Java Script 15, 16, 18 (25.02.07)

6주차 스터디 멋쟁이 사자처럼 13기

JavaScript란?

『자바스크립트(JavaScript)는 웹 개발에서 가장 널리 사용되는 프로그래밍 언어 중 하나
HTML과 CSS와 함께 웹사이트를 구성하는 핵심 요소이며, 웹 페이지를 동적으로 만들고,
사용자와 상호작용 한다.』

웹 환경에서 JavaScript의 목적 》 HTML 조작과 변경

자바스크립트의 특징

☑ 인터프리터 언어
코드를 한 줄씩 읽고 실행하는 방식(컴파일 과정 필요 없음).
코드 수정 후 바로 실행 가능.

☑ 동적 타입(Dynamically Typed)
변수의 타입을 명시적으로 선언할 필요 없음.
실행 중에 변수의 타입이 변경될 수 있음.

☑ 객체 기반(Object-Oriented)
기본적으로 객체(Object) 단위로 구성.
객체를 생성하고 조작하는 것이 주요한 프로그래밍 방식.

☑ 이벤트 기반(Event-Driven)
버튼 클릭, 마우스 이동 등의 이벤트를 감지하고 실행 가능.
addEventListener() 등을 활용.

☑ 비동기 처리(Asynchronous)
콜백 함수(Callback), 프로미스(Promise), async/await 등을 이용해 비동기 작업 수행 가능.
AJAX, API 호출, 데이터베이스 연동 등에 활용.

언어	실행 환경	컴파일 방식	프레임워크/라이브러리
JavaScript	웹 브라우저(Chrome) + Node.js(백엔드)	인터프리터 방식(즉시 실행)	React, Vue, Node.js
C, C++	운영체제(window, Linux 등)에서 컴파일 후 실행	컴파일 후 실행(빠름)	Qt, Boost
Python	인터프리터로 실행(스크립트 언어, 웹/데이터 분석, AI)	인터프리터 방식(즉시 실행)	Django, Flask
Java	JVM(Java Virtual Machine) 위에서 실행 (멀티플랫폼)	바이트코드 변환 후 JVM에서 실행	Spring, Android SDK, Hibernate

1. 콜백 함수 - 15섹션

정의 : 함수에 매개변수(파라미터)로 들어가는 함수

용도 : 순차적으로 실행하고 싶을 때만 쓰는 것

JavaScript의 함수는 "값"처럼 취급(다른 함수의 **매개변수로 함수를 전달(=콜백)** 할 수 있음)

콜백(callback) 코드 예시

```
function doSomething(callback){
    console.log("첫번째");
    callback();// 콜백 함수를 나중에 실행
  }

  function finish(){
    console.log("두번째");
  }

  doSomething(finish);
```

[1]
doSomething(finish) 호출된다.
[2]
doSomething 내부가 실행되며, 가장 먼저 console.log("첫번째')가 실행된다.
따라서 콘솔에는 "첫번째" 라는 문구가 찍힌다.
[3]
다음으로 callback() 부분이 실행되는데, 여기서 callback은 실제로 finish 함수이므로 결과적으로 finish() 함수를 호출하게 된다.
[4]
finish() 함수에서는 console.log("두번째")를 실행하므로, 콘솔에 "두번째"이라는 문구가 이어서 찍힌다.

[추가설명]

finish 함수 자체를 doSomething의 첫 번째 인자(인수)로 넘긴다.
finish라는 함수 참조가 callback이라는 매개변수 이름으로 받게 된다.
이 순간, **callback 변수가 곧 finish 함수를 가리키게 됨**

(2) doSomething 내부
callback()을 호출
callback이 곧 finish 함수를 참조하고 있으므로,
callback()은 결국 finish()와 완전히 동일한 호출이다.

비동기 처리가 필요한 상황(서버 요청, 파일 읽기 등)에서 **"작업이 끝나면 어떤 함수를 실행할 것인가?"** 명시적으로 표현할 수 있는 장점

코딩 애플

https://www.youtube.com/watch?v=-iZlNnTGotk

2. 비동기 콜백 & Ajax - 15섹션

[동기]
함수 호출 후 결과가 반환될 때까지 코드 실행이 멈추는 방식입니다. 한 줄 한 줄 순서대로 실행되기 때문에, 서버 요청(네트워크 통신)처럼 **시간이 오래 걸리는 작업이 있으면 그 동안 프로그램 흐름이 멈춰 있게 된다.**

[비동기]
프로그램이 특정 작업(예: 네트워크 요청, 파일 읽기, 타이머 등)을 수행할 때 그 작업이 완료될 때까지 메인 스레드(주 흐름)가 '**멈추지 않고**' 계속해서 다음 코드들을 실행할 수 있도록 하는 방식

간단히 말해서 시간이 오래 걸리는 작업을 "백그라운드"에서 처리하고
그 작업의 완료 시점에 맞춰 결과를 받아와 후속 처리 -> **콜백**

[비동기 콜백]
주로 서버로부터 데이터를 가져오거나, 파일 읽기 등 시간이 걸리는 작업이 있을 때 사용

자바스크립트는 기본적으로 **단일 스레드** 기반이라, 오래 걸리는 작업이 있을 경우 "다른 작업을 막지 않고" 병렬(원래는 직렬)로 진행하는 듯한 (node.js 이벤트 루프를 통한) 매커니즘 방식을 사용함

