KWEB Study: Week 2

1. Before Study
2. More in Javascript
3. ECMAScript
4. Asynchronous
5. 실습 및 과제

ECMAScript, and Async

KWEB 2학기 준회원 스터디

Designed by M.G. BAE

Before Study

- 오랜만이네요, 여러분! 추석은 잘 보내셨나요?? (어떻게 내용들은 잘 까먹으셨는지..ㅎㅎㅎ)

- 1주차에선 Node.js의 Module, yarn, Git 기초에 대해 배우고 실습해봤는데 기억하시나요?!
  - ✓ 과제 생각보다 제출 잘하셔서 놀랬어요! (앞으로도 GitHub 계속 사용해도 될 듯 합니다~)

- 2주차의 목표는 Modern Javascript! Javascript의 최신 문법들에 대해 배워볼 것입니다.
  - ✓ 지금까지 너무 예전 스타일의 Javascript만 취급해서 현업 코드와 거리가 너무 멀었어요.. 이번 주를 통해 최신 Javascript의 사용법들에 대해 알아봅시다!
  - ✓ 지난 주와 반대로 오늘은 이론의 비중이 더 높을 예정입니다~ (그치만 빠르게 넘어갈.. 크흠흠)

- 추석도 끝났으니 오늘은 약간 2학기 스터디를 좀 더 테크니컬하게 해줄 여러가지 것들을 대해 배워봅시다~ 먼저, Javascript에서 객체와 관련된 부분부터 시작하겠습니다!

Function

- Javascript에서 함수는 작업을 수행하고 값을 반환할 수도 있습니다.
  - ✓ Input을 받고 Output을 주는 함수
  - ✓ Input을 받고 Output을 주지 않는 함수
  - ✓ Input을 받지않고 Output을 주는 함수
  - ✓ Input을 받지않고 Output을 주지 않는 함수

- 함수는 모든 객체에서 사용 가능하므로 함수를 "전역 메소드"라고도 합니다.

- 객체와 관련한 부분부터 시작한다면서 왜 함수부터 짚었냐? → Javascript에서 함수도 객체입니다!

```
// function that takes input and returns output
function func1(a) {
    return a + 1;
}

// function that takes input, but doesn't return output
function func2(a) {
    console.log(a);
}

// function that doesn't take input, but returns output
function func3() {
    return 1;
}

// function that doesn't take input and doesn't return output
function func2() {
    console.log(1);
}
```

Function

- 정확하게는! Javascript의 함수는 1급 객체입니다.
  - ✓ 또한, 1급 함수이기도 합니다. 궁금하면 검색 ㄱㄱ
- 1급 객체가 무엇인가? → 아래 조건 3가지를 만족하며 Object 이면 됩니다. 자세한건 프언 시간에~
  - ✓ stored in a variable
  - ✓ passed as an argument
  - ✓ returned from other functions
- 고로, Javascript에서는 함수를 변수와 인자로써 담을 수 있습니다. → 오른쪽 코드 참고!
  - ✓ 함수는 변수와 같은 하나의 객체로 인식됩니다. 함수를 하나의 객체로 인식하기 때문에 이를 변수(객체로) 저장 할 수 있습니다.

```
function func1(a) {
    return a + 1;
}
func1(1);
// output: 2

const func1_as_variable = func1;
func1_as_variable(1);
// output: 2

function func_new(a,b,f) {
    return a + f(b);
}
func_new(1,1,func1);
// output: 3 ← (1 + func1(1))
```

잠깐, Clojure

- 1급 객체 함수의 개념을 이용하여 **스코프(scope)에 묶인 변수를 바인딩** 하기 위한 일종의 기술
  - ✓ 기능상으로, 클로저는 함수를 저장한 레코드(record)이며, 스코프(scope)의 인수(factor)들은 클로저가 만들어질 때 정의(define)되며, 스코프 내의 영역이 소멸(remove)되었어도 그에 대한 접근(access)은 독립된 복사본인 클로저를 통해 이루어질 수 있다.

- 아래는 책과 주요 사이트에 쓰여있는 클로저의 개념입니다.
  - ✓ 함수 객체와, 함수의 변수가 해석되는 유효범위
  - ✓ 독립적인 변수를 참조하는 함수들이다.
  - ✓ 내부함수가 외부함수의 맥락(context) 에 접근할 수 있는 것
  - ✓ 함수와 함수가 선언된 어휘적 환경의 조합이다.

- 잘 모르겠으니, 실습 시간에 무슨 말인지 보여 드릴게요! → 그냥 그렇구나 하고 알고 있기만 하면 됩니다.
  - ✓ (여담으로, 다른 프로그래밍 언어에는 Closure이지만 Javascript만 Clojure입니다.)

Object

- Javascript는 객체(Object) 기반의 스크립트 프로그래밍 언어입니다.

- 우리가 살아가는 세상은 많은 객체 들이 상호 작용하며 돌아가며, 객체 지향은 이러한 개념을 프로그래밍에 가져온 것입니다. → 대표적인 객체지향 프로그래밍 언어로 Java가 있죠!

- 객체지향 언어에서 객체는 데이터(주체)와 그 데이터에 관련되는 동작(절차,방법,기능)을 모두 포함하고 있는 개념입니다.
  - ✓ 예를 들어, Javascript에서 문자열 변수 "TEXT" 는 내용인 'text'를 속성값으로 문자열 처리에 관련된 함수를 메소드로 가지는 객체입니다.

- 정리하자면, 모든 객체는 자신의 속성값(Property)과 행동을 정의 하는 메소드(Method)를 가집니다.
  - ✓ Property: 객체의 속성을 나타내는 접근 가능한 이름과 활용 가능한 값을 가지는 특별한 형태
  - ✓ Method: 객체가 가지고 있는 동작

참고! JSON

- JSON(JavaScript Object Notation): 속성-값 쌍으로 이루어진 데이터 오브젝트를 전달하기 위해 인간이 읽을 수 있는 텍스트를 사용하는 개방형 표준 포맷
  - ✓ JSON is a syntax for storing and exchanging data. JSON is text, written with JavaScript object notation.

- JSON이 XML을 대체해서 설정의 저장이나 데이터를 전송 등에 많이 사용됩니다. 앞으로 자주 볼 형태이기 때문에 먼저 소개합니다.

