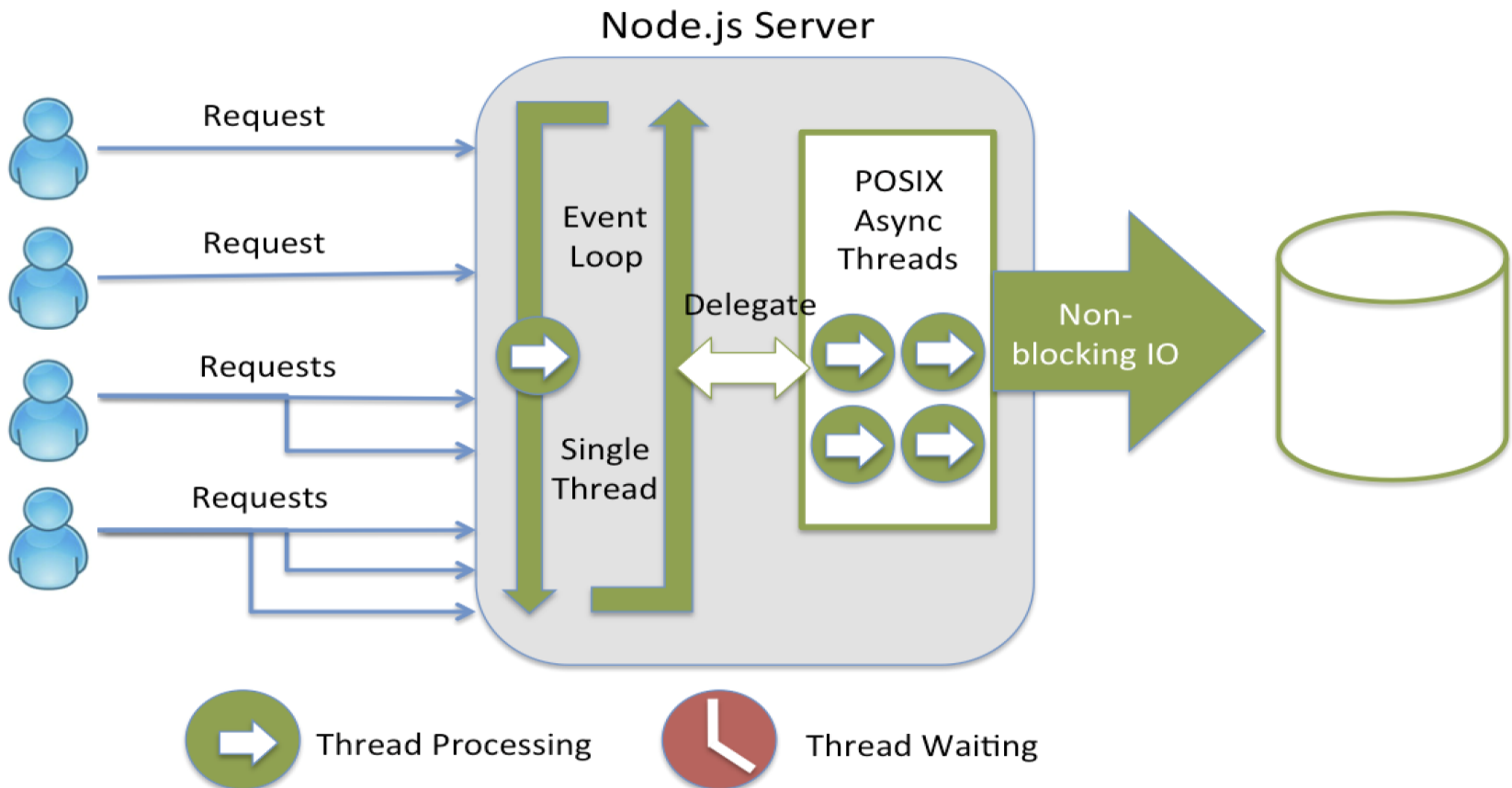
■ Node.js 역사

릴리스	코드명	출시일	LTS 상태	활동적인 LTS 시작일	유지보수 시작일	유지보수 종료일
v0.10		2013-03-11	수명 종료	-	2015-10-01	2016-10-31
v0.12		2015-02-06	수명 종료	-	2016-04-01	2016-12-31
v4	Argon ^[9]	2015-09-08	수명 종료	2015-10-01	2017-04-01	2018-04-01
v5		2015-10-29	LTS 없음	N/A		2016-06-30
v6	Boron ^[9]	2016-04-26	활동 중	2016-10-18	2018-04-18	2019-04-18
v7		2016-10-25	LTS 없음	N/A		2017-06-30
v8	Carbon ^[9]	2017-05-30	활동 중	2017-10-31	2019-04-01	2019-12 예정
v9		2017-10-01	LTS 없음	N/A		2018-06-30
v10	Dubnium ^[9]	2018-04-24	활동 중	2018-10-30	2020-04 예정	2021-04 예정

