12장 함수

정의

- 수학적으로는 input 을 받아 output을 내보내는 일련의 과정
- 프로그래밍으로는 일련의 과정을 statement로 구현하고 코드 블록으로 감싸서 하나의 실행 단위로 정의한 것
- 구성요소
    - parameter - 매개변수
    - argument - 인수
    - return value - 반환값

💡 아래의 빈칸과 서로 차이점을 서술하시오.

```
function add(x, y) { // x,y 을 뭐라하는가 _________
  return x + y;
}
add(2,5); // 들어가는 값에 대한 단어 : ______
```

함수를 왜 사용하는가?

- 코드의 중복을 억제하고 재사용성을 높이는 함수는 유지보수의 편의성을 높임
- 실수를 줄여 코드의 신뢰성을 높이는 효과
- 함수는 객체 → 이름 명명 가능
  - 적절한 이름은 함수 내부의 코드를 이해하지 않고도 함수의 역할을 파악할 수 있게 도움
  - 위 이유는 코드의 가독성 향상

💡 본인이 생각하기에 이상적인 개발자는 어떤 형태인가?
해당 질문에 대해서 진지하게 한번쯤은 고민할 만 합니다. 보통 프로그래머로 취업을 한다면 인터뷰에서 무조건 물어보는 내용이기도 합니다.

함수 정의

함수 정의 방식	예시
함수 선언문	function add (x,y){ return x + y; }
함수 표현식	var add = function (x,y) { return x + y; };
Function 생성자 함수	var add = new Function('x', 'y', 'return x + y');
화살표 함수(ES6)	var add = (x, y) => x + y;

함수 선언문

```
function add(x, y) {
    return x + y;
```

```
}

//하단의 코드는 개발자 도구와 Nodejs가 다르게 나올수있음
console.dir(add); // f add(x, y)

console.log(add(2,5)); // 7
```

- 함수 선언문은 함수 리터럴과 형태가 동일함
- 하지만 리터럴은 함수 이름을 생략할 수 있으나 함수 선언문은 함수 이름을 생략할 수 없음
    - 함수 선언문은 표현식이 아닌 문임

선언문에서는 함수 이름을 생략할 수 없다. 만약 함수 이름을 생략하면 나오는 에러는 어떤건지 확인해보세요.

```
// 함수 선언문은 표현식이 아닌 문으로 변수에 할당할 수 없음
// 하지만 함수 선언문이 변수에 할당되는 것처럼 보임
var add = function (x, y) {
    return x + y;
}

console.log(add(2,5)); // 7
```

- 해당 부분이 동작하는 이유는 JS엔진이 코드의 문맥에 따라 동일한 함수 릩럴을 표현식이 아닌 문인 함수 선언문으로 해석하는 경우와 표현식인 문인 함수 리터럴 표현식으로 해석하는 경우가 있기 때문임
- 선언문은 함수 이름을 생략할 수 없다는 점을 제외하면 함수 리터럴과 형태가 동일함
    - 함수 이름이 있는 기명 함수 리터럴은 함수 선언문 또는 함수 리터럴 표현식으로 해석될 가능성이 존재함을 의미

{ } 는 블록문일까 객체 리터럴일까? 본인의 생각을 쓰고 그 이유에 대해서 서술하시오.

함수 표현식

- 값의 성질을 갖는 객체를 일급객체라함

    - JS에서 힘수는 일급 객체에 속함
- 함수는 일급 객체이므로 함수 릩럴로 생성한 함수 객체를 변수에 할당할 수 있음
    - 해당 표현 방식을 함수 표현식이라고 함

```
var add = function(x, y) {
  return x + y;
}

console.log(add(2, 5));
```

함수 생성 시점과 함수 호이스팅

```
//함수 참조
console.log(add); // f add(x, y)
console.log(sub); // undefined

//함수 호출
console.log(add(2, 5)); // 7
console.log(sub(2, 5)); // TypeError : sub is not a function

//함수 선언문
function add(x, y){
  return x + y;
}

// 함수 표현식
var sub = function (x ,y) {
  return x + y;
}
```