- 오른쪽 코드와 같은 형식을 사용합니다.
  - ✓ package.json 생각해보세요!

```
{
    "glossary": {
        "title": "example glossary",
        "GlossDiv": {
            "title": "S",
            "GlossList": {
                "GlossEntry": {
                    "ID": "SGML",
                    "SortAs": "SGML",
                    "GlossTerm": "Standard Generalized Markup Language",
                    "Acronym": "SGML",
                    "Abbrev": "ISO 8879:1986",
                    "GlossDef": {
                        "para": "A meta-markup language, used to create markup languages such as DocBook.",
                        "GlossSeeAlso": ["GML", "XML"]
                    },
                    "GlossSee": "markup"
                }
            }
        }
    }
}
```

Object 구현

- Javascript에서 객체를 구현하는 방식입니다. (여러 방식이 있겠죠?)
  - ✓ 방법 1: 변수로 Object 선언
  - ✓ 방법 2: this 키워드로 속성, 메소드 정의 후 생성자로 생성
  - ✓ 방법 3: this 키워드로 속성, prototype 키워드로 메소드 정의 후 생성자로 생성

```
// 방법 1
const Team = {
    name: "Team",
    height: 170,
    weight: 65,
    eat: function() { this.weight += 1; },
    diet: function() { this.weight -= 1; }
}
Team.height = 173;
Team.eat();
```

```
// 방법 2
function Person(name, height, weight) {
    this.name = name,
    this.height = height,
    this.weight = weight,
    this.eat = function() { this.weight += 1; },
    this.diet = function() { this.weight -= 1; }
}

const Team = new Person("Team",170,65);
Team.height = 173;
Team.eat();
```

```
// 방법 3
function Person(name, height, weight) {
    this.name = name;
    this.height = height;
    this.weight = weight;
}

Person.prototype.eat = function() {
    this.weight += 1;
}

Person.prototype.diet = function() {
    this.weight-= 1;
}

const Team = new Person("Team",170,65);
Team.height = 173;
Team.eat();
```

Class

- 그리고 한가지 더! ES 2015부터 지원하는 방식이 있습니다!
  → Class 방식 이용
  - ✓ ES 2015가 뭔지는 조금 이따 살펴봅시다~

- class는 우리에게 익숙한 객체지향형 프로그래밍의 대표적인 키워드입니다.
  - ✓ 오른쪽 코드를 보면 엄청 깔끔해졌죠? 앞의 코드들이랑 완전히 똑같이 동작합니다!

- 사용법은 여타 프로그래밍 언어랑 비슷하니! 각자 알아보세요~
  - ✓ (구글링도 개발자의 능력입니다.)

- 참고! 명시적으로만 class지 Javascript 내부를 뜯어보면 여전히 prototype으로 구현되어 있습니다.

```
class Person {
    constructor (name, height, weight) {
        this.name = name;
        this.height = height;
        this.weight = weight;
    }
    eat() {
        this.weight += 1;
    }

    diet() {
        this.weight-= 1;
    }
}

const Team = new Person("Team",170,65);
Team.height = 173;
Team.eat();
```

Typescript

- 직전 슬라이드의 코드를 Typescript로 작성해보았습니다!
  - ✓ 지금까지 본 5종류의 코드가 사실상 동일하게 동작합니다!

- Typescript는 Javascript의 superset으로 타입 시스템 등을 지원합니다. Javascript보다 좀 더 확실하고 체계적이게 개발할 수 있죠!
  - ✓ 요새 웹 개발에서 심심치 않게 사용됩니다.
  - ✓ Typescript는 컴파일이 필요합니다. (웹에서 쓰려면 Javascript로 컴파일을 해야 쓸 수 있어요!)

- Front-end Framework인 Angular에서 거의 필수적으로 사용되며, 요새는 공식적으로 지원하는 Framework들이 많습니다.

- 준스에선 Typescript가 있다는 것 정도만 알고 넘어가시면 됩니다.

```
class Person {
    private name: string;
    private height: number;
    private weight: number;

    constructor (name, height, weight) {
        this.name = name;
        this.height = height;
        this.weight = weight;
    }
    public eat() {
        this.weight += 1;
    }

    public diet() {
        this.weight-= 1;
    }
}

const Team = new Person("Team",170,65);
Team.height = 173;
Team.eat();
```

More

- 지금까지 Javascript에서 객체와 관련된 부분들을 살펴보았습니다. 그것들 이외에도 몇 가지만 좀 더 짚고 넘어가봅시다!

- 삼항 연산자: 표현식이 참이면 앞의 것, 거짓이면 뒤에 것을 실행됩니다.
  - ✓ 형식: expression ? A : B → ex) console.log(1 < 10 ? “1이 작음” : “10이 작음”); //실행결과는 ‘1이 작음’

- 비교 연산자: === / !== vs. == / != → **===과 !==의 사용을 권장합니다.**
  - ✓ ===과 !==은 값과 타입을 모두 비교 → ex) “1” === 1(false), 1 === 1 (true)
  - ✓ ==과 !=은 값만 비교 → ex) “1” === 1(true), 1 === 1 (true)

- 예외처리: try...catch 구문 (finally 키워드도 추가되긴 하였으나 오늘은 다루지 않겠습니다.)
  - ✓ try문을 실행하다 오류가 발생하면 catch문을 실행시킵니다. Java에서의 예외처리 문법과 비슷합니다.
  - ✓ 형식: try { … } catch (e) { … } → 예제는 뒤에 실습에서 직접 사용해봅시다!

More

- 익명 함수: 진짜 말 그대로 이름이 없어서 익명함수입니다.
  - ✓ 형식: function() { … } → ex) const myFunc = function() { alert("Hello, world!"); };
  - ✓ (사실 여러분 알게 모르게 이미 스터디에서 봤고 썼어요..ㅎ)

- IIFE (즉시 실행함수, Immediately Invoked Function Expressions): 이것도 즉시 실행 되는 함수입니다.
  - ✓ 형식: (function() { … })(); → ex) (function() { alert("Hello, world!"); })();
  - ✓ Local scope를 만들 수 있습니다. → 일반적인 함수랑 this가 가리키는 것이 달라집니다.

- this 키워드 → 굉장히 중요한 내용인데 이번주에 설명하면 너무 오래 걸려서 다음주로 Pass!

- hoisting → 마찬가지로! 다음주로 Pass!

ECMAScript

- Ecma International의 ECMA-262 기술 규격에 정의된 표준화한 스크립트 언어
  - ✓ 한마디로! 그냥 표준 규격입니다. Javscript의 토대!

- 2015년부터 매년 새로운 표준 발표 → ES 5과 ES 6 사이는 차이가 큽니다!
  - ✓ ES 2015 === ES 6, 올해는 ES 2018 (ES 9)가 발표되었습니다.

- 그럼 Javascript랑 ECMAScript는 어떤 관계가 있는지? → Javascript는 ECMAScript의 구현체 정도!
  - ✓ ES 명세가 발표되자마자 브라우저에 바로 반영되진 않고! 구현되는데는 시간이 약간 걸립니다.
  - ✓ 사실 브라우저마다 Javascript를 통해 지원하는 정도도 다 다릅니다. (IE 극혐.. 퇴출 좀)
  - ✓ Javascript 말고도 ECMAScript 표준을 따르는 스크립트 언어들도 존재합니다.

- 오늘의 남은 이론 시간은 지금까지 배운 Javascript 말고 좀 더 현대적인 ES 명세들을 반영한 Javascript 의 문법에 대해 다뤄보겠습니다!

let & const (ES 6 ~)

- Javascript의 새로운! 선언자들이며, let과 const의 일반적 사용법은 아래와 같습니다.
  - ✓ let: 변수형 → let a = 1; let b = “String”;
  - ✓ const: 상수, 참조형 → const A = 2018; const fs = require(‘fs’);

- let과 const 모두 기존의 var와 다르게 변수 재선언 불가능합니다. → 오른쪽 코드 참조!