Node.js와 multi-thread server 비교

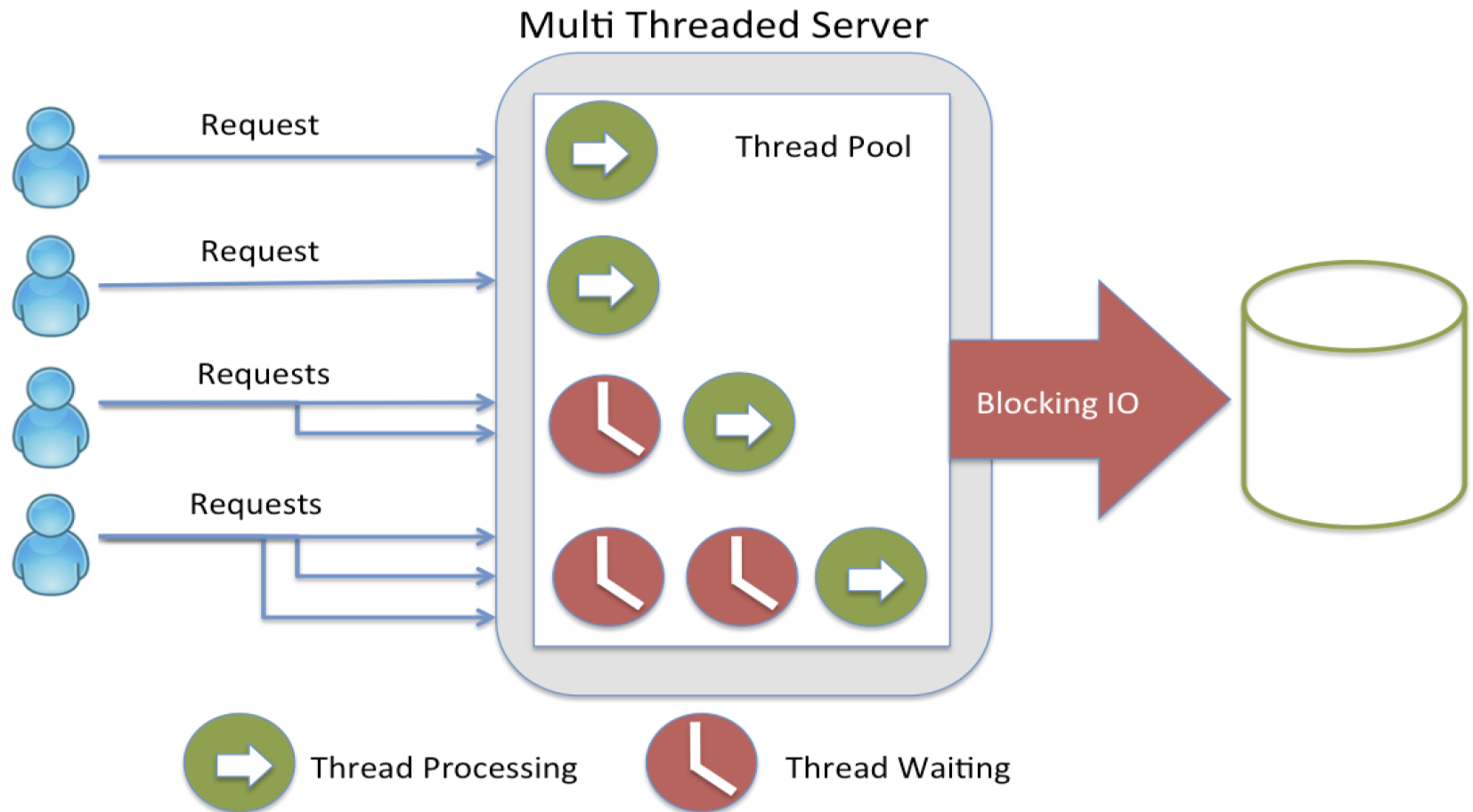
5

- Node.js Server



Node.js의 장점

■ Multi Threaded Server



How to download & Install Node.js

7

- node.js download
 - <http://nodejs.org/ko/download>




다운로드


최신 LTS 버전: 10.16.0 (includes npm 6.9.0)


플랫폼에 맞게 미리 빌드된 Node.js 인스톨러나 소스코드를 다운받아서 바로 개발을 시작하세요.

LTS
대다수 사용자에게 추천

현재 버전
최신 기능


Windows Installer
node-v10.16.0-x64.msi


macOS Installer
node-v10.16.0.pkg


Source Code
node-v10.16.0.tar.gz

Windows Installer (.msi)
Windows Binary (.zip)
macOS Installer (.pkg)
macOS Binary (.tar.gz)
Linux Binaries (x64)
Linux Binaries (ARM)

32-bit		64-bit
32-bit		64-bit
		64-bit
		64-bit
		64-bit
ARMv6	ARMv7	ARMv8

How to download & Install Node.js

8

- vscode download & Installation
 - <https://code.visualstudio.com/download>

Version 1.36 is now available! Read about the new features and fixes from June.

Download Visual Studio Code

Free and built on open source. Integrated Git, debugging and extensions.



↓ Windows

Windows 7, 8, 10

User Installer	64 bit	32 bit
System Installer	64 bit	32 bit
.zip	64 bit	32 bit



