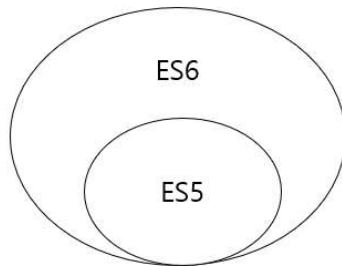
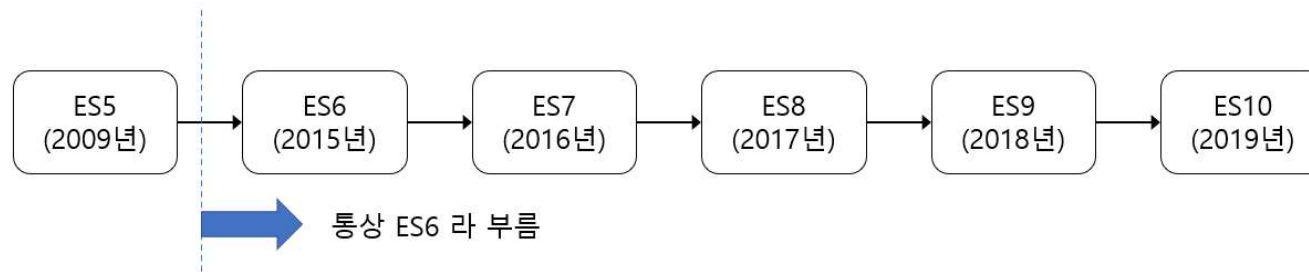


1. ES6 개요(1)

❖ ES6

- ECMAScript 6
- ECMA-262 기술 규격에 정의된 표준화된 스크립트 프로그래밍 언어

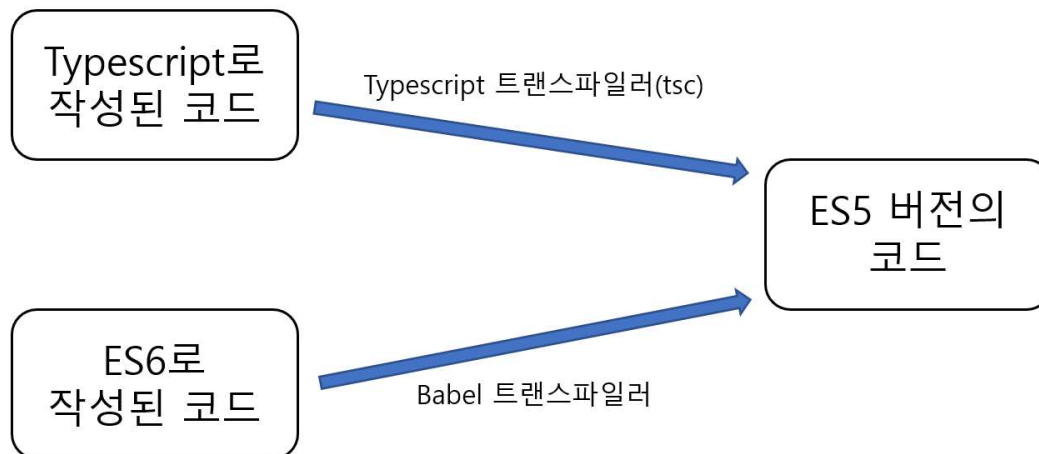


ES6는 이전 버전의 문법을 포함하여 지원하면서
Class, Arrow Function 등 새로운 문법을 추가로 지원함

1. ES6 개요(2)

❖ 트랜스파일러

- Transpile = Translate + Compile
- ES6나 Typescript 언어를 ES5와 같은 이전버전의 자바스크립트 코드로 변환함
- 대표적인 트랜스파일러(Transpiler)
 - Babel
 - tsc



