Javascript 심화

고급 함수

김태민

저번에 우리는

ECMAScript

ECMAScript는 자바스크립트의 표준 사양

ES6(2015)부터 화살표 함수, let/const, 모듈 등 현대적 기능 추가

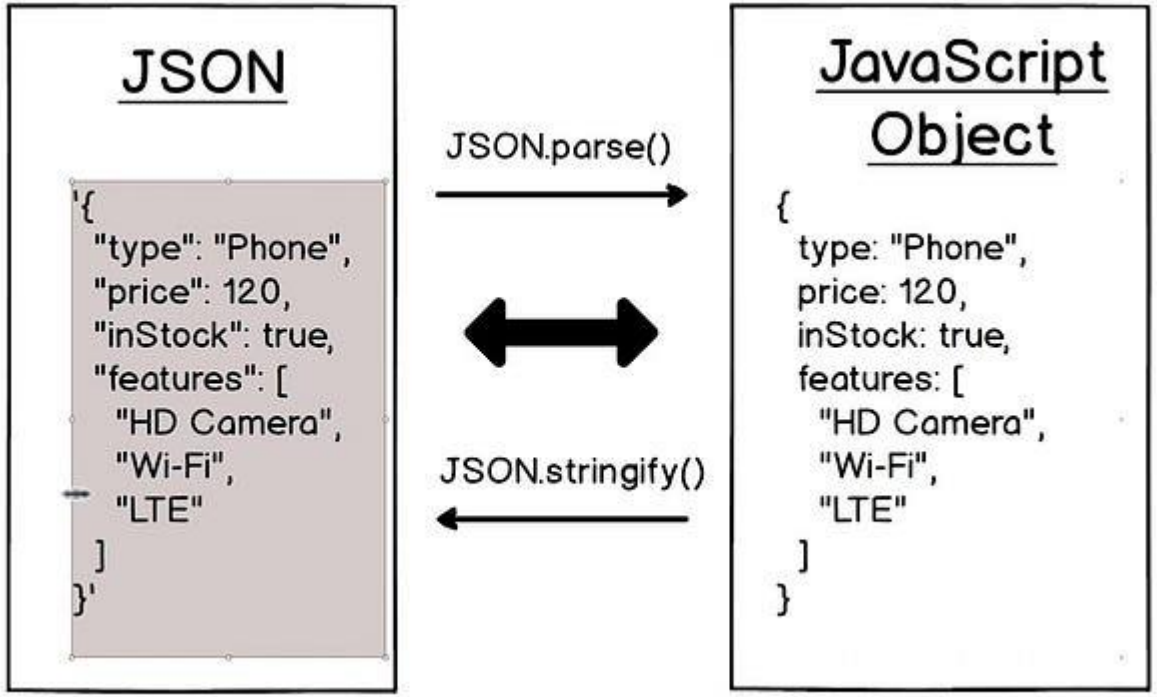
ES6+는 ES2015 이후 버전, 웹 개발 필수

```
const greet = (name) => `안녕, ${name}!`;
console.log(greet("철수")); // 출력: 안녕, 철수!
```

JSON과 객체리터럴

JSON: 문자열, {}(객체) 또는 [](배열), 엄격한 구문

객체 리터럴: 자바스크립트 코드, {}로 객체 생성, 함수 포함 가능



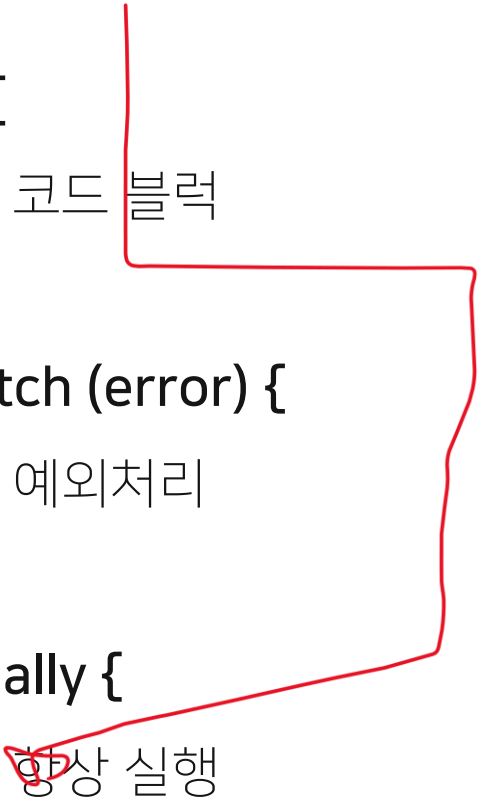
예외처리

Try-Catch

```
try {
    // 코드 블럭

} catch (error) {
    // 예외처리


}
```

```
try {
    // 코드 블럭

} catch (error) {
    // 예외처리


}
```

```
try {
    // 코드 블럭

} catch (error) {
    // 예외처리


} finally {
    // 항상 실행
}
```