- let과 const 비교
  - ✓ let: 변수의 재선언은 불가하나 재할당이 가능, 선언만 하고 나중에 값 할당 가능 (var과 차이 → 할당 전 반드시 선언 필요)
  - ✓ const는 변수 재선언, 재할당 모두 불가능, 선언과 동시에 값을 할당해야함

```
// 이미 만들어진 변수이름으로 재선언했는데 아무런 문제가 발생하지 않는다.
var a = 'test'
var a = 'test2'

// hoisting으로 인해 ReferenceError에러가 안난다.
c = 'test'
var c

// let
let d = 'test'
let d = 'test2' // Uncaught SyntaxError: Identifier 'a' has already been declared
d = 'test3'     // 가능

// const
const e = 'test'
const e = 'test2' // Uncaught SyntaxError: Identifier 'a' has already been declared
e = 'test3'    // Uncaught TypeError:Assignment to constant variable.

f = 'test' // ReferenceError: c is not defined
let f

// let은 선언하고 나중에 값을 할당이 가능하지만
let dd
dd = 'test'

// const 선언과 동시에 값을 할당 해야한다.
const aa // Missing initializer in const declaration
```

let & const vs. var (ES 6 ~)

- var이 잘 디자인되었으면 굳이 let & const가 등장하지 않았겠죠? 앞 슬라이드의 코드에서 나왔듯이 아래와 같은 문제점이 있었습니다.
  - ✓ 이미 만들어진 변수이름으로 재선언해도 문제가 발생하지 않음
  - ✓ hoisting으로 인해 ReferenceError에러가 나지 않음 (앞에서 배운 그 hoisting입니다!)

- ES 6부턴 var 사용은 자제하고 let, const 사용이 권장됩니다.
  - ✓ Javascript가 욕 먹던 이유 중 하나가 var 입니다.. 관리가 어렵자나요..

- let과 const는 Block scope를 가지며, var는 function scope입니다.
  - ✓ hoisting이 이 단위로 이루어집니다.

참고! Function scope vs. Block scope

- var은 Function scope → 선언된 것이 함수 단위로 살아있거나 사라집니다.

```
var a = 1;
if (true) {
    var a = 2;
    console.log(a); // 2
}
console.log(a); // 2
```

- let과 const는 Block scope → 선언된 것이 블록 단위로 살아있거나 사라집니다.

```
const a = 1;
let b = 10;
if (true) {
    const a = 2;
    let b = 20;
    console.log(a); // 2
    console.log(b); // 20
}
console.log(a); // 1
console.log(b); // 10
```

Template Literals (ES 6 ~)

- 반드시! 백틱(`)을 써야 사용 가능합니다.
  - ✓ `의 명칭은 grave accent이고 프로그래밍에선 backquote 혹은 backtick으로 알려져 있음!
  - ✓ 추가적으로 ` 는 보통 문장에 쓰이지 않으니 “와 ‘에 비해 신경 쓸일이 줄어듭니다.
- Multi-line string도 지원하며, 문자열에 표현식(expression)을 쉽게 삽입할 수 있게 해줍니다.
- 중첩(Nesting templates)도 가능합니다.
- 참고! Tagged Templates도 있는데 궁금하시면 알아보세요~

```
console.log("string text line 1\n"+
"string text line 2");
// 위와 아래의 코드는 동일하게 동작합니다.
console.log(`string text line 1
string text line 2`);
/*
"string text line 1
string text line 2"
*/
// ----------------------------------------------------------------------
const a = 5;
const b = 10;

console.log("Fifteen is " + (a + b) + " and\nnot " + (2 * a + b) + ".");
// 위와 아래의 코드는 동일하게 동작합니다.
console.log(`Fifteen is ${a + b} and
not ${2 * a + b}.`);
/*
"Fifteen is 15 and
not 20."
*/
// ----------------------------------------------------------------------
// 중첩도 가능합니다.
const classes = `header ${ isLargeScreen() ? '' :
`icon-${item.isCollapsed ? 'expander' : 'collapser'}` }`;
```

Arrow function (ES 6 ~)

- 짧게 설명하면 기존 함수 선언을 화살표(=>)를 통해 선언해주는 것입니다. (익명함수?)

- Lexical scoping → this 키워드 사용과 관련
  - ✓ 소스코드가 작성된 그 문맥에 결정됩니다.
  - ✓ (대부분은 dynamic scoping을 많이 사용했었고 현대 프로그래밍 언어에서 이를 사용합니다.)
  - ✓ 자세하게는 Javascript 공부를 좀 더 하시면서 알아보세요~
  - ✓ **중요!** 자신만의 this를 생성하지 않고 자신을 포함하고 있는 context의 this를 이어 받습니다.

- 형식은 오른쪽 코드 참고하세요!

```
(param1, param2, …, paramN) => { statements }
(param1, param2, …, paramN) => expression // 다음과 동일함:  => { return expression; }

// 매개변수가 하나뿐인 경우 괄호는 선택사항:
(singleParam) => { statements } === singleParam => { statements }

// 매개변수가 없는 함수는 괄호가 필요:
() => { statements }

// 객체 리터럴 식을 반환하는 본문(body)을 괄호 속에 넣음:
params => ({foo: bar})

// 나머지 매개변수 및 기본 매개변수가 지원됨
(param1, param2, ...rest) => { statements }
(param1 = defaultValue1, param2, …, paramN = defaultValueN) => { statements }

// 매개변수 목록 내 구조분해도 지원됨
const f = ([a, b] = [1, 2], {x: c} = {x: a + b}) => a + b + c;
f();  // 6

const materials = [
    "Hydrogen",
    "Helium",
    "Lithium",
    "Beryllium"
];

const materialsLength1 = materials.map(function(material){
   return material.length
});
const materialsLength2 = materials.map((material)=>{
   return material.length
});
const materialsLength3 = materials.map(material=> material.length);
```