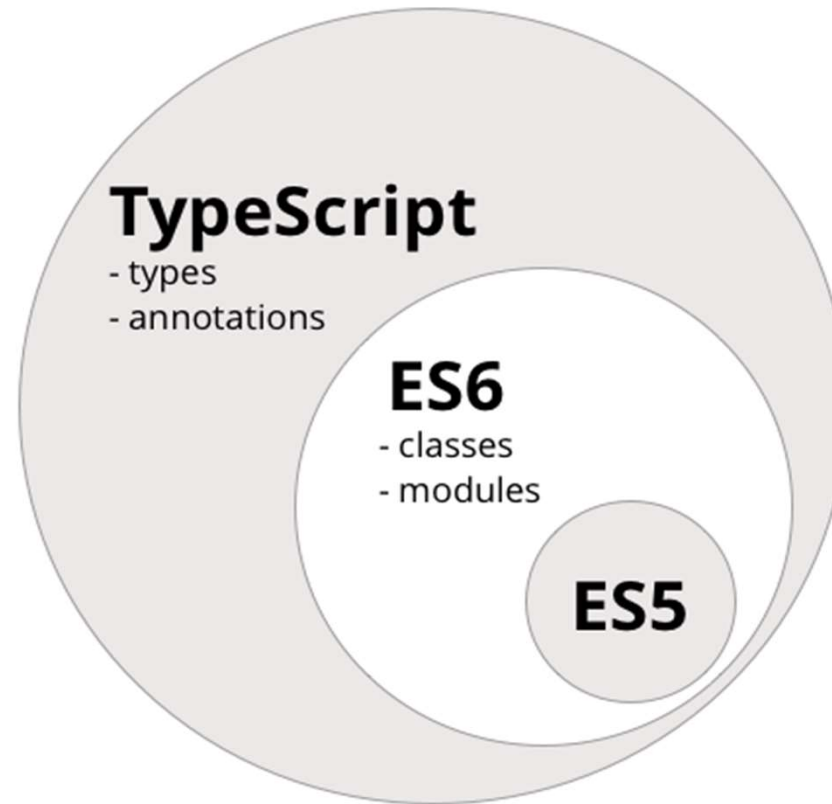
1. ES6 개요(3)

❖ Typescript란?

- ES6에 정적 타입이 추가된 것
- 자바스크립트 언어의 확장버전
 - 기존 ES6 문법을 모두 사용할 수 있음
 - 자바스크립트의 superset
- Microsoft에 의해 관리되고 있음

❖ Typescript의 장점

- 정적 타입 사용
 - 코드의 오류를 줄일 수 있음
 - 쉽고 편리한 디버깅
- IDE와 쉽게 통합됨
- 익숙한 문법
 - java나 C#과 문법이 유사함
- js와 마찬가지로 npm을 사용함



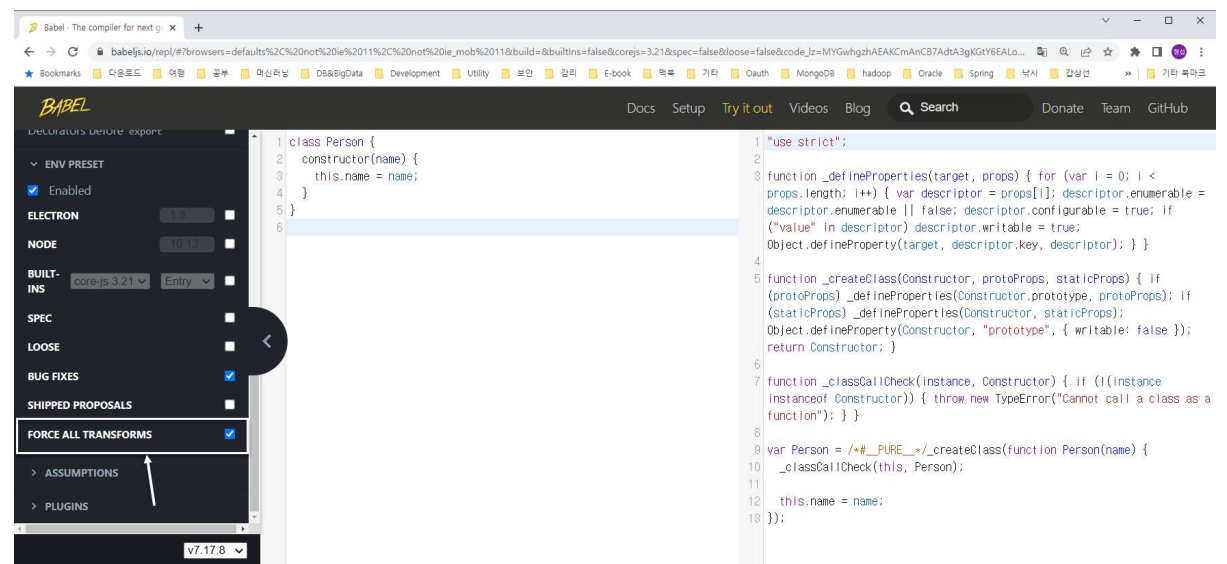
2. ES6 학습

❖ ES6 개요

- Typescript의 Subset 이므로 Typescript는 ES6의 모든 기능을 지원함
- 이 절에서는 ES6 문법중 자주 쓰이는 것 중심으로 살펴봄

❖ Babel 트랜스파일러

- ES6 코드를 이전 버전의 자바스크립트 언어로 번역(변환)해주는 도구
- 사용 방법
 - babel repl 도구 : 브라우저 기반의 도구
 - 직접 설치하여 사용