계산기 제작

계산기 서비스 목표

숫자판(0~9)과 연산 키(+, -, ×, ÷)로 사칙연산 수행

JSON 배열([])로 계산 기록 저장

제공한 코드 기반 구체적인 로직 직접 생성

실습 환경

실습환경

OS : Window / Mac

브라우저 : Chrome

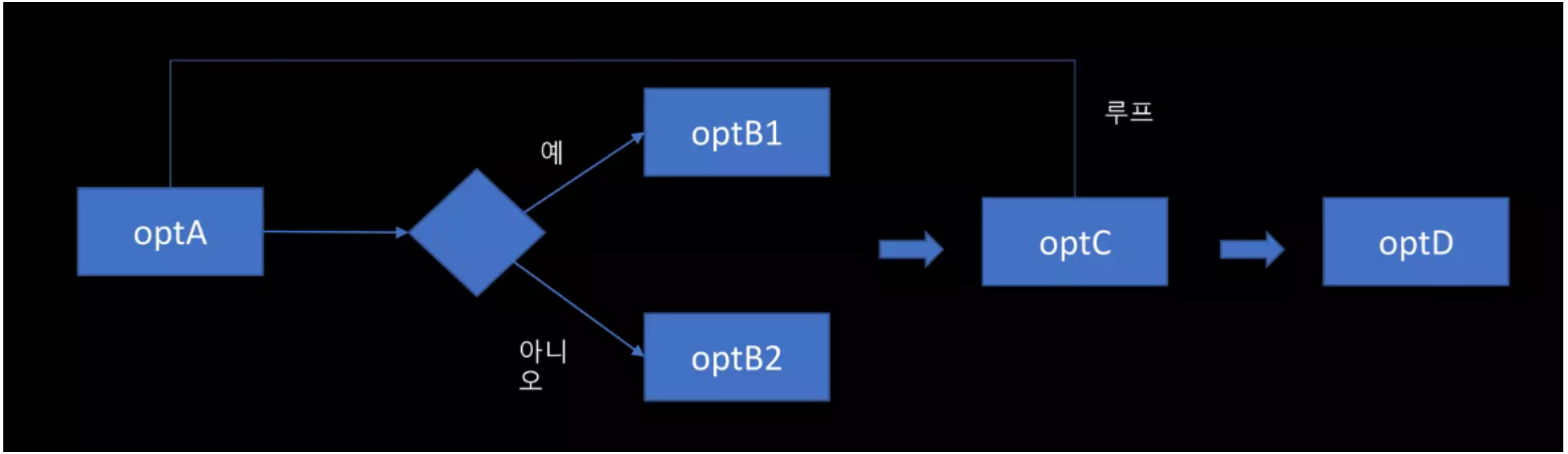
에디터 : VS Code https://code.visualstudio.com/

VS Code 익스텐션 : ESLint, Prettier, HTML CSS Support, HTML to CSS autocompletion, Auto Rename Tag, Auto Close Tag, htmltagwrap

Git : git bash, github 가입

함수형프로그래밍

명령형 코드



함수형 코드

명령형 코드 vs 함수형 코드

```
function f(list, length) {
    let i = 0;
    let acc = 0;
    for (const a of list) {
        if (a % 2) {
        acc = acc + a * a;
        if (++i == length) break;
        }
    }
    console.log(acc);
}
```

```
const add = (a,b) => a + b;
const f = (list, length) =>
        go(list,
        filter(a => a % 2),
        map(a => a * a),
        take(length),
        reduce(add));
```

함수형 프로그래밍

일급함수의 기능과 **순수함수**의 특징을 가지고 조합하는 프로그래밍 패러다임

일급 함수 : 함수를 값으로 다룰 수 있고, 변수에 담을 수 있고, 함수의 인자 혹은 결과로 사용할 수 있다.