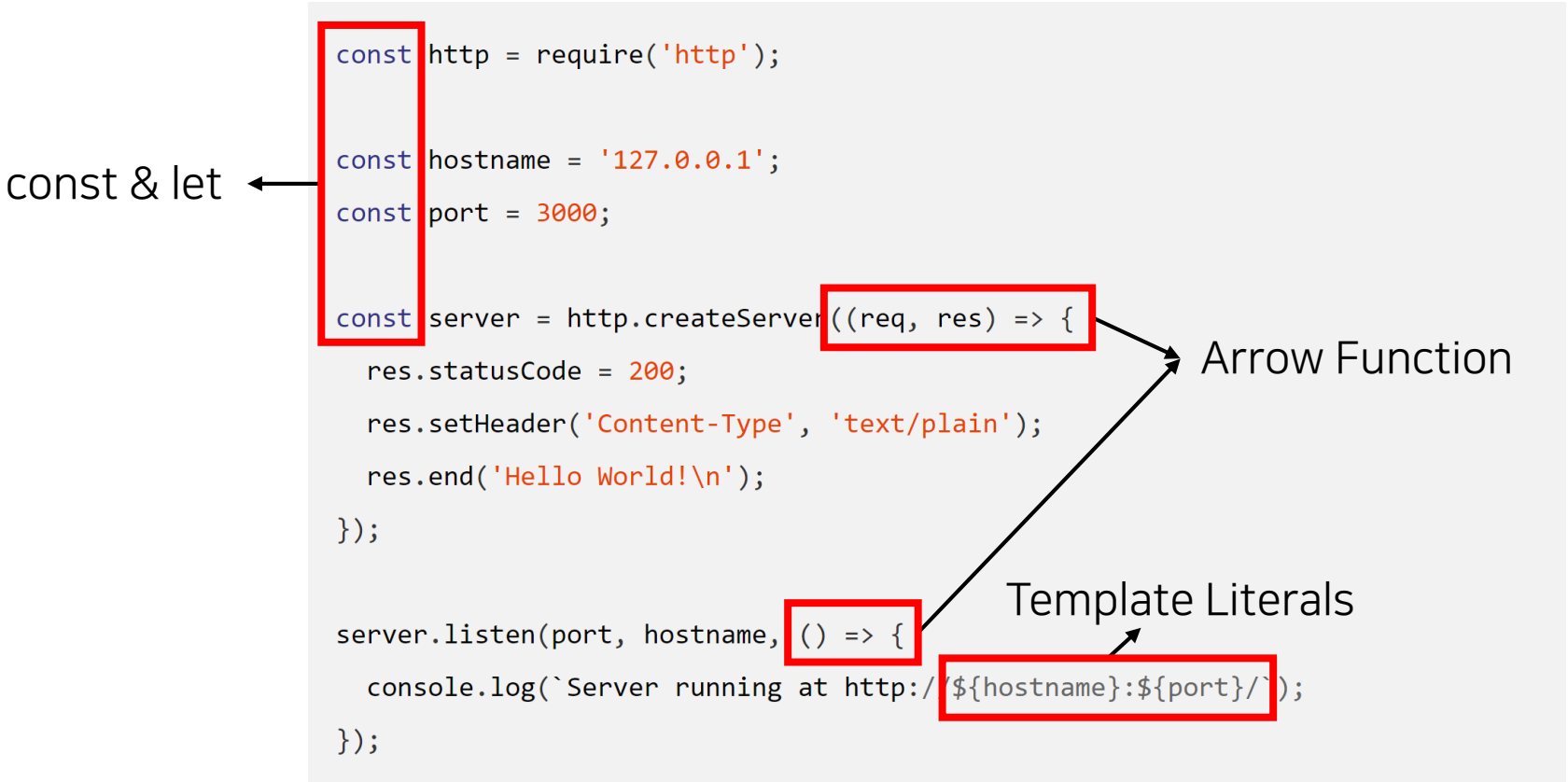
Destructuring Assignment (ES 6 ~)

- 배열이나 객체에 있는 각 데이터의 값을 자동으로 추출하기 위한 방법입니다.
- 기존 Javascript에서는 구조화된 할당만 지원했으므로 하나하나 지정해서 값을 추출했었습니다.
- ES 6부터 비구조화 할당을 통해 굉장히 편해졌습니다.
  - ✓ 배열: 길이가 일치하지 않아도 남는 부분은 무시하고 할당하거나, 할당할 배열에서 해당 위치의 값은 제외하고 할당할 수도 있습니다.
  - ✓ 객체: **변수명과 객체 속성의 이름이 같아야 합니다.** 오른쪽 코드와 같이 변경할 수는 있습니다.

```
// Array
let numbers = [1, 2, 3];
let [a, b] = numbers;
let [c, , d] = numbers;

console.log(`${a}, ${b}`); // "1, 2"
console.log(`${c}, ${d}`); // "1, 3"

let numbers = [1, 2, 3];
let [a, ...b] = numbers;

console.log(a); // 1
console.log(b); // [2,3]

// Object
let book = {
    title : 'HTML',
    price : 23000
};
let {title, price} = book; // book객체의 속성을 개별적으로 좌측의 변수에 전달
console.log(`${title}, ${price}`); // "HTML, 23000"

let book = {
    title : 'HTML',
    price : 23000
};
let {title: newTitle, price: newPrice} = book;
console.log(`${newTitle}, ${newPrice}`);
```

Node.js 예제 코드

- 아래는 맨날 쓰던 예제 코드입니다. 알게 모르게 이미 우리는 최신 문법들을 사용했습니다!




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

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World!\n');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

More

- ‘use strict’: 더 이상 어느 정도의 에러를 허용하지 않게 만듭니다. (hoisting을 막기 위해서도 사용됨.)
  - ✓ 기존에는 조용히 무시되던 에러들을 throwing합니다.
  - ✓ JavaScript 엔진의 최적화 작업을 어렵게 만드는 실수들을 바로잡습니다.
  - ✓ 엄격 모드는 ECMAScript의 차기 버전들에서 정의 될 문법을 금지합니다.

- import (ES 6 ~): Module이나 파일을 포함시키는 새로운 방법입니다.
  - ✓ const http = require(‘http’); === import http from ‘http’;
  - ✓ Node.js에서는 아직 require가 좀 더 일반적입니다. (물론, import 방식도 지원은 시작했습니다.)

- Babel: ES 6 이후의 문법을 ES 5 스타일로 변환해주는 도구, 브라우저 호환성을 위해 사용됩니다.
  - ✓ yarn이나 npm 을 통해 Node.js 프로젝트에 내려 받을 수 있습니다.

- ESLint: 자바스크립트 문법 중 에러가 있는 곳에 표시를 달아놓는 도구, 개발 시에 활용됩니다.

Asynchronous 이해하기

- 다들 윈도우나 안드로이드 운영체제에서 어플리케이션을 사용하다가 '응답없음' 메시지가 뜨면서 갑자기 화면상의 UI가 작동하지 않는 것을 경험해보셨을 것입니다.

- 이는 프로그램의 흐름이 UI를 처리하는 로직 밖에서 오랜 시간 실행되어 UI 처리 로직이 정상적인 작동을 하지 못해 일어나는 겁니다. → (말이 어렵죠? 그냥 프로그램이 해야할 걸 못해서 에러난 겁니다.)

- 파일 읽기, 네트워크 통신(API 사용, DB 처리 등), 입출력(I/O)을 실행하면 대부분의 프로그램은 흐름에 지연이 발생합니다. → (컴과 OS 시간에 자세히 배웁니다.)

- 일괄작업(순차적으로 처리)을 이용하는 프로그램이라면 지연이 큰 문제가 없으나, 웹 브라우저와 같이 사용자와 상호작용하는 UI가 있는 환경에서 지연은 주요한 문제가 됩니다.

Asynchronous 이해하기

- 비동기의 개념에 대해 좀 더 와 닿을 수 있도록 일상에서 우리가 커피를 주문할 때를 생각해봅시다.

- 커피를 주문할 때 우리는 1. 어떤 커피를 주문할지 정하고, 2. 카운터에 가서 커피를 주문하고 3. 아래 2가지 중 하나의 일을 할 수 있습니다.

- 카운터 앞에서 커피가 나오기를 기다리거나 → **동기 (Synchronous)**
  - ✓ 진동벨을 주고 받지 않아도 되지만 커피가 나올 때까지 카운터 앞에서 쭉 대기해야 합니다. (카운터 앞에서 대기하면 다른 업무는 못 보겠죠?)

- 진동벨을 받아가서 다른 일을 하다가 커피를 받아가거나 → **비동기 (Asynchronous)**
  - ✓ 커피가 준비되는 동안 카운터 앞에서 대기할 필요없이 다른 일을 볼 수 있습니다.