JavaScript는 함수 호출 → 기다리지 않고 바로 다음으로 넘어감 → 실제 결과가 준비되는 시점에 콜백 또는 .then()을 통해 결과 수신

자바스크립트가 비동기를 사용하는 이유는
비동기가 아닌 동기의 문제점 이슈 / API 통신의 결과 처리를 원활하게 진행하기 위해서

동기 비동기 예시 설명

```
function getDataFromServer(callback){
    // 서버에서 데이터 가져온다고 가정
    setTimeout(()=>{
      const data ={name:"Alice",age:20 };
      callback(data);// 서버로부터 받은 data를 콜백 함수로 넘김
    },1000);// 1초 뒤에 콜백 호출
  }

  getDataFromServer(function (result){
    console.log("서버에서 받아온 결과:",result);
  });
```

서버에서 데이터를 가져온다"는 상황을 setTimeout으로 흉내낸 것이며, 1초 뒤에 데이터를 받을 수 있게 됩니다. 동기 코드였다면 1초 기다리는 동안 코드 흐름이 멈추겠지만, 비동기(콜백) 방식이므로 그 사이에 다른 작업도 진행

Ajax (Asynchronous JavaScript And XML 약자): 서버와 통신하며 페이지 전체를 새로 고치지 않고, 부분적 데이터만 교환해서 화면을 갱신하는 기법

Ajax 요청 Fetch API - 현대적인 방법

```
fetch("https://jsonplaceholder.typicode.com/todos/1")
  .then(response =>response.json())
  .then(data =>{
    // 콜백 형태로 then 안에서 비동기 결과 받음
    console.log("서버 데이터:",data);
  })
  .catch(error =>{
    console.error("에러 발생:",error);
  });
```

[1]
fetch(url): 주어진 링크로 GET 요청을 보내고, Promise 객체를 반환합니다.
[2]
then(response => response.json()): 첫 번째 then 콜백에서 서버로부터 받은 응답을 JSON 형태로 **파싱**합니다. (response.json()도 비동기이기 때문에 다시 Promise를 반환합니다.)
[3]
두 번째 then(data => ...): JSON 파싱이 완료되면 최종 data를 반환받습니다. 콜백(익명 함수) 안에서 data를 사용할 수 있습니다.
[4]
catch(error => {...}): 요청 과정(네트워크 문제 등)에서 에러가 발생하면 이 catch 블록에서 에러를 처리합니다

fetch 호출 시점에는 당장 서버 응답이 준비되어 있지 않습니다. 따라서 결과를 즉시 반환하지 않고, "나중에 결과가 준비되면 알려주겠다(콜백을 호출하겠다)"는 Promise를 반환

코딩 애플

https://www.youtube.com/watch?v=nKD1atl6cAw

3. Closure(클로저) - 16섹션

배우기 전,

[내부 함수]
내부 함수는 자신을 둘러싸고 있는 '외부 함수'의 범위(스코프) 안에 있는 변수들에 접근

[외부 함수]
내부 함수에게 접근 권한을 주거나, 혹은 제한하는 형태로 범위(스코프)를 제공

```
function outerFunction(){
    let outerVar ='외부 변수 값';

    function innerFunction(){
      console.log(outerVar);// 내부 함수에서 외부 함수의 변수에 접근
    }

    innerFunction();// 내부 함수를 실행
  }

  outerFunction();
  // 출력: "외부 변수 값"
```

[1]
outerFunction(); 출력하기위해서 function 함수로 이동

[2]
let는 '외부 변수 값' 선언문

[3]
내부 함수인 innerFunction()이 존재 -> 내부 함수를 실행

[4]
fucntion innerFunction()에서 console.log(outerVar)를 통해 내부 함수에서 외부 함수의 변수에 접근 가능하게끔 유도

[5]
outerFunction(); -> 출력 : 외부 변수 값

앞에서는 단순히 내부 함수가 외부 함수 변수를 **"접근"**하는 예시를 보여줬다면

클로저(Closure)는 함수가 자신이 **선언된 환경(스코프)을 기억**하고, **스코프에 접근할 수 있는 개념**을 뜻함

내부 함수가 **외부 함수의 변수에 접근**할 수 있고,
심지어 **외부 함수가 종료된 후에도 그 변수를 유지**할 수 있도록 할 수 있다.

더 쉽게 설명하면 》 내부 함수가 외부 함수 변수를 **"기억"**하고 사용할 수 있게끔 하는 구조 = 클로저

클로저 형태

```
function createCounter(){
   let count =0;// 외부 함수 스코프의 변수

   // 내부 함수(클로저)
   function increase(){
     count++;//count = count + 1
     console.log(`현재 count 값: ${count}`);
   }

   return increase;// 내부 함수를 반환
 }//메모리가 없어짐.

 const counter =createCounter();
 counter();// 현재 count 값: 1
 counter();// 현재 count 값: 2
 counter();// 현재 count 값: 3
```

[1]
createCounter 함수를 호출하면, count 변수가 초기값 0으로 설정됩니다.