순수함수 : 같은 인자를 받았을 때 항상 같은 결과를 내는 함수, 외부요인으로 값이 변경되지 않는 함수.

```
const add = a => b => a + b;
const add5 = add(5);
console.log(add5)
add5(10);
```

```
순수 함수의 예

const add = (a,b) => a + b
```

```
불순 함수의 예

let c = 10;
const add = (a,b) => a + b + c
```

클로저

클로저

함수형 프로그래밍 언어에서 사용되는 특성

난해하기로 유명함

"클로저는 함수와 그 **함수가 선언된 렉시컬 환경**과의 조합이다."

렉시컬 스코프

변수의 접근 가능 범위를 정의하는 개념

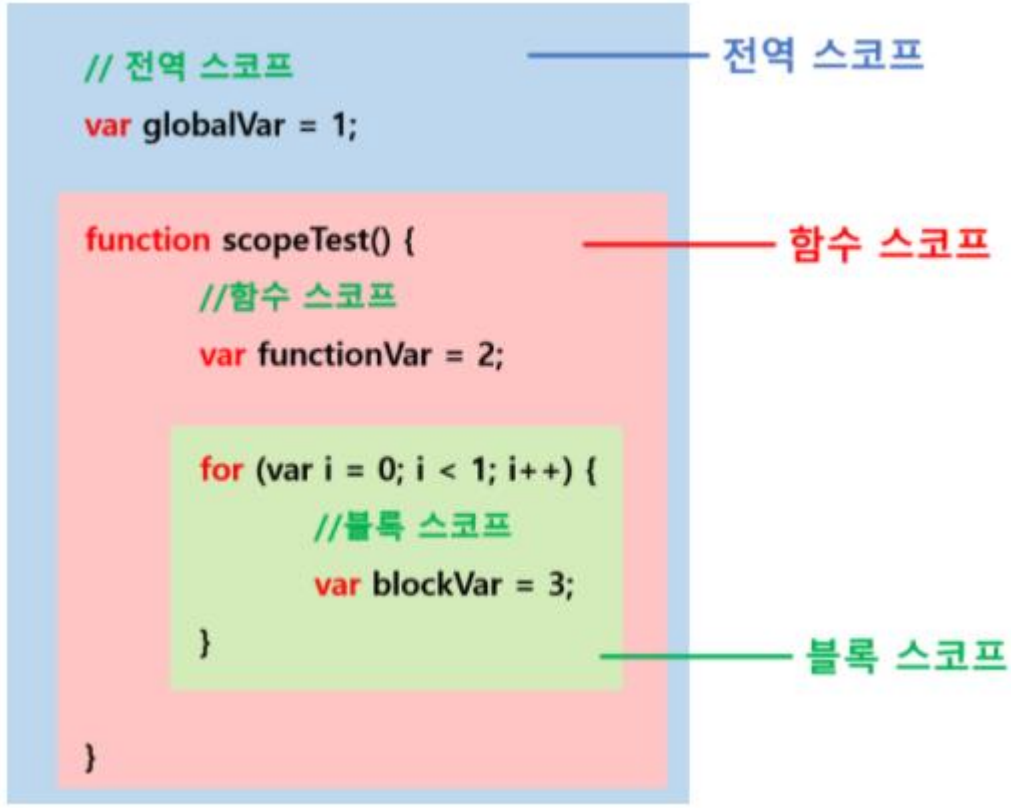
자바스크립트는 **렉시컬 스코프(정적 스코프)**를 사용

변수가 어디서 선언되었는지에 따라 접근 가능 여부가 결정

함수를 어디서 호출했는지가 아니라

함수를 어디에 정의했는지에 따라

상위 스코프를 결정



전역변수와 지역변수

전역 변수는 어디서나 접근 가능,

지역 변수는 특정 스코프 내에서만 접근 가능

```
let globalVar = "I'm global";
function myFunction() {
    let localVar = "I'm local";
    console.log(globalVar); // I'm global
    console.log(localVar); // I'm local
}
myFunction();
console.log(globalVar); // I'm global
console.log(localVar); // ReferenceError
```

클로저와 렉시컬 환경

외부함수보다 중첩함수가 더 오래 유지되는 경우

중첩함수는 이미 생명주기가 종료된 외부함수의 변수를 참조할 수 있다.