↓ .deb

Debian, Ubuntu

↓ .rpm

Red Hat, Fedora, SUSE

.deb	64 bit
.rpm	64 bit
.tar.gz	64 bit

[Snap Store](#)



↓ Mac

macOS 10.10 +

By downloading and using Visual Studio Code, you agree to the [license terms](#) and [privacy statement](#).



Hello World

```
var http = require('http');

http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.end('Hello World!');
}).listen(8080);
```

- ❖ `require`는 자바스크립트 모듈을 읽어 들인다
- ❖ `http` 모듈을 읽어 들이고 서버 어플리케이션을 생성한다
- ❖ 8080 포트로 클라이언트의 요청을 기다린다

```
$ node hello.js
```



Modules

- modules in Node.js

- 모듈(module)은 역사적으로 프로그래밍이라는 관점에서는 기본적으로 본체에 대한 독립된 하위 단위라는 필연적인 개념의 큰 틀을 따르고 있지만 본체와 모듈 간에 가지고 있었던 문제들을 해결해 나가는 과정에서 발전하였다(위키백과)
- 하나의 프로그램을 여러 개의 조각으로 쪼개 놓은 것
- Node.js에서 모듈이란 코드를 캡슐화 하는 방법
- Node.js에서는 **readymade** 모듈을 제공



Modules

- popular modules which are used in a Node js application
 - Express framework
 - Socket.io
 - Jade
 - MongoDB
 - Restify
 - BlueBird



Modules

- 모듈 종류
 - Node.js 기본 내장 모듈
 - 확장 모듈
 - 사용자 정의 모듈



Modules

- Node.js 기본 내장 모듈
 - Node.js에서 기본적으로 제공하는 모듈
 - 별도 설치 불필요
 - 사용하고자 하는 모듈을 CommonJS 형식의 **require** 함수를 이용하여 로드

