1. ES 6 개요

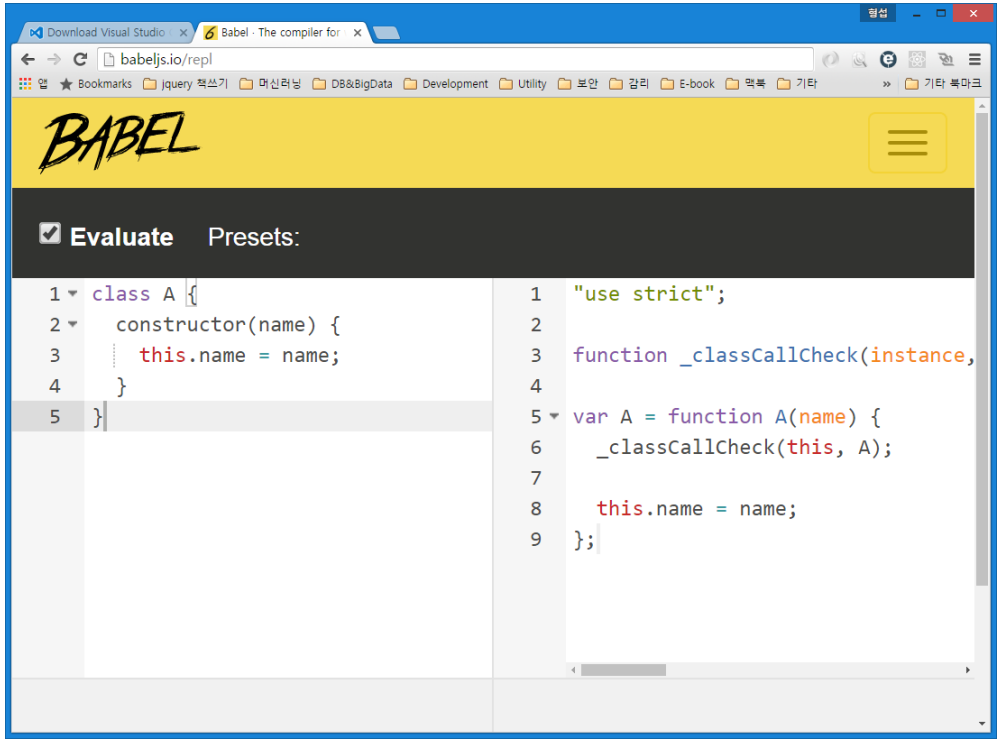
ES6 개요

- 새로운 문법이 추가되었음
- ES6는 ES5에 대해 하위 호환성을 가짐
- 현재 ES6를 완벽하게 지원하는 환경은 없음
  - 크롬 : 98%, Edge : 98%, IE11 : 11%, FF: 98%
  - Node.js 12 : 98%
  - http://kangax.github.io/compat-table/es6/
- 대표적인 트랜스 파일러
  - Babel : 72%
  - Typescript : 70%
  - 최근에 TypeScript 비중이 증가하고 있음.
    - 정적 타입을 사용하는 C#, Java 개발자들은 TypeScript 선호도가 높음

2. babel(1)

babel 개요

- ES6 코드를 ES5 코드로 트랜스파일함.
- 온라인 도구(http://babeljs.io/repl)