2.1 ES6를 위한 프로젝트 설정(1)

❖ ES6 테스트를 위한 프로젝트 생성

- 프로젝트를 위한 디렉토리 생성
- `mkdir es6-test`

❖ Visual Studio Code 실행

- 실행 후 통합 생성한 폴더 열기
 - 파일 메뉴 – 폴더 열기

❖ 프로젝트 초기화

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

2.1 ES6를 위한 환경 설정(2)

❖ 프로젝트 초기화

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

```
Press ^C at any time to quit.
package name: (es6-test)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to D:\workspace-temp\react-ts-quickstart\zappendix_es6_typescript\es6-test\package.json:

{
  "name": "es6-test",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

2.1 ES6를 위한 환경 설정(3)

❖ 패키지 설치

- `npm install --save-dev @babel/cli @babel/core @babel/preset-env`
 - `--global` 옵션은 전역, `--save-dev` 옵션은 개발 의존성으로 설치
- npm 명령어 수행 후 `package.json` 확인

```
{
  "name": "es6-test",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  > 디버그
  "scripts": {
    "build": "babel src -d build"
  },
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "@babel/cli": "^7.17.6",
    "@babel/core": "^7.17.5",
    "@babel/preset-env": "^7.16.11"
  }
}
```



2.1 ES6를 위한 환경 설정(4)

❖ babel.config.json 파일 작성

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

```
{  
  "presets": ["@babel/preset-env"]  
}
```

❖ 테스트 코드 작성

- src 폴더 생성 후 02-01.js 파일 추가

```
let name = "john";  
console.log(`Hello ${name}!!`);
```

- 작성후 통합 터미널에서 `npx babel src -d build` 명령어 실행
 - 또는 `npm run build`
- build 디렉토리의 02-01.js 파일 확인

2.2 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
```

2.2 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.

2.2 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();
```

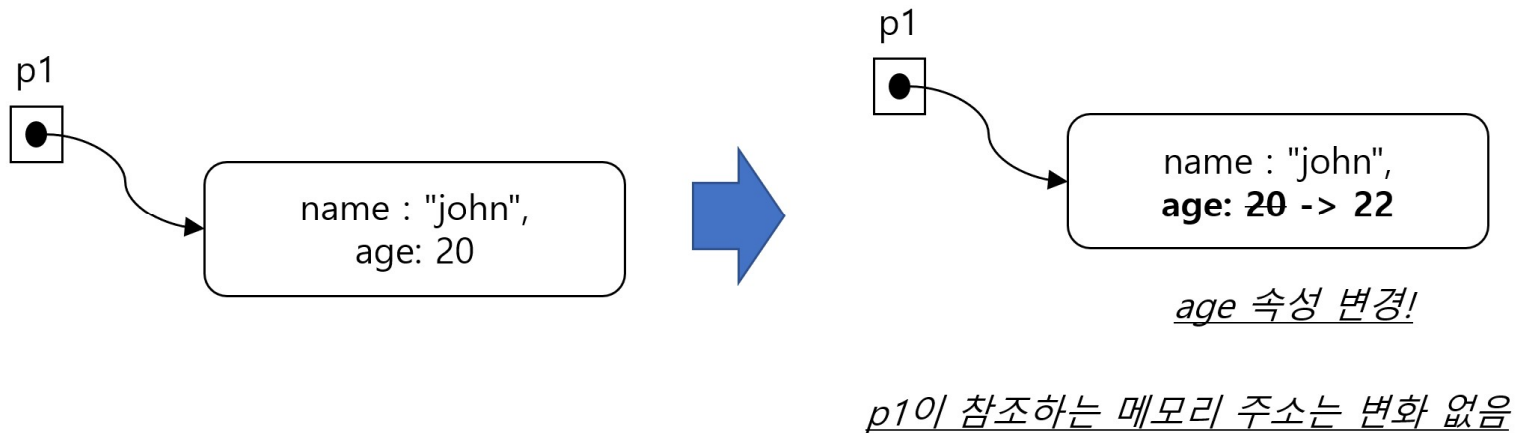


undefined
undefined

2.2 let, const(4)

❖ 예제 02-03

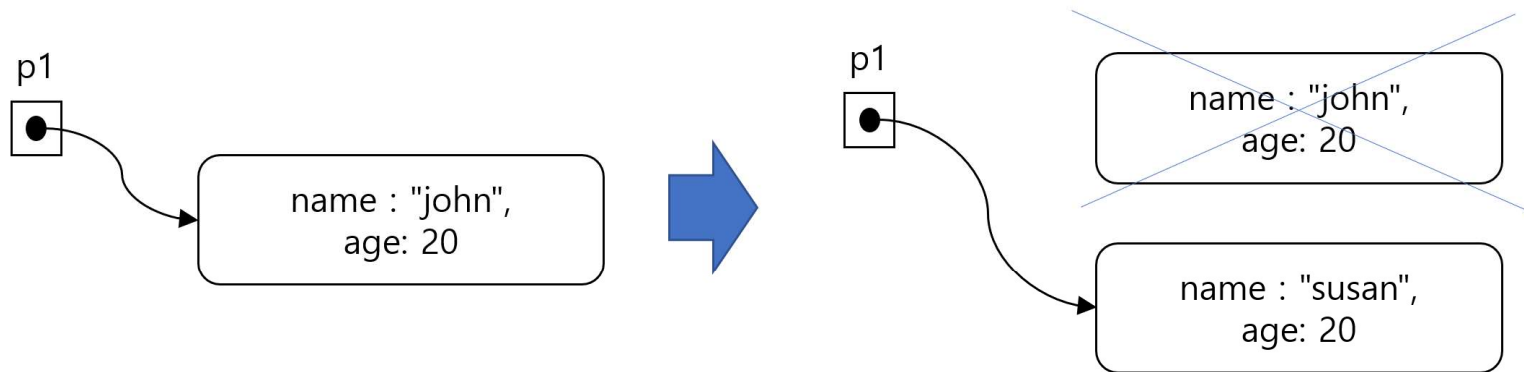
```
const p1 = { name : "john", age : 20 }  
p1.age = 22;  
  