	의미	
프로세스환경	os, process, cluster	
파일과 경로, URL	fs, path, URL, querystring, stream	
네트워크 모듈	http, https, net, dgram, dns	
전역객체(별도의 모듈 로딩 없이 사용 가능)	process, console, Buffer, require, __filename, __dirname, module.exports, Timeout	



Modules

- Node.js 확장 내장 모듈
 - npm, yarn을 이용하여 모듈 설치
 - 사용하고자 하는 모듈을 CommonJS 형식의 `require` 함수를 이용하여 로드
 - 확장 모듈 설치 및 사용방법

```
$ mkdir first-app
```

```
$ cd first-app
```

```
$ npm init -y
```

```
$ npm install express --save
```



Modules

- root folder에 app.js 파일 생성

```
const path = require('path');
const express = require('express');
const app = express();
const index = require('./routes/index')

app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'pug');
app.use('/', index);
var server = app.listen(3000, function(){
  console.log("Express server has started on port 3000")
});
```



Modules

- routes 폴더를 생성하고 하위에 index.js 파일 생성

```
const express = require('express');
const router = express.Router();

router.get('/main', (req, res) => {
  res.render('main', {title:'main'});
});

router.get('/about', (req, res) => {
  res.render('about', {title:'about'});
});

module.exports = router;
```

- ❖ express.Router 클래스를 사용하면 모듈식 마운팅 가능한 핸들러를 작성할 수 있습니다
- ❖ Router 인스턴스는 완전한 미들웨어이자 라우팅 시스템



Modules

- views folder를 생성하고, layout.pug, main.pug, about.pug 파일 생성

```
doctype html
html
  head
    title=`${title}`
  body
    h1 My Amazing App
  block content
```

layout.pug

```
extends layout
block content
  p= `hello ${title}`
```

main.pug



Modules

- 해당 프로젝트 폴더로 이동

```
$ node app.js
```

Modules

- Node.js 사용자 정의 모듈
 - 사용자가 별도의 js 파일을 생성
 - CommonJS의 `require`를 이용하여 모듈 로딩
 - es6의 `import` 구문을 사용할 경우 오류가 발생 할 수 있음
 - `module.exports` and `exports` 둘다 사용가능

```
function add(x, y){  
    return x+y;  
}  
function multiply(x, y){  
    return x*y;  
}  
module.exports = {add, multiply};
```

calculator.js

- ❖ `add`, `multiply` 함수를 `module.exports`에 객체 형태로 할당
- ❖ 객체의 `property:value` 형태를 단순화 시킨 es6 문법(이름이 동일할 경우)



Modules

```
const calc = require('./calculator');  
console.log(calc);  
console.log(calc.add(1, 2));  
console.log(calc.multiply(1, 2));
```

app.js

- ❖ calculator.js 파일로부터 모듈 로딩
- ❖ exports 객체를 로딩(만약 실행 코드가 있으면 실행)
- ❖ calc 변수에 exports 객체를 할당

```
$ node app.js
```



Callback

■ What is callback?

- 다른 코드의 인수로서 넘겨주는 실행 가능한 코드
- 콜백을 넘겨받은 코드는 이 콜백을 필요에 따라 즉시 실행 할 수도 있고, 나중에 실행 할 수도 있다
- 자바스크립트에서는 콜백으로 함수를 전달한다
- es6에서는 **arrow function**(화살표 함수)를 전달 할 수 있다
- 자바스크립트에서 콜백 함수는 비동기 함수
- A callback function is called at the completion of a given task.
- Node makes heavy use of callbacks. All the APIs of Node are written in such a way that they support callbacks.