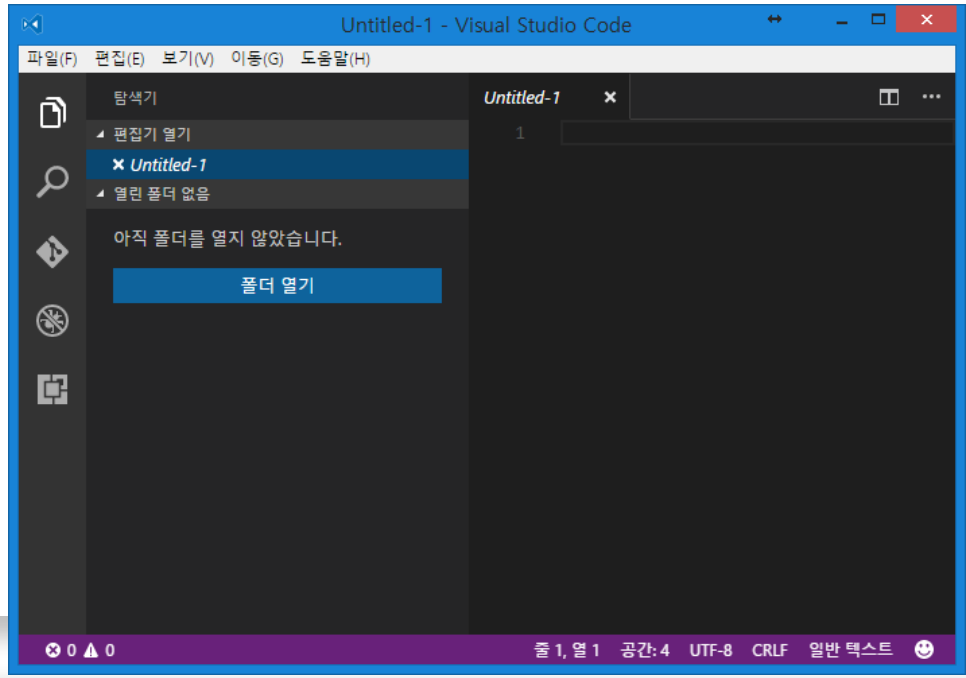
2. babel(2)

ES6 테스트를 위한 프로젝트 생성

- 프로젝트를 위한 디렉토리 생성
- mkdir testapp

Visual Studio Code 실행

- 실행 후 통합 생성한 폴더 열기
  - 파일 메뉴 - 폴더 열기
- 이 예제에서는 babel 6 사용



2. babel(3)

프로젝트 초기화

- 메인메뉴에서 '보기' - '터미널' 실행
- npm init 실행
  - 기본 값으로 입력하거나(엔터키를 계속해서 입력) 적절한 값을 입력함.
  - 아래는 입력한 사례

```
터미널                                                    1: cmd.exe

keywords:
license: (MIT)
About to write to c:\_dev\workspace\testapp\package.json:

{
  "name": "testapp",
  "version": "1.0.0",
  "description": "testapp 입니다.",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "stephen",
  "license": "MIT"
}


Is this ok? (yes) yes
```

2. babel(4)

babel을 사용하기 위한 설정

- npm을 이용한 기본 설정
  - npm install --save-dev babel-cli babel-core babel-preset-env babel-preset-stage-2
  - --global 옵션은 전역, --save-dev 옵션은 개발 의존성으로 설치
- npm 명령어 수행 후 package.json 확인

```
{
  "name": "ch02",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "babel-cli": "^6.26.0",
    "babel-core": "^6.26.3",
    "babel-preset-env": "^1.7.0",
    "babel-preset-stage-2": "^6.24.1"
  }
}
```

2. babel(5)

.babelrc 파일 작성

- .babelrc 파일은 babel 실행을 위한 기본 설정 파일
- Visual Studio Code에서 .babelrc 파일 추가후 다음과 같이 작성

```
{
    "presets" : [ "env", "stage-2" ]
}
```

테스트 코드 작성

- src 폴더 생성 후 sample.js 파일 추가

```
let name = "world";
console.log(`hello ${name}!!`);
```

- 작성후 통합 터미널에서 npx babel src -d build 명령어 실행
- build 디렉토리의 sample.js 파일 확인

3. source map(1)

source map?