Asynchronous 이해하기

- Chrome에서 개발자 도구를 열고, Console 탭에서 " while(1){} "을 입력해봅시다.
  - ✓ 개발자 도구 단축키: 윈도우 → F12 / Mac → Cmd + Option + I
  - ✓ while문을 실행하고 나면 아마 브라우저가 응답하지 않을거에요!

- 위의 결과는 Javascript의 끊임없이 돌아가는 무한 루프가 브라우저의 렌더링을 담당하는 코드로 실행 흐름을 넘겨주지 않아서 일어나는 현상입니다.

- 이와 같은 현상이 일어나지 않도록 하려고 등장한 개념이 "비동기" 입니다. → 한 작업에 묶임 ㄴㄴ
  - ✓ 비동기 구조는 다른 작업에 완료여부와 상관없이(Non-blocking) 제어권을 넘기는 방식입니다.
  - ✓ 이에 반해, 동기 구조는 순차적으로 진행되므로 앞선 작업이 끝나지 않으면(Blocking) 제어권을 넘기지 않습니다.

- 지금까지 비동기의 개념에 대해 이해해보려고 노력했는데 조금이나마 감은 잡으셨길 바랍니다! 그럼 지금부터는 Javascript (Node.js) 에서 비동기를 어떻게 구현하고 다루는지 살펴봅시다.

Callback

- Node.js의 전통적인 비동기 구현 방식입니다.
  - ✓ Node.js의 장점 중 하나가 비동기 I/O이므로 적절한 비동기 제어로 효율적이고 빠른 서버 구축 가능!

- 비동기 방식으로 만든 함수는 연산이 끝났을 때 parameter로 함수를 전달하는데, 이때 전달되는 함수를 바로 Callback 함수라고 합니다.
  - ✓ **Callback function**: 객체의 상태 변화(이벤트)가 발생한 경우에 전달되는 함수

- 참고로! 반환되는 함수안에도 새로운 함수가 있을 수도 있습니다. → Callback Hell

```
function sum_mod7_cb(a,b,callback) {
    const result = (a + b) % 7;
    callback(result);
}

sum_mod7_cb(a = 17, b = 191, function (result) {
    console.log(`Processing sum_mod7_cb\n`+
    `(${a} + ${b}) % 7 = ${result}`);
});
```

Callback Hell

- Callback Hell은 비동기 처리 Logic을 위해 Callback 함수를 연속해서 사용할 때 발생하는 문제입니다.
- 서버에서 데이터를 받아와 화면에 표시하기까지 인코딩, 사용자 인증 등을 처리해야 하는 경우가 많습니다.
- 이 모두를 비동기로 처리해야 한다면 Callback 안에 Callback을 계속 무는 형식이 됩니다.
  - ✓ 이러한 코드 구조는 가독성도 떨어지고 로직을 변경하기도 어렵습니다.
- 이러한 이유로 새로운 것들이 등장하게 되었습니다!

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

fs.writeFile('./1.txt', '11111', function(err) {
    console.log('1.txt created');
    fs.readFile('./1.txt','UTF-8', function(err,data1) {
        fs.writeFile('./2.txt',data1 + '22222', function(err) {
            console.log('2.txt created');
            fs.readFile('./2.txt','UTF-8', function(err,data2) {
                fs.writeFile('./3.txt',data2 + '33333', function(err) {
                    console.log('3.txt created');
                    fs.readFile('./3.txt','UTF-8', function(err,data3) {
                        console.log(data3);
                    });
                });
            });
        });
    });
});
```

Promise

- Promise는 자바스크립트 비동기 처리에 사용되는 객체입니다.

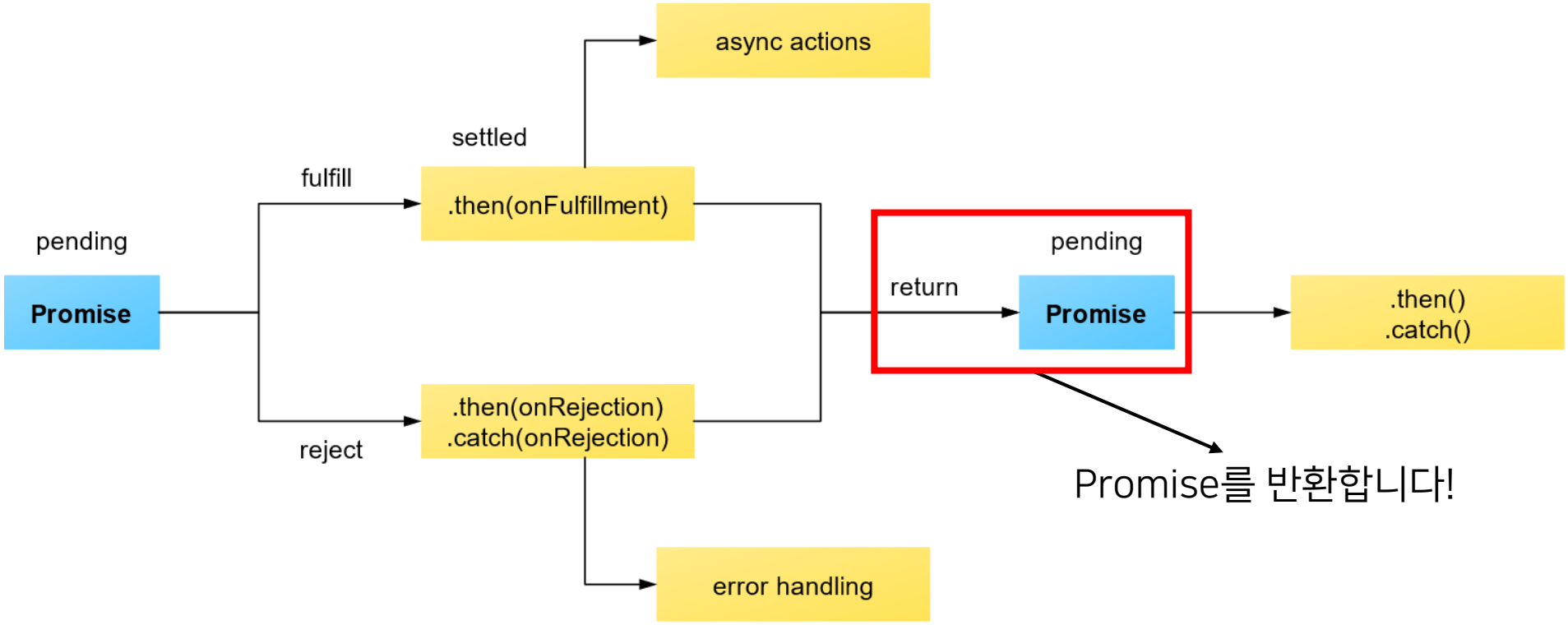
- 프로미스를 사용할 때 알아야 하는 가장 기본적인 개념이 바로 프로미스의 상태(처리 과정)입니다.

- Promise를 생성하고 종료될 때까지 3가지 상태를 갖습니다.
  - ✓ Pending(대기) : 비동기 처리 로직이 아직 완료되지 않은 상태 → new Promise로 선언
  - ✓ Fulfilled(이행) : 비동기 처리가 완료되어 프로미스가 결과 값을 반환해준 상태 → resolve로 처리되면 .then(…)
  - ✓ Rejected(실패) : 비동기 처리가 실패하거나 오류가 발생한 상태 → reject로 처리되면 .catch(…)

```
function getData() {
  return new Promise(function (resolve, reject) {
    $.get('localhost/users/1', function (response) {
      if (response) {
        resolve(response);
      }
      reject(new Error("Request is failed"));
    });
  });
}

// Fulfilled 또는 Rejected의 결과 값 출력
getData().then(function (data) {
  console.log(data); // response 값 출력
}).catch(function (err) {
  console.error(err); // Error 출력
});
```

Promise

- 아래 그림은 기본적인 Promise 처리 흐름입니다. (이 흐름만 잘 알면 Promise 마스터 클래스)



Promise

- Promise chaining → 여러 개의 프로미스를 연결하여 사용하는 것
  - ✓ 이걸 쓸 수 있으니 callback hell 해결가능
  - ✓ Chaining이 되는 이유 → then() 메서드를 호출하고 나면 새로운 프로미스 객체가 반환됩니다.