Callback

- blocking code example

```
var fs = require("fs");  
var data = fs.readFileSync('input.txt');  
  
console.log(data.toString());  
console.log("Program Ended");
```



Callback

- non-blocking code example

```
var fs = require("fs");

fs.readFile('input.txt', function (err, data) {
  if (err) return console.error(err);
  console.log(data.toString());
});

console.log("Program Ended");
```

- ❖ readFile 함수의 두번째 인자로 **callback** 함수를 전달
- ❖ readFile 함수는 실행이 완료되지 않은 채로 바로 리턴
- ❖ 파일 읽기가 끝나면 **callback** 함수 실행

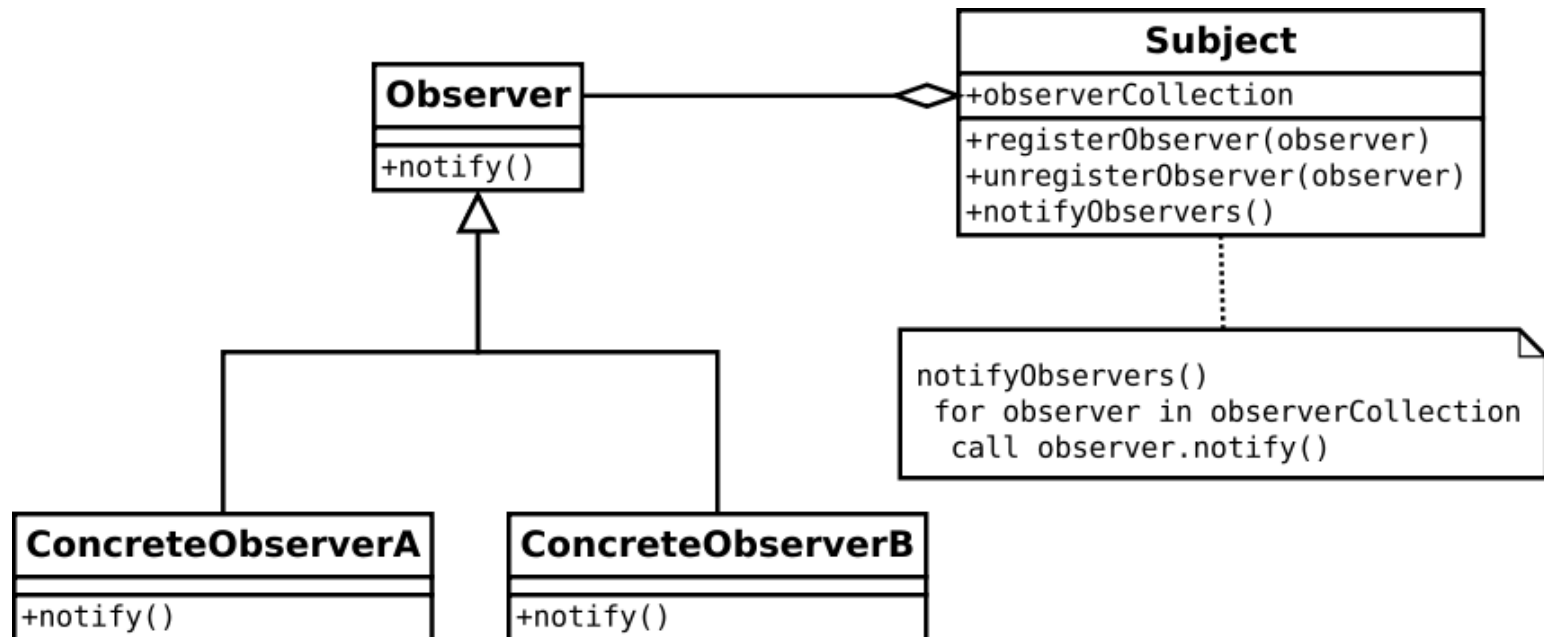


Event Loop

- event loop
 - Node.js is a single-threaded application
 - but it can support concurrency via the concept of **event** and **callbacks**.
 - Every API of Node.js is asynchronous and being single-threaded, they use **async function calls** to maintain concurrency.
 - Node thread keeps an event loop and whenever a task gets completed, it fires the corresponding event which signals the event-listener function to execute.