- 트랜스파일된 코드로 실행하지만 디버깅은 원본 코드로……
- source map 스펙은 이미 coffeescript 등에서도 널리 사용하던 것임
  - http://www.html5rocks.com/en/tutorials/developertools/sourcemaps/?redirect_from_locale=ko
  - https://docs.google.com/document/d/1U1RGAehQwRypUTovF1KRlpiOFze0b-_2gc6fAH0KY0k/edit?hl=en_US&pli=1&pli=1

babel을 이용해 트랜스파일 할 때 source map 생성하기

- npm install babel-plugin-source-map-support --save-dev
- npx babel src/test1.js -o build/test1.js --source-maps
- npx babel src -d build --source-maps

3. source map(2)

source map 테스트

- src/sample.js

```
let name = "world";
console.log(`hello ${name}!!`);
```

- npx babel src -d build --source-maps
- build/sample.js

```
"use strict";
var name = "world";
console.log("hello " + name + "!!");
//# sourceMappingURL=sample.js.map
```

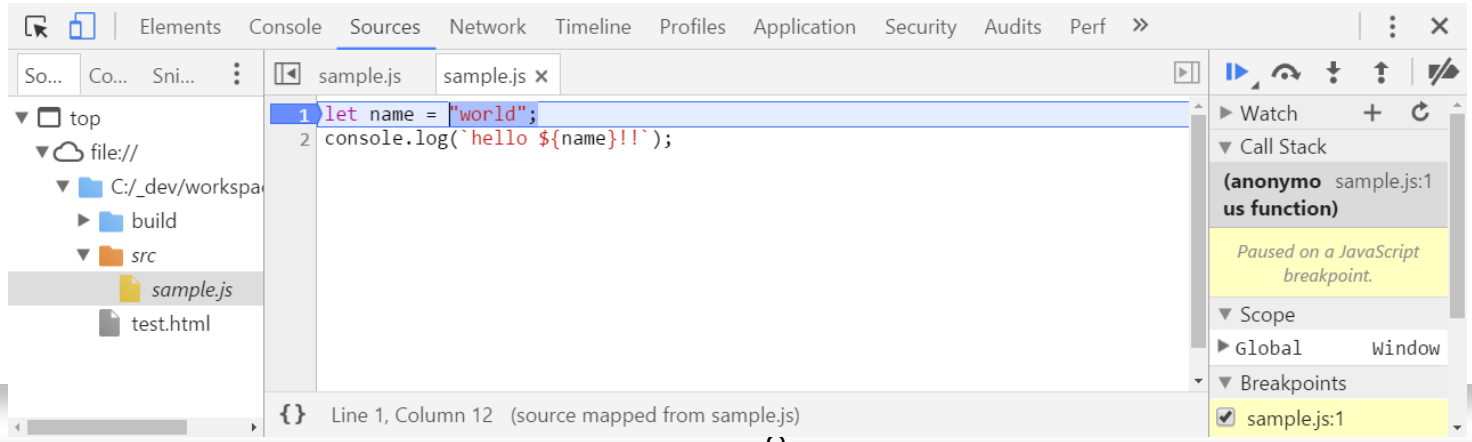
- build/sample.js.map
  - 아래 참조

3. source map(4)

크롬 브라우저에서 디버깅하기(이어서)

- build/sample.js를 참조하는 페이지 작성 & 실행(test.html)

```
<html>
    <head>
        <title>test!!</title>
        <script type="text/javascript" src="build/sample.js"></script>
    </head>
    <body>
    </body>
</html>
```



4. ES6 상세

- ES6에서 추가된 내용 다룸
- Babel을 통해서 사용하게 될 문법 중심

4.1 let, const(1)

var

- hoisting : 개발자들에게 이해하기 어려운 부분
- 함수단위 scope만 제공함
- var 중복 선언을 허용함으로써 혼란 야기

let

- var와 선언하는 방법은 유사하지만…
- 중복 선언을 허용하지 않음

```
> let a = 100;
< undefined
> let a = "hello";
✖ ▶Uncaught TypeError: Identifier 'a' has already been declared(…)   VM78:1
```

4.1 let, const(2)

let (이어서)

- block scope를 지원함.

ES6

```
let msg= "GLOBAL";
function outer(a) {
    let msg = "OUTER";
    console.log(msg);
    if (true) {
        let msg = "BLOCK";
        console.log(msg);
    }
}
```

ES5

```
"use strict";
var msg = "GLOBAL";
function outer(a) {
   var msg = "OUTER";
   console.log(msg);
   if (true) {
       var _msg = "BLOCK";
       console.log(_msg);
   }
}
```

- 대부분의 var는 let으로 대체가 가능함.

