9/27 화요일 강의자료

1교시

가변변수 VS 불변변수

▼ var | const | let 뭐가 다를까? (복습 추가)

var는 다른 두가지와 변수 선언 방식에서 차이점이 있습니다.

스코프(scope)란? : 식별자가 유요한 범위를 의미

차이점

var는 js의 초기 변수 선언방법으로 여러가지 문제점이 있었습니다.

1. **같은 변수명으로 선언을하게 되면 덮어쓰게됨**

   : 이는 코드량이 많아진다면 어디서 잘못 된건지 찾기 힘든 치명적인 문제점이 있습니다.

2. **변수가 선언되지 않아도 참조 가능**

   : var는 function level scope이기 때문에 함수 밖에 선언된 var는 모두 전역변수가 됩니다.

   이는 메모리를 잡아먹어 성능적으로 낮아지며 변수명 중복으로 인한 오류를 야기합니다.

3. **변수 호이스팅**

   : js엔진은 런타임 이전에 코드를 읽으며 var로 선언된 모든 변수에 메모리를 할당하고 undefined로 초기화합니다. 선언은 호이스팅 되고 할당은 호이스팅 됩니다.

   let 같은 경우 런타임 이전 실행되며 변수 호이스팅이 되지만 선언한 곳에 도착해야지 undefined로 초기화 됩니다. 따라서 이전에 참조 할 수 없습니다.

   const 같은 경우는 선언과 동시에 초기화 해야합니다.

```
// var
var a;
a = "test1";
console.log(a); // test1

var a = "test1";
console.log(a); // test1

a = "hello1";
```

```
console.log(a); // hello1

var a = 1;
console.log(a); // 1
```

이를 보안하기 위해 js 기술 규격을 정하는 Ecma 인터네셔널이라는 기구에서 2015년 es6문법을 발표했는데요 이중 const, let이라는 상수 변수를 통해 var를 대체하게 되었습니다.

const는 변수 재선언, 재할당이 안되는 변수로 바꿀수 없는 값인 상수의 역할을 가집니다. 또한 선언과 할당을 동시에 해야합니다.

```
// const
const b; // error
b = test2;

const b = "test2";
console.log(b); // test

b = "hello2";
console.log(b); // error

const b = 2;
console.log(b); // error
```

let은 변수에 재할당이 가능하며 선언과 할당을 따로할 수 있습니다. 하지만 재선언은 안됩니다.

```
// let
let c;
c = "test3";
console.log(c); // test3

let c = "test3";
console.log(c); // test3

c = "hello3";
console.log(c); // hello3

let c = 3;
console.log(c); // error
```

4. **function level scope VS block level scope**

var는 function level scope로 함수의 코드 블록만을 scope로 인식합니다.

따라서 if문, for문 안에 선언한 변수경우 scope로 인식하지 않고 무시하게 됩니다.

예제 코드를 보면서 이해하시길 바랍니다.

```
// function level scope
var x = 1;

if (true) {
    var x = 10; // 블록 scope가 인정되지 않는다.
}
console.log(x); // 10

for (var i = 0; i < 100; i++) {
    // do nothing
}
console.log(i); // 100

function scopeTest() {
    var y = "function level scope";
    console.log(y);
}
console.log(y); // error
```

이는 개발자가 의도한 스코프와 다르게 작동할 위험요소로 의도치 않은 버그를 야기하고, 불 필요한 전역변수를 생성함으로써 유지보수 측면, 성능 측면으로 모두 안 좋습니다.

이에 반해 const, let 같은 경우 block level scope로 if문 , for문, function 등 괄호를 기준으로 scope를 인식합니다. 아래 예제 코드를 보면서 이해하시길 바랍니다.

```
// block level scope
const x = 1;

if (true) {
    const x = 10; // 블록 scope가 인정된다.
    console.log("if문 안의 x : ", x); // 10
}
console.log("if문 밖의 x : ", x); // 1

for (var i = 0; i < 100; i++) {
    // do nothing
    const y = 10;
    if (i === 99) console.log("for문 안의 y : ", y); // 10
}
// console.log("for문 밖의 y : ", y); // error

function scopeTest() {
    const y = "block level scope";
    console.log("function 안의 y : ", y); // block level scope
}
scopeTest();
console.log("function 밖의 y : ", y); // error

console.log("============================================================");

let x = 1;

if (true) {
    let x = 10; // 블록 scope가 인정된다.
    console.log("if문 안의 x : ", x); // 10
}
console.log("if문 밖의 x : ", x); // 1

for (let i = 0; i < 100; i++) {
    // do nothing
    if (i === 99) console.log("for문 안의 i : ", i); // 99
```

```
}
// console.log("for문 밖의 i : ", i); // error

function scopeTest() {
    let y = "block level scope";
    console.log("function 안의 y : ", y); // block level scope
}
scopeTest();
console.log("function 밖의 y : ", y); // error
```

결론!