```
asyncThing1()
    .then(function() { return asyncThing2(); })
    .then(function() { return asyncThing2(); })
    .catch(function(err) { return asyncRecovery1(); })

    .then(function() { return asyncThing2(); }, function(err) { return asyncRecovery2(); })
    .catch(function(err) { console.log("Don't worry.."); })

    .then(function() { console.log("All done!"); });
```

- Promise가 좋은 것 같긴 한데, 뭔가 복잡하고 Callback Hell보단 좋지만 여전히 코드가 길죠?
  - ✓ 이를 개선해서 ES 8부터는 또다른 새로운 문법을 등장시켰습니다. → async / await

async / await

- Javascript에서 비동기를 구현하는 가장 최신 기법입니다! → ES 7부터 지원합니다.

- 기본적인 사용법은 async로 함수를 비동기 함수라고 표기하고, 내부에서 await로 비동기적 요청을 기다립니다.
  - ✓ await 키워드는 반드시 async 함수 내부에서만 사용할 수 있습니다.

- async / await 방식의 가장 큰 장점 중 하나는 비동기 코드의 겉모습과 동작을 좀 더 동기 코드와 유사하게 만들어준다는 것입니다.
  - ✓ 코드가 간결하고 깔끔해집니다!

- (사실 안을 까보면 Promise 개념 사용합니다..ㅎ)

```
// async / await
const makeRequest = async () => {
    const data = await getJSON();
    if (data.needsAnotherRequest) {
        const moreData = await makeAnotherRequest(data);
        console.log(moreData);
        return moreData;
    } else {
        console.log(data);
        return data;
    }
}

// Promise
const makeRequest = () => {
    return getJSON()
        .then(data => {
        if (data.needsAnotherRequest) {
            return makeAnotherRequest(data)
            .then(moreData => {
                console.log(moreData)
                return moreData
            });
        } else {
            console.log(data);
            return data;
        }
    });
}
```

async / await

- Production에선 현재 try...catch문과 같이 사용합니다. → async 함수에서 예기치 못한 오류 대비!
  - ✓ Production에선 모든 오류를 처리할 수 있어야..ㅎ
  - ✓ (아니면.. 갑자기 뻗어요..ㅎㅎㅎ)

- async / await 방식이 좋긴 한데 try...catch랑 같이 사용하면 성능 저하 이슈가 있습니다.
  - ✓ try...catch 때문에 callback 함수 방식보다 느릴 수 있습니다. → try...catch문이 느려요..ㅋㅋㅋ

- Callback 방식도 사실 엄청 쉽자나요? 비동기를 구현은 하나에 올인보다는 현재 시점에선 세 가지 방식 다 적절한 곳에 사용하는 것이 중요합니다.

```
const makeRequest = async () => {
    try {
        const data = await getJSON();
        if (data.needsAnotherRequest) {
            const moreData = await makeAnotherRequest(data);
            console.log(moreData)
            return moreData;
        } else {
            console.log(data);
            return data;
        }
    } catch (err) {
        console.log(err);
    }
}
```

실습

Asynchronous with Javascript

Designed by M.G. BAE

실습

- 오늘 배운게 뭔가 좀 많았죠? 고생하셨습니다!

- 실습에서는 앞서 배운 ECMAScript를 반영한 최신 Javascript의 문법들을 Node.js를 통해 사용해보고, 최종적으로는 fs 모듈을 활용해 비동기적으로 파일을 읽어오는 예제를 구현해볼 예정입니다.

- 이번주 실습은 크게 2가지 주제로 Node.js를 통해 구현해볼 예정입니다.
  - ✓ **실습 1 – 최신 JS 사용해보기**: Modern Javascript 사용해보기
  - ✓ **실습 2 – Node.js로 비동기 구현**: Node.js를 활용해 Javascript로 비동기를 구현해보기

- (오늘의 과제는 실습 2의 결과물을 활용하시면 쉽게 하실 수 있으니 잘 따라와주세요!)

실습 1 – 최신 JS 사용해보기

- 실습에 앞서 yarn을 이용해 이번주에 사용할 프로젝트를 생성해봅시다. nodemon도 설치해봅시다!
  - ✓ yarn init
  - ✓ yarn add nodemon --dev

- 그러고 나면! 생성된 package.json 파일을 지난 주와 같이 오른쪽 코드로 수정해봅시다.

- 그러고 나면! index.js 파일을 생성하고 아래의 명령어를 입력해봅시다.
  - ✓ yarn start:dev

```
{
  "name": "week2",
  "version": "1.0.0",
  "description": "KWEB 준스 2주차 실습",
  "main": "index.js",
  "author": "배민근",
  "license": "MIT",
  "devDependencies": {
    "nodemon": "^1.18.4"
  },
  "scripts": {
    "start" : "node index.js",
    "start:dev": "nodemon index.js"
  }
}
```

실습 1 – 최신 JS 사용해보기

- 실습 1은 예제 코드가 따로 없습니다! 대신 앞의 모든 새로운 Javascript 예제들을 한번씩 사용해보세요.
  - ✓ 익숙해지셔야 실습 2부터 앞으로의 실습까지 쭉 수월하실겁니다.

- 앞으로 이런 문법들 설명없이 사용할 예정이니 여러분들이 찾아가며 공부해보세요!

- 시간은 10-15분 정도 드리겠습니다! (이론 빡셌으니 잠깐 쉬어갑시다~)

실습 2 – Node.js로 비동기 구현

- 이제 좀 Javascript의 최신 사용법에 대해 입문 정도는 하셨나요? (실습 1에서 농땡이만 치신건 크흠..)
- 문법이야 크게 걱정하실 필요 없고 그냥 많이 쓰면 익숙해집니다. 역시 컴과는 삽질이죠! ㅋㅋㅋ
- 실습 2에선 앞에서 배운 Callback Function, Promise, async / await를 이용해 3가지 방법으로 Node.js 에서 비동기 방식으로 파일을 읽어오는 코드를 작성할 예정입니다.
- Callback, Promise, async /await 순으로 진행될 예정이며, 이번 주 과제랑 연관 깊은 실습이니 열심히 따라오세요~

실습 2 – Node.js로 비동기 구현

- 실습 2는 Callback으로 먼저 시작할게요~

- 새로운 명령어인 git clone을 이용해 만들어놓은 파일을 가져와봅시다.
  - ✓ git clone https://github.com/baemingun/kweb_week2_practice.git