- 함수 선언문으로 정의할 함수와 함수 표현식으로 정의한 함수의 생성 시점이 다르기 때문임
- 모든 선언문이 그렇듯 함수 선언문도 코드가 한 줄씩 순차적으로 실행되는 시점인 런타임 이전에 JS엔진에 의해 먼저 실행됨
    - 함수 선언문으로 함수를 정의하면 런타임 이전에 함수 객체가 먼저 생성됨 그리고 JS엔진은 함수 이름과 동일한 이름의 식별자를 암묵적으로 생성하고 생성된 함수객체를 할당
- 함수 선언문이 코드의 선두로 끌어 올려진 것처럼 동작하는 JS 고유의 특징을 함수 호이스팅이라 함

함수 호이스팅과 변수 호이스팅

- var 키워드를 사용한 변수 선언문과 함수 선언문은 런타임 이전에 JS 엔진에 의해 먼저 실행되어 식별자를 생성한다는 점이 동일
  - 변수 호이스팅의 경우 var 키워드로 선언된 변수를 undefined로 초기화됨
  - 함수 호이스팅의 경우 함수 표현식의 함수 리터럴도 할당문이 실행되는 시점에 평가되어 함수 객체가 됨
    - 즉 함수 표현식으로 함수를 정의하면 함수 호이스팅이 발생하는 것이 아니라 변수 호이스팅이 발생함

Function 생성자 함수

- JS가 기본 제공하는 빌트인 함수인 Function 생성자 함수에 매개변수 목록과 함수 몸체를 문자열로 전달하면서 new 연산자와 함께 호출되면 함수 객체를 생성해서 반환
  - 사실 new 연산자 없이 호출해도 결과는 동일함

```
var add = new Function('x', 'y', 'return x + y');
console.log(add(2,5));
```

하단의 에러는 왜 날까?

```
var add1 = (function() {
  var a = 10;
  return function (x, y){
    return x + y + a;
  };
}());

console.log(add1(1,2)); // 13

var add2 = (function() {
  var a = 10;
  return new Function('x', 'y', 'return x + y + a;')
}());

console.log(add2(1,2)); // ReferenceError: a is not defined
```

화살표 함수

- ES6에서 도입된 화살표 함수는 function 키워드 대신 화살표를 사용해 함수를 선언

```
const add = (x, y) => x + y;
console.log(add(2,5));
```

- 생성자 함수로 사용할 수 없으며 기존 함수와 this 바인딩 방식이 다르고 prototype 프로퍼티가 없고 arguments 객체를 생성하지 않음

함수 호출

매개변수와 인수

```
function add(x, y){ // x, y는 매개변수
  return x + y;
}

var result = add(1, 2); // 1,2가 인수
```

아래 함수를 실행해보고 결과 값을 적으시오.

```
function add(x, y){
  console.log(x,y);
  return x+y;
}

add(2, 5);

console.log(x, y);
```

```
function add(x, y) {
  return x + y;
}

console.log(add(2));
```

```
function add(x, y) {
  console.log(arguments);
  return x + y;
}

console.log(add(2,5,10));
```

인수확인

```
function add(x, y){
  return x + y;
}

console.log(add(2));
console.log(add('a', 'b'));
```

- 해당 코드를 돌려보면 에러가 나지 않으며 JS 문법상 어떠한 문제도 없으며 JS엔진은 아무런 이의 제기없이 위 코드를 실행할 것임
- 왜저러는가?
  1. JS 함수는 매개변수와 인수의 개수가 일치하는지 확인하지 않음

2. JS는 동적 타입언어이며 JS 하뭇는 매개변수의 타입을 사전에 지정할 수 없음

- 그렇다면 인수 값을 검사하는 logic을 짜면 되지 않나?

```
// 타입 검사
function add(x, y) {
  if(typeof x !== 'number' || typeof y !== 'number'){
    throw new TypeError(`인수는 모두 숫자여야 합니다. ( -.-)`);
  }
  return x + y;
}

console.log(add(2));
console.log(add('a', 'b'));
```

```
// 단축평가를 이용한 것
function add(a, b, c){
  a = a || 0;
  b = b || 0;
  c = c || 0;
  return a + b + c;
}

console.log(add(1,2,3));
console.log(add(1,2));
console.log(add(1));
console.log(add());
```

```
function add(a = 0, b = 0, c = 0){
  return a + b + c;
}

console.log(add(1,2,3));
console.log(add(1,2));
console.log(add(1));
console.log(add());
```