유지보수시 왜인지 모를 error로 부터 벗어나기 위해 상수로 사용할 변수는 const를 변수는 let으로 선언하자! (~~var Bye Bye~ 😊~~)

참고자료

js의 역사 : https://erokuma.tistory.com/entry/자바스크립트의-역사와-ECMAScript-대해

var의 문제점 : https://happycording.tistory.com/entry/let-const-란-왜-써야만-하는가-ES6

비교 정리 : https://velog.io/@seeh_h/var-let-const

모듈 내보내고 가져오기

▼ 모듈을 불러오는 방법은?(require VS import)

require문법과 import 문법이 있습니다.

(require / exports) 는 NodeJS에서 사용되고 있는 CommonJS 키워드이고 Ruby 언어 스타일과 비슷하다라고 볼수 있습니다.

cf. CommonJS는 JS에서 모듈화작업을 할 수 있게 만들어 줍니다.

(import / export) 는 ES6(ES2015)에서 새롭게 도입된 키워드로서 Java나 Python 언어 방식과 비슷하다.

차이점

1. require()는 코드를 적은 곳에서 불러와 어느 지점에서나 호출이 가능한 반면 import는 항상 맨 위로 이동하여 파일 시작부분에서만 실행됩니다.(호이스팅)

- 하지만 비동기를 사용하면 나중에 불러올 수 있습니다.

  참고 : https://inpa.tistory.com/entry/JS-📚-모듈-사용하기-import-export-정리?category=889099#브라우저(HTML)에서_모듈_사용_하기

- cf. 호이스팅이란 쉽게 생각하면 실행될때 최 상단에 선언된것으로 인식해서 먼저 실행된다고 이해하시면 됩니다.

  호이스팅 예제 : https://simplejs.gitbook.io/olaf/09.-hoisting

2. 일반적으로 import는 모듈의 필요한 부분만 불러올수 있기 때문에 더 선호되며 성능적으로 우수합니다.
3. 하지만 Import 를 사용하기 위해서는 webpack이나 Babel등의 컴파일러가 필요하다는 단점이있습니다.

- cf. 컴파일러란 우리가 작성한 언어(JavaScript)를 기계가 알아들을 수 있는 기계어(어셈블리어)로 번역해준다.

  (기본 개발 용어 알아보기 : https://www.youtube.com/watch?v=GYmuQJiPeM4)

이러한 import 방식을 사용할 수 있게 최근 나오는 브라우저들은 ES6문법인 <script type="module" src="index.js"></script>을 지원해줍니다. 이렇게 명시된 src는 import, export 구문을 사용할 수 있습니다.

하지만 구형 브라우저는 이 모듈을 지원하지 않으며 이런 경우 webpack이나 Babel, parcel등의 모듈 번들러를 통한 변환과정을 거쳐야 합니다.

참고 자료

require과 Import의 차이점 : https://inpa.tistory.com/entry/NODE-📚-require-⚔️-import-CommonJs와-ES6-차이-1

비동기로 import하기 : https://inpa.tistory.com/entry/JS-📚-모듈-사용하기-import-export-정리?category=889099#브라우저(HTML)에서_모듈_사용_하기

▼ import, export 사용하기(ES6 Module)

export 문법 설명

모듈을 내보내는 방법으로는 default export, named export 가 있습니다.

1. default export :

   export default를 사용해서 기본값으로 내보낼 함수, 변수, 클래스 등을 지정할 수 있습니다.

   선언을 할 때 바로 사용하는 방법과

   선언 후 나중에 지정해주는 방법이 있습니다.

```
// 선언시 지정
export default function euroToWon(money) {
    var won = money * 1329.87;
    return won;
}


// 선언 후 나중에 지정
function euroToWon(money) {
    var won = money * 1329.87;
    return won;
}

export default elevation;
```

2. named export :

   기본값을 지정하지 않고 export를 할 수도 있습니다.