```
USER@MG-PEN MINGW64 ~/Desktop
$ git clone https://github.com/baemingun/kweb_week2_practice.git
Cloning into 'kweb_week2_practice'...
remote: Enumerating objects: 11, done.
remote: Counting objects: 100% (11/11), done.
remote: Compressing objects: 100% (9/9), done.
remote: Total 11 (delta 1), reused 11 (delta 1), pack-reused 0
Unpacking objects: 100% (11/11), done.
```

- 이런 식으로 GitHub에 공개된 Repository를 Local 환경으로 Clone해올 수도 있습니다~
  - ✓ yarn install 잊지 마세요~

실습 2 – Node.js로 비동기 구현

- git clone으로 가져온 코드를 오른쪽과 같이 수정해봅시다.
  - ✓ 코드만 읽어보면 당연히 txt 파일 하나가 존재해야 할텐데 모르는 사람 없겠죠? 생성해주세요!
  - ✓ **중요!** txt 파일 인코딩 UTF-8로 저장해주세요.

- 그러면 nodemon 이 실행되어 있다면 코드가 바뀌면 바로 반영되어 실행됩니다. 확인해보세요!
  - ✓ 정상적으로 구현되었다면 txt 파일의 내용이 브라우저에 표시될 것입니다~

- (사실 server.listen도 이미 Callback 함수로 구현되어 있었는데 이제 뭔말인지 아시겠죠?)

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

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
    fs.readFile('./practice.txt','UTF-8', function(err,data) {
        if (err) {
            console.log(err);
            res.statusCode = 500;
            res.setHeader('Content-Type', 'text/plain; charset=uft-8');
            res.end("Server Error");
        } else {
            res.statusCode = 200;
            res.setHeader('Content-Type', 'text/plain; charset=uft-8');
            res.end(data);
        }
    });
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

다시 잠깐, Clojure

- 아까 Clojure 실습에서 보여준다고 했죠? 앞선 코드를 다시 한 번 살펴보면 뭔가 이상합니다.
- Callback의 실행은 fs 모듈안일텐데 그 속에는 createServer의 콜백함수의 res 변수가 정의 되어 있지 않는데 어떻게 잘 실행되는지..?
- 어떻게 readFile 함수의 콜백이 res 변수를 사용할 수 있는지..?
  - ✓ 이 의문 사항을 바로 Clojure가 해결해줍니다.
- **참고!** 함수에 클로저 정보가 저장되는 것을 바인딩이라고 한다.

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

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
    fs.readFile('./practice.txt','UTF-8', function(err,data) {
        if (err) {
            console.log(err);
            res.statusCode = 500;
            res.setHeader('Content-Type', 'text/plain; charset=uft-8');
            res.end("Server Error");
        } else {
            res.statusCode = 200;
            res.setHeader('Content-Type', 'text/plain; charset=uft-8');
            res.end(data);
        }
    });
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

실습 2 – Node.js로 비동기 구현

- 이번엔 똑같은 코드를 Promise로 구현해봅시다.

- 오른쪽에 벌써 코드를 마련해두었습니다. 코드를 수정하고 실행이 정상적으로 되는지 확인 ㄱㄱ
  - ✓ 함수로 따로 뺀 이유는 코드가 더 깔끔해보이니까요!
  - ✓ 그냥 new Promise로 하셔도 됩니다~

- 딱히 크게 설명드릴 건 없습니다. 올바르게 실행되면 resolve가 실행, 못 읽으면 reject가 실행되는 코드입니다.

- 이 Promise 예제로는 부족하니 앞으로 Promise를 따로 조금씩 더 공부하는 것을 권장드립니다!

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

const hostname = '127.0.0.1';
const port = 3000;

function readFile(fileName,type) {
    return new Promise((resolve, reject) => {
        fs.readFile(fileName,type, (err,data) => {
            err ? reject(err) : resolve(data);
        });
    });
}

const server = http.createServer((req, res) => {