이러한 중첩함수를 클로저라고 부름

```
const x = 1;
function outer() {
    const x = 10;
    const inner = function () { console.log(x);};
    return inner;
}
const innerFunc = outer();
innerFunc();
```

클로저 활용 – 데이터 은닉

클로저는 상태(state)를 안전하게 변경하고 유지하기 위해 사용

상태가 의도치 않게 변경되지 않도록 상태를 안전하게 은닉하고

특정 함수에게만 상태 변경을 허용하기 위해 사용

함수가 호출할 때마다 호출된 횟수를 누적하여

출력하는 카운터

```
// 카운트 상태 변수
let num = 0;

// 카운트 상태 변경 함수
const increase = function () {
  // 카운트 상태를 1만큼 증가시킨다.
  return ++num;
}

console.log(increase());
console.log(increase());
console.log(increase());
```

클로저 활용 – 데이터 은닉

카운트 상태가 전역 변수라 언제든 누구나 접근할 수 있고 변경할 수 있음(암묵적 결합)

이는 의도치 않게 상태가 변경될 수 있음을 의미

만약 누군가에 의해 의도치 않게 카운트 상태, 즉 전역 변수 num 값이 변경되면 이는 오류로 이어짐

카운트 상태를 안전하게 유지하기 위해서는

increase 함수만이 num 변수를 참조하고 변경할 수 있게 해야함

전역 변수 num을 increase 함수의 지역변수로 바꾸어 보자

클로저 활용 – 데이터 은닉

Num을 지역변수로 바꿨더니 출력결과가 항상 1이다.

다시 말해, 상태가 변경되기 이전 상태를 유지하지 못한다.

이전 상태를 유지할 수 있도록 클로저를 사용해보자.

```
// 카운트 상태 변경 함수
const increase = function () {
  // 카운트 상태 변수
  let num = 0;

  // 카운트 상태를 1만큼 증가시킨다.
  return ++num;
}

console.log(increase());
console.log(increase());
console.log(increase());
```

클로저 활용 – 데이터 은닉

즉시 실행함수가 호출됨 ()

실행함수가 반환한 함수가 increase 변수에 할당됨

실행함수가 반환한 클로저는 자신이 정의된 위치에 의해
결정된 상위 스코프인 즉시 실행함수의 렉시컬 환경을 기억

즉시 실행함수가 반환한 클로저는 num 사용 가능해짐
상태가 의도치 않게 변경되지 않도록 안전하게 은닉됨

```
// 카운트 상태 변경 함수
const increase = (function () {
  // 카운트 상태 변수
  let num = 0;

  // 클로저
  return function () {
    // 카운트 상태를 1만큼 증가시킨다.
    return ++num;
  }
}())

console.log(increase());
console.log(increase());
console.log(increase());
```

클로저 활용 – 데이터 은닉

카운트 증가/감소

```
const counter = (function () {
  let num = 0;     // 카운트 상태 변수

  // 클로저인 매서드를 갖는 객체를 반환한다.
  // 객체 리터럴은 스코프를 만들지 않는다.
  // 따라서 아래 매서드들의 상위 스코프는 즉시 실행 함수의 렉시컬 환경이다
  return {
    increase() {
      return ++num;
    },
    decrease() {
      return num > 0 ? –num : 0;
    }
  };
}());
```

고차함수

고차함수 Higher-Order Function

함수의 형태로 리턴할 수 있는 함수

다른 함수(caller)의 인자로 전달되는 함수를 콜백함수(callback function)

콜백함수의 이름은, 어떤 작업이 완료되었을 때 호출하는 경우가 많아서. (일 끝나고 전화해 : 콜백)

콜백함수를 전달받은 고차함수는 함수 내부에서 이 콜백함수를 호출할 수 있음.

Caller는 조건에 따라 콜백함수의 실행 여부를 결정할 수 있음.

아예 호출하지 않을 수도 있고, 여러 번 실행할 수도 있음.