Event Loop

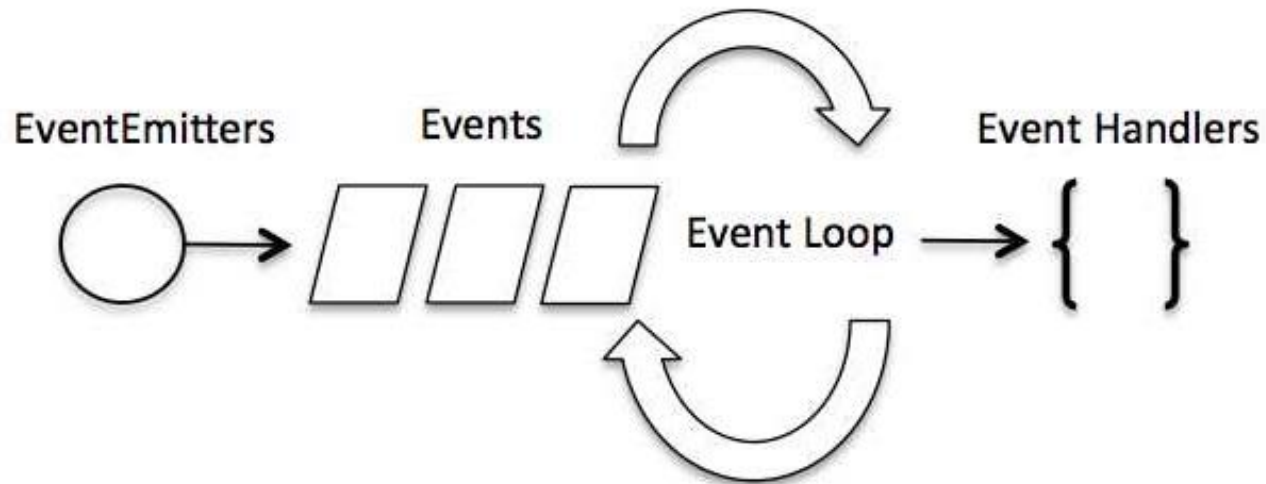
■ observer pattern



- ❖ 옵저버 패턴(observer pattern)은 객체의 상태 변화를 관찰하는 관찰자들, 즉 옵저버들의 목록을 객체에 등록하여 상태 변화가 있을 때마다 메서드 등을 통해 객체가 직접 목록의 각 옵저버에게 통지하도록 하는 디자인 패턴이다. 주로 분산 이벤트 핸들링 시스템을 구현하는 데 사용된다. 발행/구독 모델로 알려져 있기도 하다.

Event Loop

- event-driven programming
 - Node.js uses events heavily and it is also one of the reasons why Node.js is pretty fast compared to other similar technologies
 - As soon as Node starts its server, it simply initiates its variables, declares functions and then simply waits for the event to occur.
 - In an event-driven application, there is generally a main loop that listens for events, and then triggers a callback function when one of those events is detected.





Event Loop

■ code example

```
// Import events module
var events = require('events');

// Create an EventEmitter object
var EventEmitter = new events.EventEmitter();

// Create an event handler as follows
var connectHandler = function connected() {
    console.log('connection succesful.');
```

```
    // Fire the data_received event
    EventEmitter.emit('data_received', {data:'hello world'});
}

// Bind the connection event with the handler
EventEmitter.on('connection', connectHandler);
// Bind the data_received event with the anonymous function
EventEmitter.on('data_received', function(data) {
    console.log(`${data.data} data received succesfully.`);
});

// Fire the connection event
EventEmitter.emit('connection');

console.log("Program Ended.");
```



Event Loop

■ code example

```
var EventEmitter = require('events');  
// 1. setInterval 함수가 동작하는 interval 값을 설정합니다. 1초에 한번씩 호출  
var sec = 1;  
  
// 2. timer변수를 EventEmitter 로 초기화  
exports.timer = new EventEmitter();  
  
// 3. javascript 내장함수인 setInterval 을 사용해서 1초에 한번씩 timer 객체에  
tick 이벤트 발생  
setInterval(function(){  
    exports.timer.emit('tick', {time:sec++*1000});  
}, 1000);
```

custom_timer.js

```
var custom_module = require('./custom_timer');  
  
custom_module.timer.on('tick', function(data){  
    console.log(data.time);  
});
```

call_timer.js



Buffer

■ What is Buffer?