console.log(p1);
```



2.2 let, const(5)

❖ 예제 02-04

```
const p1 = { name : "john", age : 20 }  
p1 = { name:"susan", age: 20 };  
  
console.log(p1);
```



p1이 참조하는 메모리 주소가 바뀌는 것이므로 허용하지 않음.

2.3 기본 파라미터와 가변 파라미터(1)

❖ 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
```

2.3 기본 파라미터와 가변 파라미터(2)

❖ 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']

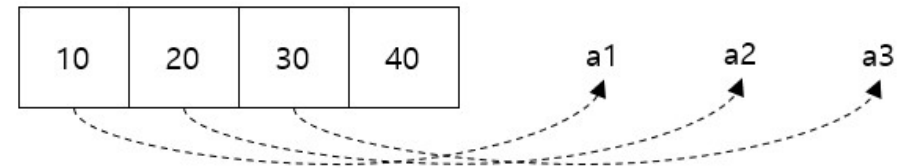
2.4 구조분해 할당(1)

❖ 구조 분해 할당

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

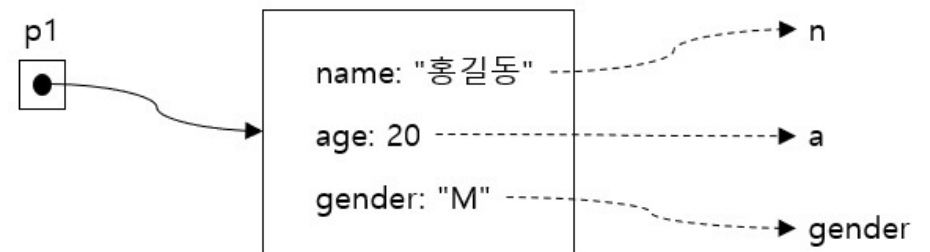
```
let arr = [10,20,30,40];  
let [a,b,c] = arr;  
console.log(a, b, c);
```

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



2.4 구조분해 할당(2)

❖ 구조 분해 할당(이어서)

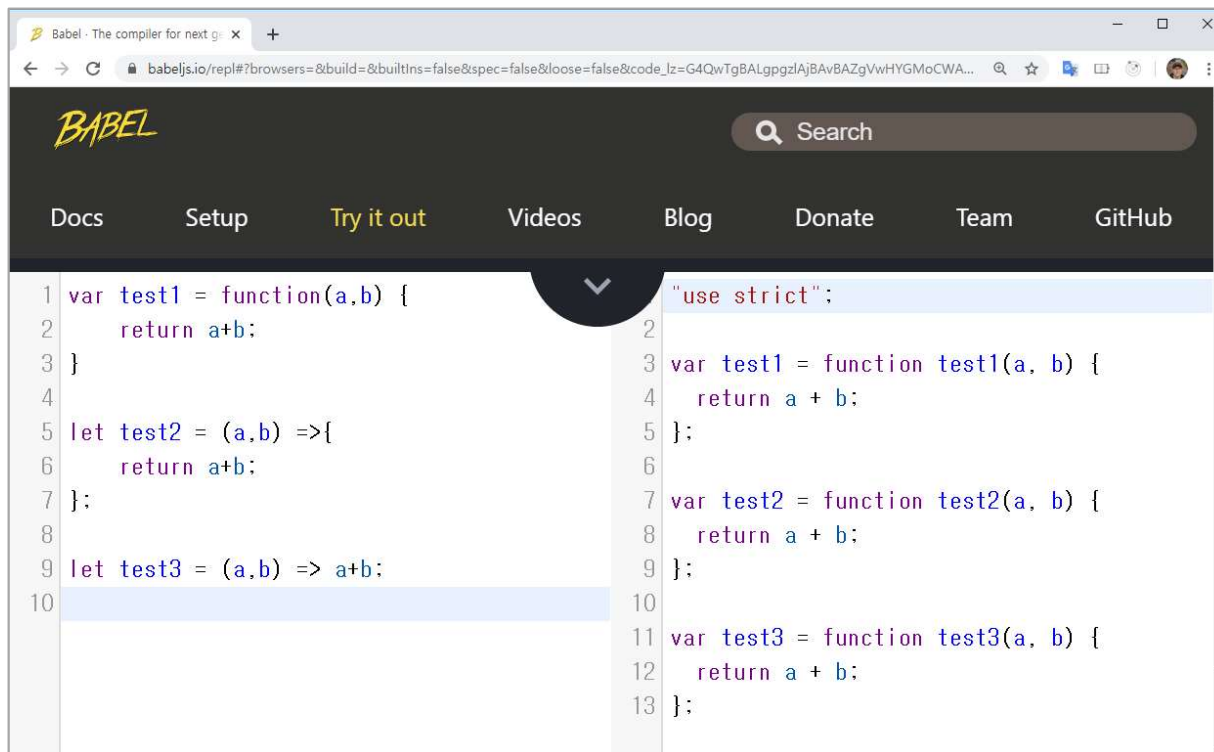
```
function addContact({name, phone, email="이메일 없음", age=0}) {  
  console.log("이름 : " + name);  
  console.log("전번 : " + phone);  
  console.log("이메일 : " + email);  
  console.log("나이 : " + age);  
}  
  
addContact({  
  name : "이몽룡",  
  phone : "010-3434-8989"  
})
```