4.1 let, const(3)

const

- 상수
- 값이 한번 초기화되면 변경이 불가능하다.
- block scope를 지원함.

기존의 var는?

- hoisting!! - 변수를 미리 생성!
- block scope 지원하지 않음

```
//에러 안남
console.log(A1);
var A1 = "hello";
```

```
var msg = "hello";
function test() {
    console.log(msg);
    if (false) {
        var msg = "world";
    }
    console.log(msg);
}
test();
```

```
undefined
undefined
```

4.2 Default Parameter

파라미터 값을 전달하지 않았을 때의 기본값을 정의

```
function addContact(name, mobile,
                    home="없음",
                    address="없음",
                    email="없음") {
    var str = `name=${name}, mobile=${mobile}, home=${home},
              address=${address}, email=${email}`;
    console.log(str);
}

addContact("홍길동", "010-222-3331")
addContact("이몽룡", "010-222-3331", "02-3422-9900", "서울시");
```

```
name=홍길동, mobile=010-222-3331, home=없음, address=없음, email=없음
name=이몽룡, mobile=010-222-3331, home=02-3422-9900, address=서울시, email=없음
```

4.3 Rest Operator

가변 파라미터

- 마지막에 배치해야 함.
- Rest Operator를 지원하지 전에는 arguments를 이용해 가변인자를 처리하였음 → 더이상 arguments를 이용하지 않아도 됨.

```
function foodReport(name, age, ...favoriteFoods) {
    console.log(name + ", " + age);
    console.log(favoriteFoods);
}

foodReport("이몽룡", 20, "짜장면", "냉면", "불고기");
foodReport("홍길동", 16, "초밥");
```

```
이몽룡, 20
[ '짜장면', '냉면', '불고기' ]
홍길동, 16
[ '초밥' ]
```

4.4 Destructuring(1)

구조 분해 할당

- 배열, 객체의 값들을 여러 변수에 추출하여 할당할 수 있도록 하는 새로운 표현식

```
let arr = [10,20,30,40];
let [a,b,c] = arr;
console.log(a, b, c);
```

```
10 20 30
```

```
let p1 = {name:"홍길동", age:20, gender:"M"};
let { name:n, age:a, gender } = p1;
console.log(n,a,gender);
```

```
홍길동 20 M
```

4.4 Destructuring(2)

구조 분해 할당(이어서)