- 바이너리 데이터들의 **stream**을 읽거나, 조작하는 매커니즘
- **Buffer**클래스는 **Node.js**의 일부로 도입되어 **TCP 스트림**이나 파일시스템 같은 작업에서의 **octet 스트림**과의 상호작용을 가능하게 하기 위해서 만들어 졌다.
- With **TypedArray** now available, the **Buffer** class implements the **Uint8Array** API in a manner that is more optimized and suitable for **Node.js**.
- Instances of the **Buffer** class are similar to arrays of integers from 0 to 255 (other integers are coerced to this range by **& 255** operation) but correspond to fixed-sized, raw memory allocations outside the **V8 heap**
- The size of the **Buffer** is established when it is created and cannot be changed.



Buffer

■ buffer allocation

```
// Creates a zero-filled Buffer of length 10.
const buf1 = Buffer.alloc(10);

// Creates a Buffer of length 10, filled with 0x1.
const buf2 = Buffer.alloc(10, 1);

// Creates an uninitialized buffer of length 10.
// This is faster than calling Buffer.alloc() but the returned
// Buffer instance might contain old data that needs to be
// overwritten using either fill() or write().
const buf3 = Buffer.allocUnsafe(10);

// Creates a Buffer containing [0x1, 0x2, 0x3].
const buf4 = Buffer.from([1, 2, 3]);

// Creates a Buffer containing UTF-8 bytes [0x74, 0xc3, 0xa9, 0x73, 0x74].
const buf5 = Buffer.from('test');

// Creates a Buffer containing Latin-1 bytes [0x74, 0xe9, 0x73, 0x74].
const buf6 = Buffer.from('test', 'latin1');
```



Buffer

- Node.js 하위 버전에서는 **Buffer** 클래스의 생성자를 이용하여 버퍼를 생성

- ❖ `Buffer.from(array)` returns a new Buffer that contains a copy of the provided octets.
- ❖ `Buffer.from(arrayBuffer[, byteOffset[, length]])` returns a new Buffer that shares the same allocated memory as the given ArrayBuffer.
- ❖ `Buffer.from(buffer)` returns a new Buffer that contains a copy of the contents of the given Buffer.
- ❖ `Buffer.from(string[, encoding])` returns a new Buffer that contains a copy of the provided string.
- ❖ `Buffer.alloc(size[, fill[, encoding]])` returns a new initialized Buffer of the specified size. This method is slower than `Buffer.allocUnsafe(size)` but guarantees that newly created Buffer instances never contain old data that is potentially sensitive.
- ❖ `Buffer.allocUnsafe(size)` and `Buffer.allocUnsafeSlow(size)` each return a new uninitialized Buffer of the specified size. Because the Buffer is uninitialized, the allocated segment of memory might contain old data that is potentially sensitive.



Buffer

- character encodings
 - When string data is stored in or extracted out of a **Buffer** instance, a character encoding may be specified.

```
const buf = Buffer.from('hello world', 'ascii');

console.log(buf.toString('hex'));
// Prints: 68656c6c6f20776f726c64
console.log(buf.toString('base64'));
// Prints: aGVsbG8gd29ybGQ=

console.log(Buffer.from('fhqwhgads', 'ascii'));
// Prints: <Buffer 66 68 71 77 68 67 61 64 73>
console.log(Buffer.from('fhqwhgads', 'utf16le'));
// Prints: <Buffer 66 00 68 00 71 00 77 00 68 00 67 00 61 00 64 00 73 00>
```




Buffer

- **Buffers and TypedArray**

- Buffer instances are also Uint8Array instances. However, there are subtle incompatibilities with TypedArray.
- For example, while `ArrayBuffer#slice()` creates a copy of the slice, the implementation of `Buffer#slice()` creates a view over the existing Buffer without copying, making `Buffer#slice()` far more efficient.



