# JavaScript - Function

Sung-Dong Kim,
School of Computer Engineering,
Hansung University

# Contents


- Basics (기초)
- Parameters (매개 변수)
- Scope (영역)
- First-Class Function (일급 함수)
- Callback Function (콜백 함수)
- 함수 리턴하기
- Higher-order Function (고차함수)
- Composition Function (합성 함수)
- Curring (커링)

# Basics

# Basics

- Fundamental building block
- Declaration
  - function name(param1, param2, …) {body … return; };
- One function == one thing
- Naming: doSomething, verb, command
- Function is an object in JavaScript

# Basics

- 정의 방법
  - 함수 선언문: function square(x) { return x * x; }
  - 함수 리터럴: let square = function(x) { return x * x; };
  - Function 생성자: let square = new Function('x', 'return x * x;');
  - 화살표 함수: let square = x => x * x;

# Parameters

# Parameters

- Parameters
  - Primitive parameters: passed by value
  - Object parameters: passed by reference

```
function changeName(obj) {
    obj.name = 'coder';
}
const ellie = {name: 'ksd'};
changeName(ellie);
console.log(ellie);
```

# Parameters

Default Parameters

```
function showMessage(message, from) {
    console.log(`${message} by ${from}`);
}
function showMessage(message, from) {
    if (from == undefined) {
        from = 'unknown';
    }
    console.log(`${message} by ${from}`);
}
function showMessage(message, from='unknown') {
    console.log(`${message} by ${from}`);
}

showMessage('Hi!');
```

# Parameters

Rest Parameters

```
function printAll(...args) {
    for (let i=0; i<args.length; i++) {
        console.log(args[i]);
    }
    for (const arg of args) {
        console.log(arg);
    }
    args.forEach((arg) => console.log(arg));
}
printAll('0', 'dream', "coding", "KSD", 'sung-dong');
```

# Scope

# Scope

- Scope
  - 안에서는 밖을 볼 수 있음 – 밖에서는 안을 볼 수 없음
  - Closure: 중첩된 함수에서 자식 함수가 부모 함수에 정의된 변수를 접근할 수 있는 것

```
let globalMessage = 'global';
function printMessage() {
    let message = 'Hello';
    console.log(message);
    console.log(globalMessage);
    function printAnother() {
        console.log(message);
        let childMessage = 'child';
        console.log(childMessage);
    }
    printAnother();
}
printMessage();
```

# First-Class Function

# First-Class Function

- 일급 함수 (일급 객체): function은 variable과 같이 다룰 수 있음
  - 변수, 프로퍼티 값
  - 함수의 인자
  - 함수의 반환값
  - 함수형 프로그래밍

```
const print = function() {
    console.log('print')
}
print();
const printAgain = print;
printAgain();
```

```
function plus(a, b) {
    return a + b;
}
function minus(a, b) {
    return a - b;
}

let p = plus;
console.log("10 + 20 = %d", p(10, 20));
```

```
//함수를 parameter로 받는 함수
function calculate(a, b, func){
    return func(a, b);
}

//함수를 넘겨서 계산함
console.log(calculate(10, 20, minus));
console.log(calculate(10, 20, plus));
```

# First-Class Function

- 함수의 프로퍼티
  - caller - 호출 함수
  - length - 인자 개수
  - name
  - prototype - 프로토타입 객체의 참조
  - apply()/call()
    - 함수 실행: this 값과 함수 인자 이용
  - bind(): 객체에 함수를 바인드

```
function say(greetings, honorifics) {
    console.log(greetings + " " + honorifics + this.name);
}
let sung = { name: 'Sung Kim' };
let jace = { name: 'Jaehyun Kim'};
say.apply(sung, ["hello!", "Mr. "]);
say.apply(jace, ["hi!", "Boy, "]);
say.call(sung, "hello!", "Mr. ");
say.call(jace, "hi!", "Boy, ");
```

```
let saySung = say.bind(sung);
saySung('Hello!', "Mr. ");
```

# Callback Function

# Callback Function

- Callback function
  - callback = 함수를 전달해서 상황에 따라 적절한 함수를 호출하도록 하는 것
  - 전달된 함수들 = callback functions
- 예
  - addEventListner()에 전달되는 함수 (이벤트 처리기)
  - 타이머 함수에 전달되는 함수: setInterval(f(), tm); setTimeout(f(), tm);

```
const printYes = function() {
    console.log('yes');
};
const printNo = function print() {
    console.log('no!');
};
```

```
function randomQuiz(answer, printYes, printNo) {
    if (answer === 'love you') {
        printYes();
    } else {
        printNo();
    }
}

randomQuiz('wrong', printYes, printNo);
randomQuiz('love you', printYes, printNo);
```

```
const sum = (a, b) => a + b;
const printResult = (result) => {
  console.log(`결과는 ${result} 입니다.`);
};
 const calculationAndPrint = (calculationResult, callback) => {
  callback(calculationResult);
};
calculationAndPrint(sum(10, 20), printResult);
```

# 함수 리턴하기

# 함수 리턴하기

```
const returnFunction = () => (a, b) => a + b;

const plus = returnFunction();
                        // plus = (a, b) => a + b;

console.log(plus(10, 20));
```