    readFile('./practice.txt','UTF-8')
        .then((data) => {
            res.statusCode = 200;
            res.setHeader('Content-Type', 'text/plain; charset=uft-8');
            res.end(data);
        })
        .catch((err) => {
            console.log(err);
            res.statusCode = 500;
            res.setHeader('Content-Type', 'text/plain; charset=uft-8');
            res.end("Server Error");
        });
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

실습 2 – Node.js로 비동기 구현

- 이번엔 똑같은 코드를 ES 7부터 구현된 가장 최신 기법인 async/await로 구현해봅시다!

- 역시나 코드는 오른쪽에 준비해두었습니다.
  - ✓ 제일 깔끔해보이지 않나요? ㅋㅋㅋㅋ
  - ✓ (가까운 미래엔 아마 이 방식만 사용될 수도..?)

- util.promisity(fs.readFile)이 뭔지? → 해당 함수를 Promise를 반환하게 변환해줍니다.
  - ✓ async 함수는 암묵적으로 Promise를 반환합니다.

- try...catch문은 왜 쓰는지 → 앞에 슬라이드 참고!

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

const hostname = '127.0.0.1';
const port = 3000;

// Convert fs.readFile into Promise version of same
const fs_readFile = util.promisify(fs.readFile);

const server = http.createServer(async (req, res) => {
    try {
        const data = await fs_readFile('./practice.txt','UTF-8');
        res.statusCode = 200;
        res.setHeader('Content-Type', 'text/plain; charset=uft-8');
        res.end(data);
    } catch (err) {
        console.log(err);
        res.statusCode = 500;
        res.setHeader('Content-Type', 'text/plain; charset=uft-8');
        res.end("Server Error");
    }
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

과제 (~ 10/14 23:59:59)

- 과제기한: 2018년 10월 14일 일요일 자정까지

- 2주차 과제는 Callback으로 작성된 코드를 Promise와 async / await 방식으로 수정하는 것입니다. 이 코드는 현재 GitHub에 업로드해 두었습니다.
  - ✓ 이번에도 기한이 제법 길죠? 공부하면서 해보라고 약간 길게 냈습니다!

- 이번 주의 과제는 1개지만 아래의 과정을 따라 Step by Step으로 진행하시길 바래요!
  - ✓ **단계 1 – GitHub Repository 만들기 및 Initial commit**: git clone 후, 본인의 저장소에 commit 하기
  - ✓ **단계 2 – Promise 방식 구현 및 업로드**: 주어진 코드를 Promise 방식으로 수정한 후, commit 하기
  - ✓ **단계 3 – async / await 방식 구현 및 업로드**: 주어진 코드를 async / await 방식으로 수정한 후, commit 하기
  - ✓ **단계 4 – issue 생성**: 여러분들 코드를 피드백받을 issue를 하나 생성해봅시다.

- 제출방법: 위 단계들을 따라 완료한 후, GitHub Repository 주소를 준회원 톡방에 올리시면 됩니다.

과제 명세

- 최종적으론 코드 구현 후, 이슈까지 열어서 준회원 톡방에 여러분의 Repository 주소를 올리시면 됩니다.
  - ✓ **여러분의 GitHub Repository에는 적어도 3~4개 이상의 commit 로그가 찍혀 있어야합니다.**
  - ✓ (과제 전용 Repository 하나 만드셔서 앞으로 계속 그 저장소에만 commit해서 제출하셔도 좋고 매번 새로 만드셔도 됩니다.)

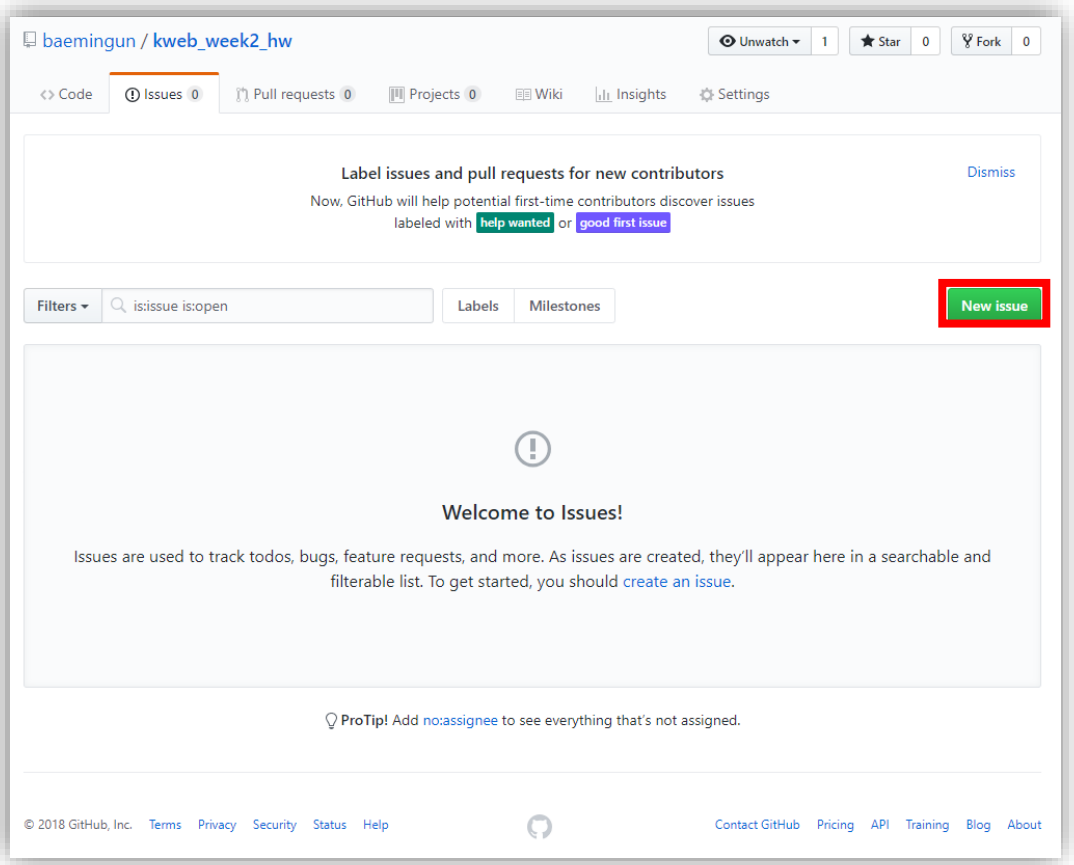
- **단계 1** - 아래 저장소를 git clone 명령어로 clone 해온 뒤, 1주차를 참고해 여러분들의 저장소를 만든 후 clone 받은 파일들을 commit (push까지!) 해주시면 됩니다.
  - ✓ https://github.com/baemingun/kweb_week2_hw.git

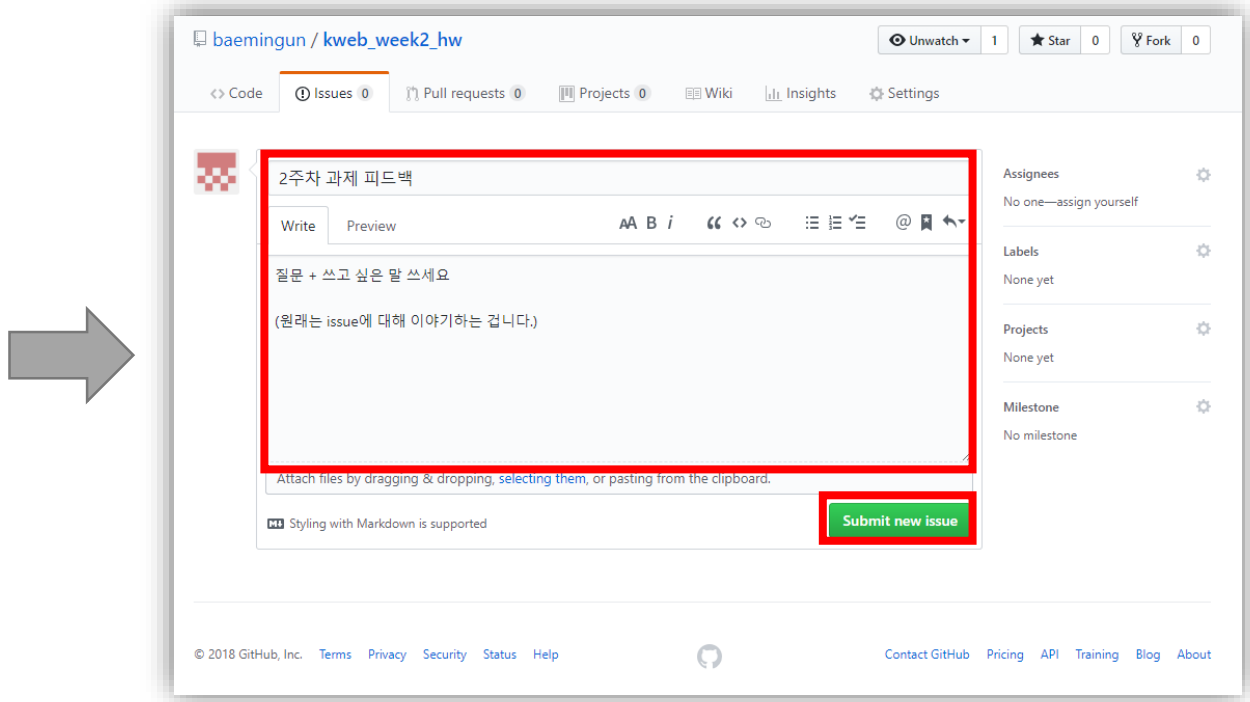
- **단계 2** – 단계 1에서 주어진 파일을 Promise 방식으로 완전히 똑같이 동작하는 코드를 구현한 후, 단계 1에서 만든 여러분의 저장소에 commit (push까지!) 해주시면 됩니다.

- **단계 3** – 단계 1에서 주어진 파일을 async / await 방식으로 완전히 똑같이 동작하는 코드를 구현한 후, 단계 1에서 만든 여러분의 저장소에 commit (push까지!) 해주시면 됩니다.

과제 명세

- **단계 4** – 아래 그림을 참고해 여러분의 저장소에 피드백을 받을 수 있도록 isssue를 열어주시면 됩니다.





That’s all for today!