매개변수의 최대 개수

- ECMAScript 사양에는 매개변수의 최대 개수에 대해 명시적으로 제한하고 있지 않음
    - 다만 물리적 한계는 있어서 JS엔진마다 매개변수의 최대 개수에 대한 제한이 있겠지만 충분히 많은 매개변수를 지정할 수 있음

- 매개변수는 순서에 의미가 있음
  - 매개 변수가 많아 지면 함수 호출 시 전달해야 하는 인수의 순서를 고려 해야 함
  - 함수의 사용성이 낮아지고 사용자로 하여금 이해하기 어렵게 만들며 실수 유발하기 쉬운 코드가 됨
  - 매개변수의 개수나 순서가 변경된다면 함수의 호출 방법이 바뀌고 사용하는 코드 전체를 수정해야 하는 대참사가 일어남
- 보통 함수를 설계할 때 한 가지 일에 특화된 함수를 설계하지 여러가지 일을 동시에 하는 함수를 설계 하지 않음
  - 함수는 한 가지 일만 해야 하며 가급적 작게 만들어야 함

반환문

- return 반환할 값; 형식으로 이루어짐
- 그렇다면 반환이 안되는 경우는 어떤 경우인가?

```
function a() {
  return; // return할 값이 없어서 undefined로 반환
}

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

```
function a() {
  // return 없으면 undefined로 반환
}

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

```
function a(x,y) {
  return //ASI 기능으로 자동 세미콜론 삽입
  x + y;
}

console.log(a(1,2));
```

- 반환문은 함수 몸체 내부에서만 수행 가능
  - 전역에서 하면? 가차없이 SyntaxError : Illegal return statement

Call by Reference, Call by Value

해당 단원은 여러분을 위해 비어드렸습니다. 반드시 공부해와 주세요. 문서 형태는 마음대로 지만 다만 본인이 공부했다는 티는 나셔야 합니다.

다양한 형태의 함수

즉시 실행 함수

- 함수 정의와 동시에 즉시 호출되는 함수이며 한번 호출 후 다시 호출할 수 없음
    - IIFE(Immediately Invoked Function Expression)

```
//익명 즉시 실행 함수
(function (){
var a = 3;
var b = 5;
return a * b;
}());
```

- 재호출 시 ReferenceError 발생

```
//기명 즉시 실행 함수
(function a(){
var a = 3;
var b = 5;
return a * b;
}());

a(); // ReferenceError: a is not defined
```

- (…) 그룹연산자로 감싸지 않으면 나오는 에러

```
function () { //SyntaxError: Function statements require a function name
  // ...
}();
```

```
function a() {
  // ...
}();  //SyntaxError: Unexpected token ')'
```

```
function foo() {} (); // function foo() {}; ();
(); //SyntaxError: Unexpected token ')'
```

재귀 함수

- 함수가 자기 자신을 호출하는 것을 재귀 호출이라 함

```
//기본함수
function countDownOrigin(n) {
  for(var i = n; i >= 0; i--) console.log(i);
}
countDown(10);

//재귀함수
function countDownRecursive(n) {
  if(n < 0) return;
  console.log(n);
  countDownRecursive(n-1);
}
countDownRecursive(10);
```

재귀함수로 팩토리얼을 구현해보시오. 그리고 해당 코드에 대한 리뷰를 해보세요.

- 재귀함수는 탈출조건이 없으면 무한 재귀 호출되며 탈출 조건이 없으면 함수가 무한 호출되어 스택 오버플로(stack overflow) 에러가 발생함

중첩 함수

- 다른 이름으로는 내부 함수라고 불림

```
function outer() {
  var x = 1;
  function inner() {
    var y = 2;
    console.log(x + y);
  }
  inner();
}

outer();
```

콜백 함수

우선 어떤 task를 반복하여 수행하는 함수가 존재한다 생각해보자

```
function replay(n) {
  for(var i = 0; i < n; i++) console.log(i);
}

replay(5);
```