Buffer

```
const arr = new Uint16Array(2);

arr[0] = 5000;
arr[1] = 4000;

// Copies the contents of `arr`.
const buf1 = Buffer.from(arr);
// Shares memory with `arr`.
const buf2 = Buffer.from(arr.buffer);

console.log(buf1);
// Prints: <Buffer 88 a0>
console.log(buf2);
// Prints: <Buffer 88 13 a0 0f>

arr[1] = 6000;

console.log(buf1);
// Prints: <Buffer 88 a0>
console.log(buf2);
// Prints: <Buffer 88 13 70 17>
```



Buffer

■ Buffers and Iteration

- Buffer instances can be iterated over using for..of syntax
- for of는 ECMAScript2015(es6) 문법

```
const buf = Buffer.from([1, 2, 3]);  
  
for (const b of buf) {  
  console.log(b);  
}  
// Prints:  
// 1  
// 2  
// 3
```



File System

- Definition

- The fs module provides an API for interacting with the file system in a manner closely modeled around standard POSIX functions.
- To use this module: `const fs = require('fs');`
- All file system operations have synchronous and asynchronous forms.
- The asynchronous form always takes a completion callback as its last argument.



File System

■ Read a File in Node.js

- blocking and non-blocking, synchronous, asynchronous
- blocking은 파일 읽기가 끝날 때 까지 실행의 흐름이 멈춰 있는 것
- non-blocking은 함수 실행 후 바로 다음 실행 문장으로 흐르는 것

```
const buf = Buffer.from('hello world', 'ascii');

console.log(buf.toString('hex'));
// Prints: 68656c6c6f20776f726c64
console.log(buf.toString('base64'));
// Prints: aGVsbG8gd29ybGQ=

console.log(Buffer.from('fhqwhgads', 'ascii'));
// Prints: <Buffer 66 68 71 77 68 67 61 64 73>
console.log(Buffer.from('fhqwhgads', 'utf16le'));
// Prints: <Buffer 66 00 68 00 71 00 77 00 68 00 67 00 61 00 64 00 73 00>
```



File System

- Read a File in asynchronously

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

fs.readFile(path.join(__dirname, "DATA"), 'utf8', function(err, contents){
  console.log(contents);
});
```

- Read a File in synchronously

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

var contents = fs.readFileSync('DATA', 'utf8');

console.log(contents);
```



File System

- write a file in asynchronously

```
const fs = require('fs');
var file_contents = "This is test file to test fs module of Node.js";
fs.writeFile('contents.txt', file_contents, 'utf8', function(error, data){
  if (error) {throw error};
  console.log("ASync Write Complete");
});
```

- write a file in synchronously

```
const fs = require('fs');
var file_contents = "This is test file to test fs module of Node.js";
fs.writeFileSync('sync_contents.txt', file_contents, 'utf8');
console.log("Complete");
```



File System

- Open File
 - Asynchronous file open
 - Alternatively, you can open a file for reading or writing using `fs.open()` method.

```
fs.open(path, flags[, mode], callback)
```

parameter	description	비고
path	Full path with name of the file as a string.	
flags	The flag to perform operation	
mode	The mode for read, write or readwrite. Defaults to 0666 readwrite.	
callback	A function with two parameters <code>err</code> and <code>fd</code> . This will get called when file open operation completes.	



File System

- open 함수를 이용한 file read

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

fs.open('myfile', 'r', function(err, fd){
  if(err){
    if(err.code==='ENOENT'){
      console.error('myfile does not exist');
      return;
    }
    throw err;
  }
  let buf = Buffer.alloc(100);
  let offset = 0;
  fs.read(fd, buf, offset, buf.length, null, function(error, bytesRead, buffer){
    if (error){
      console.log(error);
      return;
    }
    let data = buffer.toString('utf8');
    console.log(data);
  });
});
```