   이 방법도 default export와 마찬가지로 선언시와 선언 후 지정하는 방법이 있습니다.

```
// 선언시 지정
export function yuanToWon(money) {
    var won = money * 192.53;
    return won;
}

export function yenToWon(money) {
    var won = money * 9.88;
    return won;
}


// 선언 후 나중에 지정
function yuanToWon(money) {
    var won = money * 192.53;
    return won;
}

function yenToWon(money) {
    var won = money * 9.88;
    return won;
}

export { yuanToWon, yenToWon };
```

3. export 시 이름 변경

   export 할 때 기존의 이름이 아닌 다른 이름으로 바꿔서 내보내기를 할 수 도 있습니다.

```
// export 시 이름 변경
function test() {
    console.log("export test");
}

export { test as changeName };
```

export를 사용해서 내보내기를 한 모듈만 import를 할 수 있습니다 .

import 문법 설명

import를 하는 방법은 export된 방법에 따라 다르게 적용됩니다.

지정한 경로에 export 된 것을 불러오는 방법은 다음과 같습니다.

```
import * as asia from "./exchange/asia.js"; // 모두 불러야 하는 경우 * 로 불러와서 이름을 변경해 줄 수 있습니다.
import { dollarToWon } from "./exchange/dollar.js"; // named export 된경우
import euroToWon, { changeName } from "./exchange/euro.js"; // default export가 된 경우에는 그냥 불러옴
```

ES6문법중 module 사용하기 : https://snowdeer.github.io/javascript/2020/01/09/html5-how-to-use-module/

type module : https://ko.javascript.info/modules-intro

참고 : https://inpa.tistory.com/entry/JS-📚-모듈-사용하기-import-export-정리#모듈_import

2교시

템플릿 리터럴

▼ `` 사용법

이미 많이 사용해보셨죠?

실습 코드를 보면서 복습해 보겠습니다! 🙂

화살표 함수

▼ funciton VS arrow funciton

차이점1

함수 호이스팅

function은 호이스팅에 의해 최상단에 선언되지만 arrow function은 호이스팅되지 않습니다.

```
// 호이스팅 차이
// function
```

```
func(); // hi~
function func() {
    return console.log("hi~");
}

// arrow function
arrFunc1(); // error
const arrFunc1 = () => console.log("hi~");

// arrFunc1(); // hi~

const arrFunc2 = () => {
    return console.log("hi~");
};
```

차이점2

this의 사용법

일반function은 호출한 객체를 가리키지만

arrow function 은 부모(상위 스코프)의 this를 가리킨다 (렉시컬 스코프)

```
// this 차이
// function
function fun() {
    this.name = "하이";
    return {
        name: "바이",
        speak: function () {
            console.log(this.name); // 바이
        },
    };
}

const fun1 = new fun();
fun1.speak(); // 바이

// arrow func
function arrFun() {
    this.name = "하이";
    return {
        name: "바이",
        speak: () => {
            console.log(this.name); // 하이
        },
    };
}

const fun2 = new arrFun();
fun2.speak(); // 하이
```

차이점3

생성자 함수로 사용여부

일반 함수에는 prototype 프로퍼티가 있기 때문에 생성자로 사용할 수 있지만

화살표 함수에는 prototype 프로퍼티가 없어 생성자로 사용할 수 없습니다.

```
// func
function fun() {
    this.num = 1234;
}

const funA = new fun();
console.log(funA.num); // 1234

//arrow func
const arrFun = () => {
    this.num = 1234;
};

const funB = new arrFun(); // Error
```

화살표함수를 사용하면 안되는 경우

1. 메소드를 화살표 함수로 정의하는 것은 지양해야합니다.

   : this가 메소드를 호출한 객체가 아닌 상위 스코프의 this를 가르키기 때문입니다.

2. 생성자를 만들면 안됩니다.

   : 화살표 함수는 prototype 프로퍼티를 가지고 있지 않기 때문에 사용할 수 없습니다.

참고자료

정리 : https://poiemaweb.com/es6-arrow-function

호이스팅 : https://bgpark.tistory.com/21

클래스

▼ class, instance간의 관계

예를들자면

포유류 라는 class 안에 강아지, 고양이라는 instance가 있는 것 입니다.

붕어빵 틀과 붕어빵으로 예시를 많이 들어보셨죠?

class는 인스턴스가 가지고 있어야하는 속성과 메서드 들을 포함한 틀이라고 생각하시고

instance는 class라는 틀로 찍어낸 실제 객체를 말합니다.