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

2.5 Arrow Function Expression(1)

❖ 화살표 함수

- 핵심적인 차이는 this와 관련되어 있음



The screenshot shows the Babel REPL interface. On the left, the input code is:

```
1 var test1 = function(a,b) {  
2   return a+b;  
3 }  
4  
5 let test2 = (a,b) =>{  
6   return a+b;  
7 };  
8  
9 let test3 = (a,b) => a+b;  
10
```

A dropdown arrow is visible between the input and output panes. On the right, the output code is:

```
2 "use strict";  
3 var test1 = function test1(a, b) {  
4   return a + b;  
5 };  
6  
7 var test2 = function test2(a, b) {  
8   return a + b;  
9 };  
10  
11 var test3 = function test3(a, b) {  
12   return a + b;  
13 };
```

2.5 Arrow Function Expression(2)

❖ JS에서의 this

- 현재 호출 중인 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)

2.5 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.  
add = add.bind(obj);
```

- 메서드를 어느 객체의 메서드 형태로 호출하느냐에 따라 `this`가 연결됨. --> Lexical Binding

2.5 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.

2.5 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

2.6 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 };
```

2.6 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();
```


2.7 Template Literal(1)

❖ backtick(`)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.toString()} 에 연락했다.`;
```

- 여러줄도 표현가능

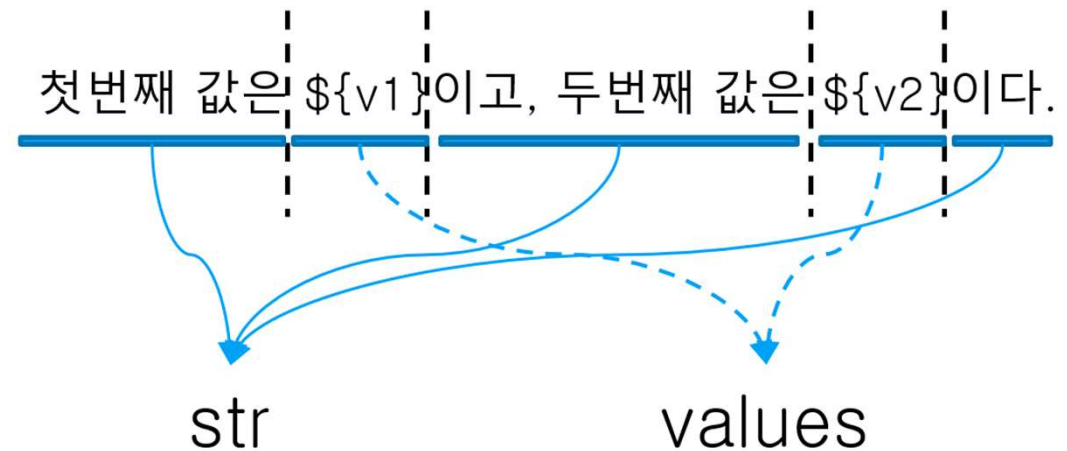
```
var product = "갤럭시S7";  
var price = 199000;  
var str = `${name}의 가격은  
    ${price}원 입니다.`;  
console.log(str);
```

2.7 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..