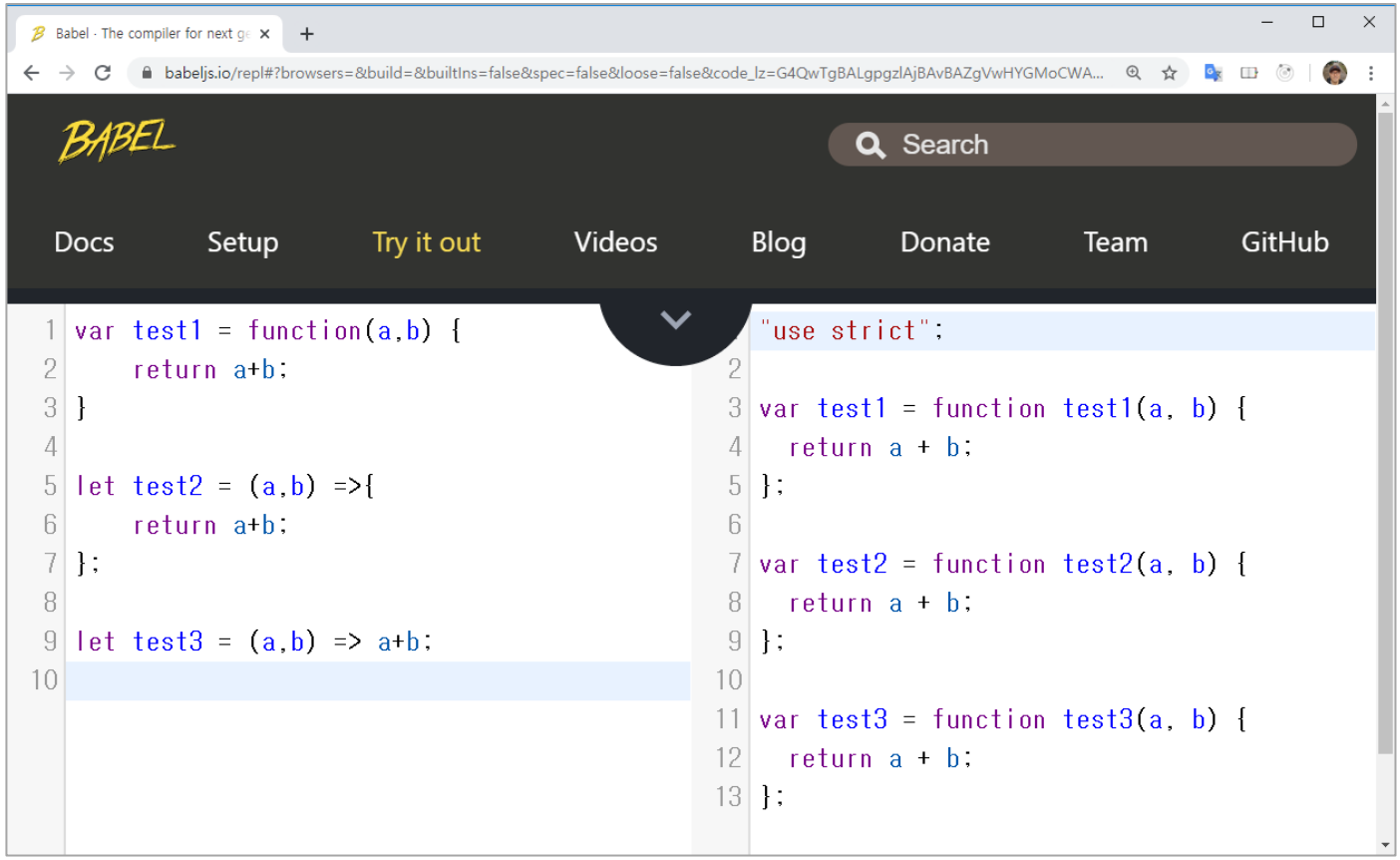
```
function addContact({name, phone, email="이메일 없음", age=0}) {
    console.log("이름 : " + name);
    console.log("전번 : " + phone);
    console.log("이메일 : " + email);
    console.log("나이 : " + age);
}

addContact({
    name : "이몽룡",
    phone : "010-3434-8989"
})
```

```
이름 : 이몽룡
전번 : 010-3434-8989
이메일 : 이메일 없음
나이 : 0
```

4.5 Arrow Function Expression(1)

화살표 함수 표현식

- 핵심적인 차이는 this와 관련되어 있음



4.5 Arrow Function Expression(2)

JS에서의 this

- 현재 호출중인 메서드를 보유한 객체를 가리킴 (default)

```
var obj = { result: 0 };
obj.add = function(x,y) {
   this.result = x+y;
}
//아래 코드에서의 this는?     obj임
obj.add(3,4)
console.log(obj)
```

- 위코드를 다음과 같이 실행하면?

```
var add2 = obj.add();
//호출될 때 add2() 메서드를 보유한 객체가 없으므로 Global(전역)객체가 this가 됨.
add2()
```

- this가 바인딩되는 시점?
  - 메서드를 호출할 때마다 this가 바인딩됨.
    - 또한 메서드를 호출할 때 직접 this를 지정할 수 있음(apply, call 메서드)
    - 또한 this가 미리 바인딩된 새로운 함수를 리턴할 수 있음(bind)

4.5 Arrow Function Expression(3)

- apply(), call() 메서드

```
var add = function(x,y) {
   this.result = x+y;
}
var obj = {};
//add 함수에 obj를 직접 this로 지정하여 호출함
add.apply(obj, [4,5])
//add.call(obj, 3,4)
```