[2]
createCounter는 내부 함수 increase를 반환하고 종료됩니다.

[3]
원래라면 createCounter가 끝났으니 count라는 변수가 사라질 것 같지만,
내부 함수 increase가 count에 대한 참조를 '기억'하고 있어 count가 계속 유지됩니다.

[4]
counter()를 호출할 때마다 count가 증가하고, 콘솔에 로그가 찍힙니다.

클로저 심화 과정

private variable (은닉화/캡슐화)

```
function createSecret(secretValue){
  // secretValue는 외부에서 직접 접근 불가능
  return function getSecret(){
    return secretValue;
  };
}

const secretGetter =createSecret('비밀 메시지');
console.log(secretGetter());// "비밀 메시지"
```

[1]
secretValue는 createSecret 함수 스코프 안에만 존재합니다.

[2]
createSecret 함수가 종료되어도, 반환된 내부 함수(getSecret)는 secretValue를 기억하고 있으므로 계속 접근할 수 있습니다.

[3]
그러나 클로저 내부에서 직접 노출하지 않으면 외부에서 마음대로 secretValue를 수정하거나 읽을 수 없습니다.

클로저 활용 예시

```
function createClickHandler(initialCount =0){
  let count =initialCount;

  return function (){
    count++;
    console.log(`버튼이 ${count}번 클릭되었습니다.`);
  };
}

const button =document.querySelector('button');
button.addEventListener('click',createClickHandler());
```

count가 클로저를 통해 유지되므로, 버튼을 누를 때마다 누적된 클릭 횟수가 출력

4. 함수의 호출(Apply) - 18섹션

함수를 호출할 때 사용하는 메서드는 크게 call(), **apply()**, bind() 가 있다.

함수와 **this(문맥)**를 연결해 사용할 때 중요한 메서드!

Call()

```
function introduce(greeting,age){
    console.log(`${greeting}! 제 이름은 ${this.name}이고, 나이는 ${age}살입니다.`);
  }

  const person ={name:"홍길동"};

  // call() 사용
  introduce.call(person,"안녕하세요",20);
  // 출력: "안녕하세요! 제 이름은 홍길동이고, 나이는 20살입니다."
```

introduce 함수를 호출하면서 this를 person 객체로 바인딩

person , “안녕하세요” -> greeting, 20 -> age

---

apply()

```
function sumAll(){
    return Array.from(arguments).reduce((acc,cur)=>acc + cur,0);//순회
  }

  function introduce(greeting,age){
    console.log(`${greeting}! 제 이름은 ${this.name}이고, 나이는 ${age}살입니다.`);
  }

  const person ={name:"홍길동"};

  // apply() 사용
  introduce.apply(person, ["안녕하세요",20]); //배열 형태
  // 출력: "안녕하세요! 제 이름은 홍길동이고, 나이는 20살입니다."

  // apply()로 sumAll에 배열 전달하기
  console.log(sumAll.apply(null, [1,2,3,4]));// 10
```

call() 함수 호출 + 전개 연산

이처럼 **apply**는 첫 번째 인자로 **this**로 쓸 객체를, 두 번째 인자로 인자 배열을 넘기는 메서드입니다.
만약 배열 형태가 아닌 여러 인자를 쉼표로 직접 나열하고 싶다면 **call()**을 사용할 수도 있고,
영구적으로 **this**를 묶어서 새 함수를 만들고 싶다면 **bind()**를 사용할 수도 있습니다

---

bind()

```
function introduce(greeting,age){
  console.log(`${greeting}! 제 이름은 ${this.name}이고, 나이는 ${age}살입니다.`);
}

const person ={name:"홍길동"};

// bind() 사용
const introduceWithPerson =introduce.bind(person,"안녕하세요");

// introduceWithPerson을 호출하면 this가 이미 person으로 고정됨
introduceWithPerson(25);
// 출력: "안녕하세요! 제 이름은 홍길동이고, 나이는 25살입니다."
```

introduce.bind(person, "안녕하세요")는 this가 person으로 고정된 새로운 함수를 반환합니다.

그 함수를 변수 introduceWithPerson에 할당해 두었다가 원하는 시점에 호출

---

call(): this 바인딩 + 함수 즉시 호출, 인자를 쉼표로 구분
apply(): this 바인딩 + 함수 즉시 호출, 인자를 배열(혹은 유사 배열)로 전달
bind(): this 바인딩 + 새로운 함수 반환 (지정된 this가 영구적으로 유지)

call(): 인자 개수가 명확하고 쉼표로 전달하기 편할 때, this를 바인딩해서 바로 함수를 호출하고 싶을 때
apply(): 인자 배열이 있거나, 유사 배열을 인자로 넘기기에 편리할 때
bind(): 특정 객체에 결합된 함수를 나중에 여러 번 호출하거나, 이벤트 핸들러 등에서 this를 유지하고 싶을 때