2.8 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, ... }

2.8 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'`

2.8 Module(3)

❖ Basic Example

src/02-19-module.js

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

src/02-20-main.js

```
import { add, multiply } from './02-19-module';

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

2.8 Module(4)

❖ Default export

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

src/02-19-module.js

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

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

src/02-02-main.js

```
import getBase, {add, multiply} from './02-19-module';
console.log(multiply(4));
console.log(add(4));
console.log(getBase());
```

2.9 Promise(1)

❖ 비동기 처리를 위한 콜백 처리

- Callback Hell : 콜백함수들이 중첩되어 지옥을 경험함
 - 디버깅 어려움.
 - 예외처리 어려움
- ES6 Promise는 Callback Hell 문제 해결

```
// Promise 객체의 생성
const p = new Promise((resolve, reject) => {
  //비동기 작업 수행
  //이 내부에서 resolve(result)함수를 호출하면 then에 등록해둔 함수가 호출됨
  // reject(error)가 호출되거나 Error가 발생되면 catch에 등록해둔 함수가 호출됨.
});

p.then((result)=> {

})
.catch((error)=> {

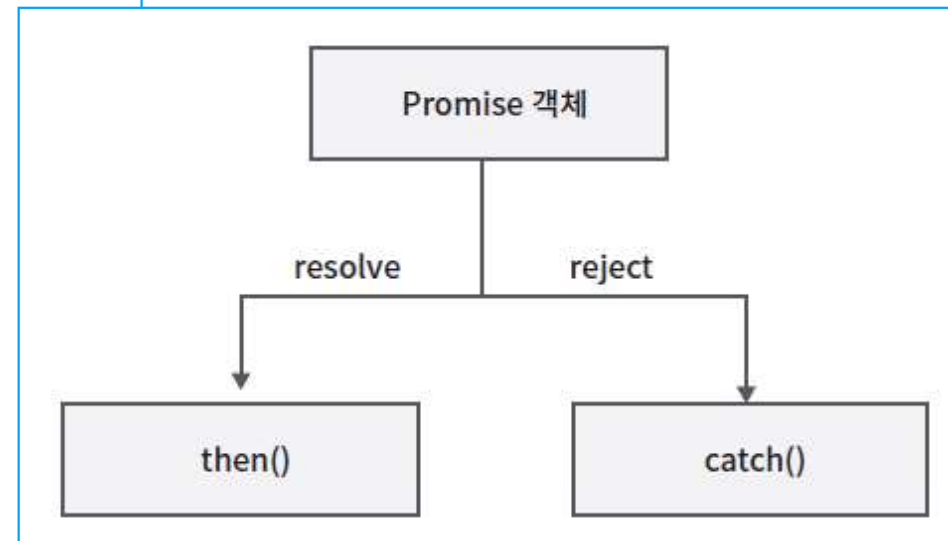
})
```

2.9 Promise(2)

❖ Promise 패턴

- 자바스크립트 비동기 처리를 수행하는 추상적인 패턴

```
const p = new Promise((resolve, reject) => {  
  setTimeout(() => {  
    var num = Math.random(); //0~1사이의 난수 발생  
    if (num >= 0.8) {  
      reject("생성된 숫자가 0.8이상임 - " + num);  
    }  
    resolve(num);  
  }, 2000);  
});  
  
p.then((result) => {  
  console.log("처리 결과 : ", result);  
}).catch((error) => {  
  console.log("오류 : ", error);  
});  
  
console.log("## Promise 객체 생성!");
```

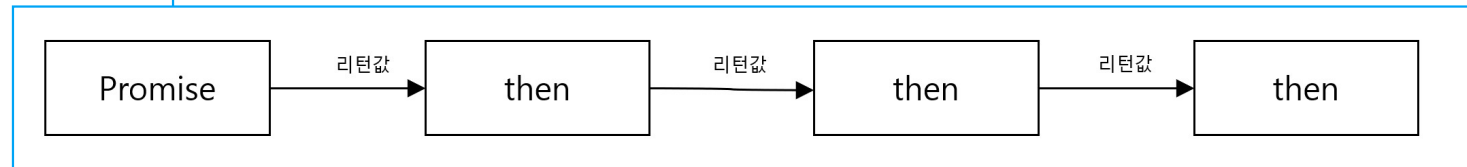


2.9 Promise(3)

❖ 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((resolve, reject)=> {  
  resolve("first!")  
})  
  
p.then((msg)=> {  
  console.log(msg);  
  return "second";  
})  
.then((msg)=>{  
  console.log(msg);  
  return "third";  
})  
.then((msg)=>{  
  console.log(msg);  
})
```