- 여기서 console.log의 의존성이 강해서 예를 들어 조건문 기능을 추가한다면 무조건 코드를 새로 지어야 함

```
function replay1(n) {
  for(var i = 0; i < n; i++) console.log(i);
}

replay1(5);

//추가된 함수
function replay2(n) {
  for(var i = 0; i < n; i++){
    if(i % 2) console.log(i);
  }
}

replay2(5);
```

```
//외부에서 전달받은 f를 n만큼 반복 호출
function replay(n,f) {
  for(var i = 0; i < n; i++) f(i); // i를 전달하면서 f를 호출
}

var logAll = function(i) {
  console.log(i);
};

// 반복호출할 함수를 인수로 전달
replay(5, logAll);


var logOdd = function(i) {
  if(i % 2) console.log(i);
};

// 반복호출할 함수를 인수로 전달
replay(5, logOdd);
```

- 여기서 함수 *f* 로 추상화 했으며 해당 값은 외부에서 전달받음

- JS 함수는 일급 객체이므로 함수의 매개변수를 통해 함수를 전달할 수 있음
- 함수의 매개변수를 통해 다른 함수의 내부로 전달되는 함수를 callback function이라 하며 매개변수를 통해 함수의 외부에서 콜백 함수를 전달받은 함수를 Higher-Order Function(HOF)라고 지칭함
    - 고차 함수는 콜백 함수를 자신의 일부분으로 합성함
    - 고차 함수는 매개변수를 통해 콜백 함수의 호출 시점을 결정해서 호출함
    - 콜백 함수는 고차 함수에 의해 호출되며 이때 고차함수는 필요에 따라 콜백 함수에 인수를 전달할 수 있음
    - 고차 함수에 콜백 함수를 전달할 때 콜백 함수를 호출하지 않고 함수 자체를 전달해야 함

callback 지옥이라는 말이 유명하다. 직접 지옥을 만들어보자.
그리고 calllback 지옥이 왜 위험한지 서술하시오.

순수 함수, 비순수 함수

- 함수형 프로그래밍에서는 어떤 외부 상태에 의존하지 않고 변경하지도 않는 부수 효과가 없는 함수를 순수 함수라고함
    - 동일한 인수가 전달되면 언제나 동일한 값을 반환함
    - 어떤 외부 상태에도 의존하지 않고 오직 매개변수를 통해 함수 내부로 전달된 인수에게만 의존해 값을 생성함
        - 외부상태란?

          전역 변수, 서버 데이터, 파일, console, DOM 등이 존재함
    - 최소 하나 이상의 인수를 전달받으며 인수를 전달받지 않은 순수함수는 언제나 동일한 값을 반환하므로 결국 상수와 같음
        - 이는 순수 함수는 불변성을 유지한다고 볼 수 있음
- 그럼 부수 효과가 있는 함수는 비순수 함수라고 지칭함
    - 외부 상태에 따라 반환값이 달라지는 함수이며 외부 상태에 의존하는 함수

아래의 코드 중 어떤 것이 순수 함수이며 어떤 것이 비순수 함수인지 서술하시오.

```
var count = 0;

function increase(n) {
  return ++n;
}

count = increase(count);
console.log(count);

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

```
var count = 0;

function increase() {
  return ++count;
}

count = increase(count);
console.log(count);

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

- 함수형 프로그래밍은 순수 함수와 보조 함수의 조합을 통해 외부 상태를 변경하는 부수 효과를 최소화해서 불변성을 지향하는 프로그래밍 패러다임
  - 대개 logic 내에 존재하는 조건문과 반복문을 제거해서 복잡성을 해결하며 변수 사용을 억제하거나 생명 주기를 최소화해서 상태 변경을 피해 오류를 최소화하는 것을 목표로 잡음
  - 조건문이나 반복문은 logic flow를 이해하기 어렵게 해서 가독성을 해치며 변수의 값은 누군가에 의해 언제든지 변경될 수 있어 오류 발생의 근본적인 원인이라고 봄
  - 결론적으로 함수형 프로그래밍은 순수 함수를 통해 부수 효과를 최대한 억제해 오류를 피하고 프로그램의 안정성을 높이려는 노력의 일환이라 할 수 잇음
- JS 는 멀티 패러다임 언어여서 OOP와 FP를 적극적으로 활용함