차를 만들때 설계도르 만들었다고 실제로 차가 생기는건 아니죠

const myCar = new Car() 를 통해 Car라는 설계도로 인스턴스를 만들겠다고 하는 순간 비로서 myCar라는 객체가 생겨나는 것입니다.

이때 Class안에 constructor는 생성자라는 특수한 메서드로서 인스턴스를 생성시에 자동으로 호출되므로 특별한 절차 없이 객체를 초기화 합니다.

사용예시를 코드를 통해 확인해 보겠습니다 🙂

3교시

forEach() 함수

▼ 향상된 for 문

forEach 문

array에서만 사용가능한 메서드로 원소들을 나열해서 검사하는 기능이 있습니다.

```
const arr = ["엘리스", "sw", "3기", "레이서 분들"," 화이팅"]


for(let i=0; i> arr.length; i++){
    console.log(item)
}

arr.forEach(item => {
    console.log(item)
});
```

https://developer.mozilla.org/ko/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach

for ... in 문

객체에 사용하는 for문으로 객체 안에 있는 프로퍼티들을 가지고 반복적인 동작들을 수행할 때 사용합니다.

객체의 key값에 바로 접근할 수 있기 때문에 일반적인 for문으로는 대체할 수 없는 특별한 반복문입니다.

객체 내부의 프로퍼티값들을 하나씩 다뤄야 하는 경우에 유용하게 사용할 수 있습니다.

```
let obj = {
    name : "엘리스 SW",
    age : 3,
```

```
    jobs : "레이서",
    track : "엔지니어"
}

for(let key in obj){
    console.log(key)
    console.log(obj[key])
}
```

이때 주의할 점은 객체는 정수형 프로퍼티 네임을 오름차순으로 먼저 정렬하고, 나머지 프로퍼티들은 추가한 순서대로 정렬하는 특징이 있기 때문에 순서에 유의해서 사용해야 합니다.

for …of 문

배열을 다루는 반복문으로 배열안에 값들을 변수로받아 사용합니다 forEach와 마친가지로 배열의 길이만큼 자동으로 실행하기 때문에 간략하게 사용할 수 있습니다.

```
for (let item of arr) {
    console.log(item);
}
```

사실 배열도 객체이기 때문에 for in에 배열도 사용할 수 있지만 for in문은 객체에 최적화 되어있다보니 실행하는 환경에따라 배열에 있는 length프로퍼티 같은 것들이 변수로 할당될 가능성이 있어 배열에는 for in문을 사용하지 않기를 권장합니다.

배열의 index를 활용하고 싶다면 그냥 for문을 사용하길 권장드립니다.

map() 함수

▼ map 심화

여러가지 map활용 예제를 보면서 map 사용법을 익혀봅니다!

▼ map 사용시 기존array가 변경된다면?

array가 변경된다면 변경된 array를 콜백 함수에 전달하게 됩니다.

코드를 통해 예제를 확인하세요!

reduce() 함수

▼ reduce()란?

reduce 함수는 배열의 각 요소에 대해 주어진 리듀서(reducer)함수를 콜백함수로 실행하고 하나의 결과값을 반환합니다.

reduce()는 2개의 인자값을 받습니다.

- `callbackfn` : 배열의 각 요소에 실행할 리듀서 함수
- `initialValue` : 리듀서 함수에 첫번째 누적값으로 제공할 초기 값(없으면 배열의 첫번째 요소가 초기값이 됨)

리듀서함수는 4개의 인자값을 받습니다.

- `accumulator` : 누적값
- `currentValue` : 현재 요소값
- `currentIndex` : 현재 요소값의 index
- `array` : reduce()를 호출한 배열

사용방법과 예시는 코드를 통해 확인해보겠습니다 🙂

```
const arr = [1, 2, 3, 4, 5];

console.log("초기값 없음");
arr.reduce((total, now, nowIdx, arr) => {
    console.log("total :", total);
    console.log("now :", now);
    console.log("nowIdx :", nowIdx);
    console.log("arr :", arr);
    console.log("\n");

    return total + now;
});

console.log("\n초기값 있음");
arr.reduce((total, now, nowIdx, arr) => {
    console.log("total :", total);
    console.log("now :", now);
    console.log("nowIdx :", nowIdx);
    console.log("arr :", arr);
    console.log("\n");

    return total + now;
}, 0);
```