2.9 Promise(4)

❖ Promise Chain에 catch 추가

- then() 내부에서 오류가 발생하면 가장 가까운 catch()에 등록된 함수가 호출됨

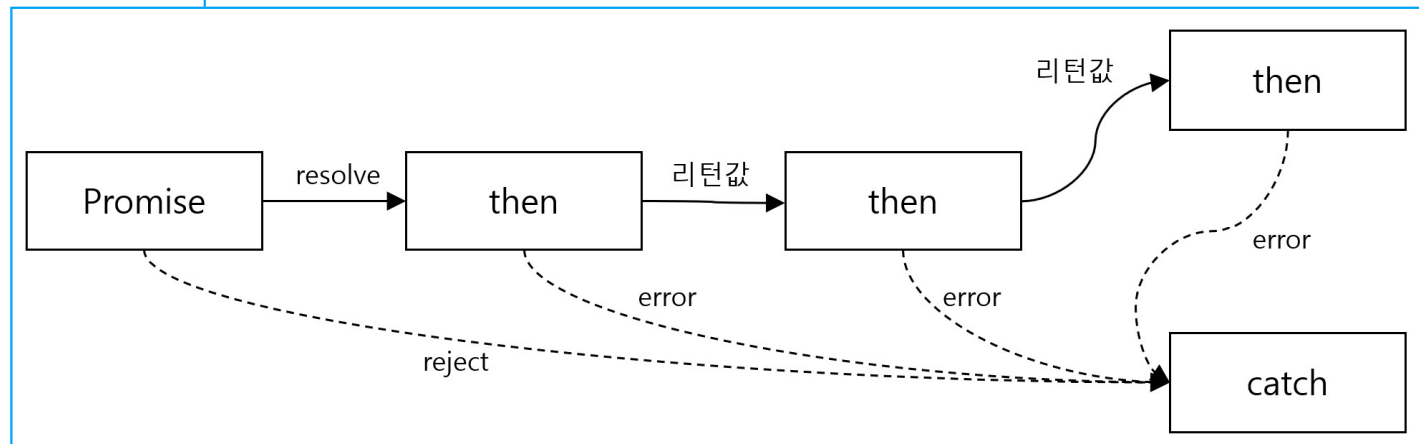
```
var p = new Promise((resolve, reject) => {  
  resolve("first!");  
});
```

```
p.then((msg) => {  
  console.log(msg);  
  throw new Error("## 에러!!");  
  return "second";  
})
```

```
.then((msg) => {  
  console.log(msg);  
  return "third";  
})
```

```
.then((msg) => {  
  console.log(msg);  
})
```

```
.catch((error) => {  
  console.log("오류 발생 ==> " + error);  
});
```



2.10 async/await(1)

❖ Promise 패턴의 단점

- 복잡한 코드 구조
 - 특히 비동기 처리를 순차적으로 하고 싶을 때
- 불편한 예외 처리
 - try/catch를 사용하고 싶다!

```
const axios = require('axios');
const listUrl = "https://todosvc.bmaster.kro.kr/todolist_long/gdhong";

//4건의 목록을 조회한 후 첫번째, 두번째 할일을 순차적으로 조회합니다.
const requestAPI = () => {
  let todoList = [];
  axios.get(listUrl)
    .then((response) => {
      todoList = response.data;
      console.log("# TodoList : ", todoList);
      return todoList[0].id;
    })
    .then((id) => {
      return axios.get(listUrl + "/" + id);
    })
    .then((response) => {
      console.log("## 첫번째 Todo : ", response.data);
      return todoList[1].id;
    })
    .then((id) => {
      axios.get(listUrl + "/" + id).then((response) => {
        console.log("## 두번째 Todo : ", response.data);
      });
    });
};

requestAPI();
```

2.10 async/await(2)

❖ async/await

```
const axios = require('axios');
const listUrl = "https://todosvc.bmaster.kro.kr/todolist_long/gdhong";

//4건의 목록을 조회한 후 첫번째, 두번째 할일을 순차적으로 조회합니다.
const requestAPI = async () => {
  let todoList;
  let response = await axios.get(listUrl);
  todoList = response.data;
  console.log("# TodoList : ", todoList);
  response = await axios.get(listUrl + "/" + todoList[0].id);
  console.log("## 첫번째 Todo : ", response.data);
  response = await axios.get(listUrl + "/" + todoList[1].id);
  console.log("## 두번째 Todo : ", response.data);
};

requestAPI();
```

2.10 async/await(3)

❖ 예외 처리는?