자바스크립트에서는 대표적으로 map, filter, reduce

Array.map()

하나의 데이터(배열)을 다른 데이터(배열)로 매핑할 때 사용

```
const cartoons = [
  {
    title: '식객',
    artist: '허영만',
    averageScore: 9.66,
  }
]; // 만화책의 모음

const subtitles = cartoons.map((cartoon) => cartoon.title); // 각 책의 제목 모음
```

Array.filter()

모든 배열의 요소 중에서 특정 조건을 만족하는 요소를 걸러내는 매소드

걸러내는 기준이 되는 조건은 filter 메소드의 인자로 전달(함수형태)

걸러내기 위한 조건을 명시한 함수를 인자로 받는 고차함수

```
const numbers = [1, 2, 3, 4, 5];
const evens = numbers.filter(num => num % 2 === 0);
console.log(evens); // [2, 4]
```

Array.reduce()

배열을 하나의 데이터로 응축할 때 사용

```
const cartoons = [
  { title: '식객', artist: '허영만', averageScore: 9.66 }
]; // 만화책의 모음

const subtitles = cartoons.reduce((acc, cartoon) => acc + cartoon.averageScore, 0); // 누적 평점
```

고차함수 조합

```
const numbers = [1, 2, 3, 4, 5];
const result = numbers
    .filter(num => num % 2 === 0)
    .map(num => num * 2)
    .reduce((acc, curr) => acc + curr, 0);
console.log(result); // 12 (2*2 + 4*2)
```

오늘 우리는

함수형 코드

렉시컬 스코프

변수의 접근 가능 범위를 정의하는 개념

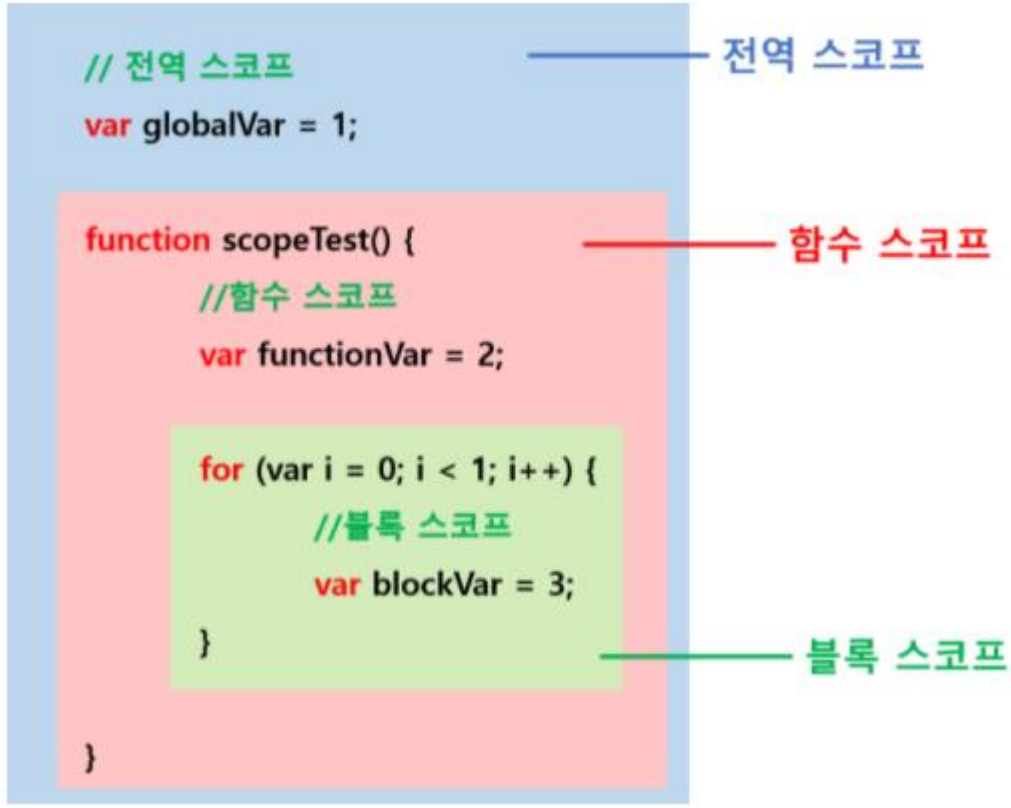
자바스크립트는 **렉시컬 스코프(정적 스코프)**를 사용

변수가 어디서 선언되었는지에 따라 접근 가능 여부가 결정

함수를 어디서 호출했는지가 아니라

함수를 어디에 정의했는지에 따라

상위 스코프를 결정



고차함수 조합

```
const numbers = [1, 2, 3, 4, 5];
const result = numbers
    .filter(num => num % 2 === 0)
    .map(num => num * 2)
    .reduce((acc, curr) => acc + curr, 0);
console.log(result); // 12 (2*2 + 4*2)
```

감사합니다