- bind() 메서드

```
var add = function(x,y) {
   this.result = x+y;
}
var obj = {};
//add 함수에 obj를 직접 this로 연결한 새로운 함수를 리턴함.
var add2 = add.bind(obj);
```

- 메서드를 어느 객체의 메서드 형태로 호출하느냐에 따라 this가 연결됨. --> Lexical Binding

4.5 Arrow Function Expression(4)

- 전통적인 함수가 중첩되었을 때의 문제점 이해

```
var obj = { result:0 };
obj.add = function(x,y) {
  console.log(this);
  function inner() {
     this.result = x+y;
  }
  inner();
}
obj.add(4,5)
```

  - add() 메서드 내부에 inner 함수가 정의되어 있음
  - 바깥쪽 함수 바로 안쪽 영역의 this? --> obj를 참조함.
  - inner() 함수 내부의 this가 obj를 참조할 것인가?
    - 그렇지 않음. inner() 와 같이 호출했기 때문에 inner() 내부의 this는 전역객체를 참조함. 즉 전역변수 result에 덧셈한 결과가 저장될 것임.
  - 이 문제를 해결하려면?
    - apply(), call(), bind()를 이용하거나
    - 화살표 함수를 이용한다.

4.5 Arrow Function Expression(5)

- 문제 해결1 : bind()

```
var obj = { result:0 };
obj.add = function(x,y) {
  function inner() {
    this.result = x+y;
  }
  inner = inner.bind(this);
  inner();
}
obj.add(4,5)
```

- 문제 해결2 : apply()

```
var obj = { result:0 };
obj.add = function(x,y) {
  function inner() {
    this.result = x+y;
  }
  inner.apply(this);
}
obj.add(4,5)
```

- 문제 해결3 : 화살표 함수

```
var obj = { result:0 };
obj.add = function(x,y) {
  var inner = () => {
    this.result = x+y;
  }
  inner()
}
obj.add(4,5)
```

- 화살표 함수는 lexical binding이 아님
- 함수가 중첩되었을 때 바깥쪽 함수의 this가 안쪽 함수로 지정됨.
- React 클래스 컴포넌트 작성할 때 알고 있어야 하는 개념

4.6 Object Literal(1)

새로운 객체 리터럴

- 객체 속성 표기

```
var name = "홍길동";
var age = 20;
var email = "gdhong@test.com";
var obj = { name, age, email };

console.log(obj);
```

- 속성명과 변수명이 같은 경우는 생략 가능

```
var obj = { name: name, age: age, email: email };
```

4.6 Object Literal(2)

새로운 객체 리터럴(이어서)

- 새로운 메서드 표기법

```
let p1 = {
    name : "아이패드",
    price : 200000,
    quantity : 2,
    order : function() {
        if (!this.amount) {
            this.amount = this.quantity * this.price;
        }
        console.log("주문금액 : " + this.amount);
    },
    discount(rate) {
        if (rate > 0 && rate < 0.8) {
            this.amount = (1-rate) * this.price * this.quantity;
        }
        console.log((100*rate) + "% 할인된 금액으로 구매합니다.");
    }
}
p1.discount(0.2);
p1.order();
```

4.7 Template Literal(1)

backtit(`)으로 묶여진 문자열

- 템플릿 대입문(${}) 로 문자열 끼워넣기 기능 제공
  - 템플릿 대입문에 수식 구문, 변수, 함수 호출 구문 등 모든 표현식이 올 수 있음.
  - 템플릿 문자열을 다른 템플릿 문자열 안에 배치하는 것도 가능
  - ${ 을 나타내려면 $ 또는 {을 이스케이프시킴

```
var d1 = new Date();
var name = "홍길동";
var r1 = `${name} 님에게 ${d1.toDateString() }에 연락했다.`;
```

- 여러줄도 표현가능

```
var product = "갤럭시S7";
var price = 199000;
var str = `${name}의 가격은
    ${price}원 입니다.`;
console.log(str);
```