- Promise 객체의 catch() 가 아닌 try / catch

```
const axios = require('axios');
const listUrl = "https://todosvc.bmaster.kro.kr/todolist_long/gdhong";

//전체 목록을 조회한 후 한 건씩 순차적으로 순회하며 조회하기
//async/await 예외 처리
const requestAPI = async () => {
  let todoList;
  try {
    let response = await axios.get(listUrl);
    todoList = response.data;
    console.log("# TodoList : ", todoList);
    for (let i = 0; i < todoList.length; i++) {
      response = await axios.get(listUrl + "/" + todoList[i].id);
      console.log(`# ${i + 1}번째 Todo : `, response.data);
    }
  } catch (e) {
    if (e instanceof Error) console.log(e.message);
    else console.log(e);
  }
};
requestAPI();
```

2.11 Spread Operator(1)

❖ 일명 전개 연산자

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

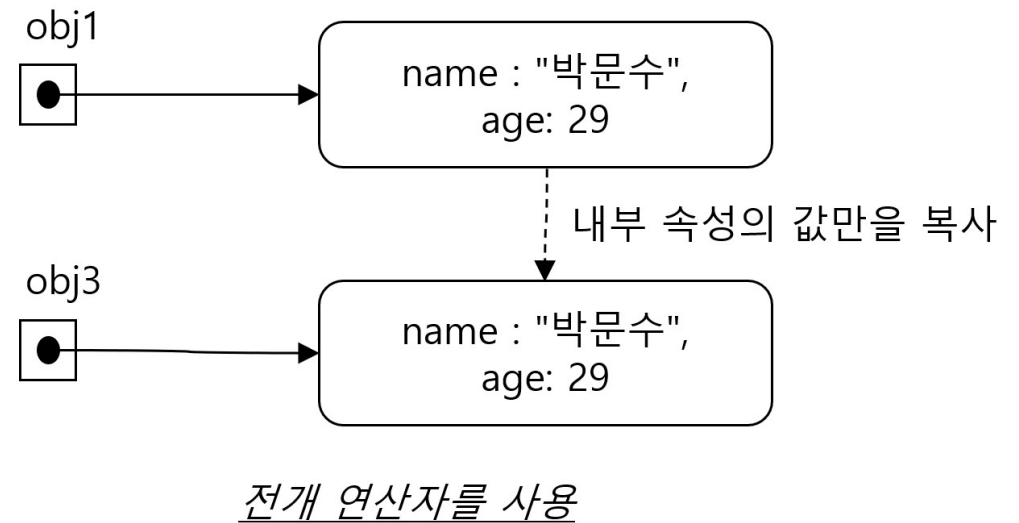
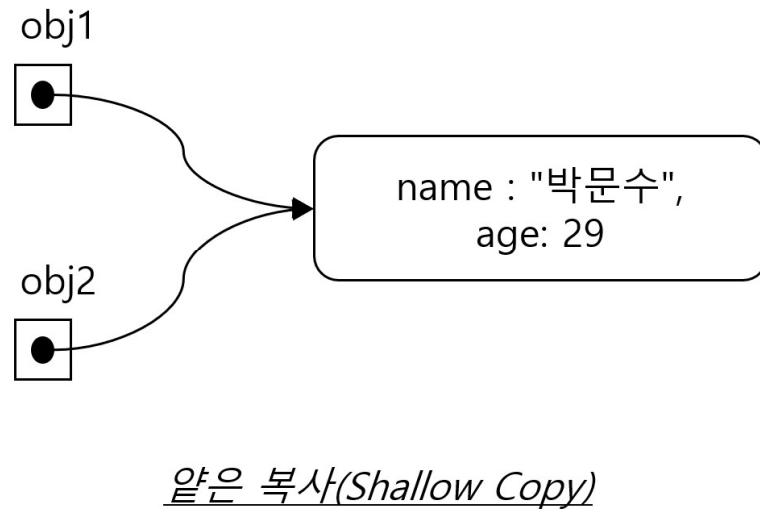
```
let obj1 = { name: "박문수", age: 29 };
let obj2 = obj1; //shallow copy! obj1, obj2는 동일한 객체를 참조
let obj3 = { ...obj1 }; //객체 내부의 값은 복사하지만 obj3, obj1은 다른 객체
let obj4 = { ...obj1, email: "mspark@gmail.com" }; //새로운 속성 추가

obj2.age = 19;
console.log(obj1); //{ name:"박문수", age:19 }
console.log(obj2); //{ name:"박문수", age:19 }
console.log(obj3); //{ name:"박문수", age:29 }   age가 바뀌지 않음
console.log(obj1 == obj2); //true
console.log(obj1 == obj3); //false

let arr1 = [100, 200, 300];
let arr2 = ["hello", ...arr1, "world"];
console.log(arr1); // [ 100, 200, 300 ]
console.log(arr2); // [ "hello", 100, 200, 300, "world" ]
```

2.11 Spread Operator(2)

❖ shallow copy와 전개 연산자 사용 비교



2.12 Class(1)

❖ ES5

- 유사 클래스 : 함수를 이용해 클래스 기능을 만들어냄
- 작성이 힘들
 - 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

2.12 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 p2 = new Person('이몽룡', '010-222-3332', '경기도');  
console.log(p1.toString());  
console.log(Person.getPersonCount());
```

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

2

2.12 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());
```