# 함수 리턴하기

- returnFunction: 함수를 리턴하는 함수
  - 인자를 받지않고 호출됨
  - 인자 2개를 받아 합을 리턴하는 함수를 리턴
- 커링(curring): 화살표를 2번 이상 사용하는 방법
- returnFunction() → 내부적으로 2개의 인자를 가지는 함수 == plus

# 고차함수

# 고차 함수 (higher-order function)

- 함수를 인자로 받거나 함수를 반환하는 함수
- 예: Array.map() 메소드 - 배열 내 모든 요소를 인자로 제공받는 callback 함수를 호출하여 새로운 배열을 만듬

# 고차 함수 (higher-order function)

```
const arr1 = [1, 2, 3];
const arr2 = [];
for(let i=0; i<arr1.length; i++) {
  arr2.push(arr1[i] * 2);
}

// prints [2, 4, 6]
console.log(arr2);
```

```
const arr1 = [1, 2, 3];
const arr2 = arr1.map(function(item) {
  return item * 2;
});
console.log(arr2);

const arr3 = arr1.map(item => item * 2);
console.log(arr3);
```

# 고차 함수 (higher-order function)

```
const birthYear = [1975, 1997, 2002,
1995, 1985];
const ages = [];
for(let i=0; i<birthYear.length; i++){
  let age = 2023 - birthYear[i];
  ages.push(age);
}
console.log(ages);
```

```
const birthYear = [1975, 1997, 2002, 1995, 1985];
const ages = birthYear.map(year => 2023 - year);

console.log(ages);
```

# 합성함수

# 합성 함수 (composition function)

- 함수들을 조합하여 새로운 함수를 만드는 것

```
const multiple5 = x => x * 5;

const add10 = x => x + 10;

 const plus = (a, b) => a + b;

const minus = (a, b) => a - b;


console.log(multiple5(add10(20)));          // (20 + 10) * 5

console.log(minus(plus(10, 20), 40));       // (10 + 20) - 40
```

# 합성 함수 (composition function)

- 기존 함수를 재활용

```
const add2AndSquare = (num) => {
        num = num+2;
        num = num*num;
        return num;
}
add2AndSquare(3);
```

```
const add2 = (num) => num+2;
const square = (num) => num*num;
const add2AndSquare = (num) => sqaure(add2(num));
```

# 합성 함수 (composition function)

```
const compose = (func1, func2) => val => func2(func1(val));

const add2_mul3 = compose(add2, multiply3);   //  = multiply3(add2(val))

add2_mul3(5);
```

# 커링

# 커링 (curring)

- 인자를 여러 개 받는 함수를 분리하여, 인자가 하나인 함수의 체인(중첩 함수)으로 만드는 방법
- 함수의 전달인자 몇 개를 미리 채워 더 간단한 함수를 만드는 방법
- 함수를 재사용하는데 유용하게 쓰일 수 있는 기법
- 매개 변수가 항상 비슷할 때 유용
  - 외부 함수 인자는 변하지 않는 것
  - 내부 함수 인자는 변하는 것

# 커링 (curring)

```
function greet(greeting, name) {
        console.log(greeting + ", " + name);
}
```

```
greet('hello', 'sung-dong');
greet('hello', 'world');
```

```
function greet(greeting) {
        return function(name) {
            console.log(greeting + ", " + name);
        }
}
```

```
let hello = greet('hello');
hello('sung-dong');
hello('world');
```

# 커링 (curring)

- 2개 이상의 화살표로 함수를 정의하는 것
- 화살표 개수만큼 인자(파라미터)를 전달해야 함
- 화살표 개수보다 적은 개수의 인자를 전달 받으면 함수를 리턴
- add(): 인자를 두 개 받아서 합을 리턴하는 함수
- add10(): 인자를 하나 받아서 10을 더한 값을 리턴하는 함수

# 커링 (curring)

```
const add = x => y => x + y;

const add10 = add(10);


console.log(add10(20));

console.log(add(10)(20));
```

# 커링 예 1

```
const coffeeMachine = liquid => espresso => `${espresso}+${liquid}`;
const americanoMachine = coffeeMachine('water');
const latteMachine = coffeeMachine('milk');
const americano = americanoMachine('coffee bean');
const latte = latteMachine('coffee bean');


console.log(americano);
console.log(latte);
```

# 커링 예 2

```
const fs = require('fs');
const openFileAndPrint = path => fileName => fs.readFile(path + fileName, (err, data) => {
  if (err) throw err;
  console.log(data.toString());
});

const thisDirOpenFileAndPrint = openFileAndPrint('./');
const parentDirOpenFileAndPrint = openFileAndPrint('../');
thisDirOpenFileAndPrint('104_curring_example.js');
parentDirOpenFileAndPrint('package.json');
```

# 재귀 함수: 1~n 더하기

```
const sumNumber = (start, end, accumulator) => {
  if (start > end)
      return accumulator;
  return sumNumber(start + 1, end, accumulator + start);
};

console.log('result:', sumNumber(1, 10, 0));
```