4.7 Template Literal(2)

Tagged Template Literal

```
var getPercent = function(str, ...values) {
      //str : [ '첫번째 값은 ', '이고, 두번째 값은 ', '이다.' ]
      //values : [ 0.222, 0.78999 ]
}

var v1 = 0.222;
var v2 = 0.78999;
var r2 = getPercent`첫번째 값은 ${v1}이고, 두번째 값은 ${v2}이다.`;
```

- tagged template 함수 뒤에 template literal이 따라오면…
- tagged template 함수
  - 첫번째 인자 : 대입 문자열이 아닌 나머지 문자열들의 배열
  - 두번째 이후 인자 : 대입 문자열에 할당될 값들..

4.8 Class(1)

ES5

- 유사 클래스 : 함수를 이용해 클래스 기능을 만들어냄
- 작성이 힘듬
  - 상속 : Prototype으로 구현
  - 캡슐화 : Closure로 구현

ES6

- class 키워드 사용

```
class Polygon {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
}
```

- 함수는 호이스팅(Hoisting)하지만 class는 그렇지 않음

4.8 Class(2)

class 예

```
class Person {
    constructor(name, tel, address) {
        this.name = name;
        this.tel = tel;
        this.address = address;
        if (Person.count) { Person.count++; } else { Person.count = 1; }
    }
    static getPersonCount() {
        return Person.count;
    }
    toString() {
        return `name=${this.name}, tel=${this.tel}, address=${this.address}`;
    }
}
var p1 = new Person('홍길동', '010-222-3331', '서울시');
var p1 = new Person('이몽룡', '010-222-3332', '경기도');
console.log(p1.toString());
console.log(Person.getPersonCount());
```

```
name=이몽룡, tel=010-222-3332, address=경기도
2
```

4.8 Class(3)

상속

```
class Person {
    ……
}

class Employees extends Person {
    constructor(name, tel, address, empno, dept) {
        super(name,tel,address);
        this.empno = empno;
        this.dept = dept;
    }
    toString() {
        return super.toString() + `, empno=${this.empno}, dept=${this.dept}`;
    }
    getEmpInfo() {
        return `${this.empno} : ${this.name}은 ${this.dept} 부서입니다.`;
    }
}

let e1 = new Employees("이몽룡", "010-222-2121", "서울시", "A12311", "회계팀");
console.log(e1.getEmpInfo());
console.log(e1.toString());
console.log(Person.getPersonCount());
```

4.8 Class(4)

ES5 와 ES6 비교

- Function
  - ES5 : function
  - ES6 : function 또는 Arrow Function
- Class
  - ES5 : constructor function
  - ES6 : class 키워드 사용
- Method
  - ES5 : function을 prototyp에 작성
  - ES6 : class 내부에 method로 작성
- Constructor
  - ES5 : constructor function
  - ES6 : class 내에 constructor 작성

4.9 Promise(1)

비동기 처리를 위한 콜백 처리

- Callback Hell : 콜백함수들이 중첩되어 지옥을 경험함
  - 디버깅 어려움.
  - 예외처리 어려움

```
doSomething(param1, param2, function(err, paramx){
    doMore(paramx, function(err, result){
        insertRow(result, function(err){
            yetAnotherOperation(someparameter, function(s){
                somethingElse(function(x){
                    ……
                });
            });
        });
    });
});
```

4.9 Promise(2)

Promise 패턴

- 자바스크립트 비동기 처리를 수행하는 추상적인 패턴

```
var p = new Promise(function(resolve, reject) {
    setTimeout(function() {
        var num = Math.round(Math.random()*20);
        var isValid = num % 2;
        if (isValid) { resolve(num); }
        else { reject(num); }
    }, 1000);
});

p.then(function(num) {
    console.log("SUCCESS : " + num);
}).catch(function(num) {
    console.log("FAIL : " + num);
});

console.log("Hello!!");
```

4.9 Promise(3)

Promise Chaining

- then 메서드의 리턴값은 다시 Promise 객체 리턴 가능 → 연속적인 작업 처리시에 유용함
- Promise 객체를 직접 생성하여 리턴할 수도 있음

```
var p = new Promise(
    function(resolve, reject) {
    setTimeout(function() {
        var num = Math.round(Math.random()*20);
        var isValid = num % 2;
        if (isValid) { resolve(num); }
        else { reject(num); }
    }, 1000);
});
```

```
p.then(function(num) {
    console.log("SUCCESS1 : " + num);
    return num*2;
}).then(function(num) {
    console.log("SUCCESS2 : " + num);
    return num*2;
}).then(function(num) {
    console.log("SUCCESS3 : " + num);
})
```

```
SUCCESS1 : 9
SUCCESS2 : 18
SUCCESS3 : 36
```

4.9 Promise(4)

Promise.all()

- 모든 작업이 완료되면 resolve promise를 리턴함.
- 작업 중 하나라도 reject이 되면 reject promise를 리턴함

```
var req1 = new Promise(function(resolve, reject) {
            setTimeout(function() { resolve('작업1'); }, 3000);
});
var req2 = new Promise(function(resolve, reject) {
            setTimeout(function() { resolve('작업2'); }, 1000);
});

Promise.all([req1, req2]).then(function(results) {
            console.log('Then: ', results);
}).catch(function(err) {
            console.log('Catch: ', err);
});
```

4.9 Promise(5)

Promise.race()

- 주어진 promise 들 중에서 하나라도 완료되면 resolve하는 메서드

```
var req1 = new Promise(function(resolve, reject) {
            setTimeout(function() { resolve('작업1'); }, 3000);});
var req2 = new Promise(function(resolve, reject) {
            setTimeout(function() { resolve('작업2'); },1000);});

Promise.race([req1, req2]).then(function(results) {
            console.log('Resolve : ', results);
}).catch(function(err) {
            console.log('Reject : ', err);
});
```

4.10 Module(1)

Module

- 여러 디렉토리와 파일에 나눠서 코드를 작성할 수 있도록 함.
- 자바스크립트 파일은 모듈로써 임포트 될 수 있음

Export

- 모듈안에서 선언된 모든 것은 local(private)
- 모듈 내부의 것들을 public으로 선언하고 다른 모듈에서 이용할 수 있도록 하려면 export 해야 함.
- export 대상 항목
  - let, const, var, function, class
- export let a= 1000;
- export function f1(a) { … }
- export { n1, n2 as othername, … }

4.10 Module(2)

Import

- 다른 모듈로부터 값, 함수, 클래스들을 임포트할 수 있음
- import * as obj from '모듈 경로'
- import { name1, name2 as othername, … } from '모듈 경로'
- import default-name from '모듈 경로'

4.10 Module(3)

Basic Example

module1.js

```
const base = 100
const add  = (x) => base + x
const multiply = (x) => base * x
export { add, multiply };
```

main.js

```
import { add, multiply } from './module1';

console.log(multiply(4));
console.log(add(4));
```

4.10 Module(4)

Default export

- default export를 사용해 단일 값을 익스포트, 임포트 할 수 있음

module1.js

```
const base = 100
const add  = (x) => base + x
const multiply = (x) => base * x
const getBase = ()=>base

export default getBase;
export { add, multiply };
```

main.js

```
import getBase, {add, multiply} from './module1';
console.log(multiply(4));
console.log(add(4));
console.log(getBase());
```

4.11 Spread Operator

일명 전개 연산자

- 객체나 배열을 복제할 때 자주 사용함
  - 기존 객체,배열을 그대로 둔 채 새로운 객체, 배열을 생성함.

```
let arr = [10,20,30];
let arr2 = [ ...arr ];
console.log(arr2);
let arr3 = [ "hello" ,...arr, "world"];
console.log(arr3);

let obj = { a:100, b:200 };
let obj2 = { ...obj };
console.log(obj === obj2); //false
let obj3 = { ...obj, c:300, d:400 };
console.log(obj3); //{a:100, b:200, c:300, d:400 }
```