함수

함수란?

> 함수는 **일련의 동작**에 이름을 붙이고 **실행시키고 싶을 때 호출**해서 사용하는 것

> **세탁 → 헹굼 → 탈수** 를 예시로 처리하고 싶은 것을 함수로 정리해 두었다가 호출만 하면 바로 실행 시키는 것

javascript 언어에서 함수는, C 언어의 함수와 유사하다.

javascript 함수의 독특한 부분은, 함수가 자료형이라는 점이다.
C 언어의 함수나 Java의 메소드는, 실행 가능한 코드일 뿐, 자료형은 아니다.

함수가 자료형이라는 말은, 함수가 어떤 값이라는 말이다.
그래서 함수를 변수에 대입할 수도 있고,
함수를 파라미터 값으로 전달할 수도 있고,
함수를 리턴 값으로 리턴할 수도 있다.

함수 정의

```
function hello(name) {
    console.log("hello " + name);
}
```

위 함수의 이름은 hello 이다.
파라미터 변수는 name 이다.
javascript 언어는 약타입 언어이므로, 파라미터 변수의 타입을 선언하지 않고,
파라미터 변수 이름만 선언한다.

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

위 함수의 이름은 add 이다.
파라미터 변수는 각각 a, b 이다.
add 함수의 리턴 값은 number이지만, 리턴 타입도 선언하지 않는다.

함수 호출

함수 호출 #1

```
function hello(name) {
    console.log("hello " + name);
}

hello("홍길동");
```

함수를 호출하는 방법은 C 언어와 같다.

```
"hello 홍길동"
```

함수 호출 #2

```
hello("홍길동");

function hello(name) {
    console.log("hello " + name);
}
```

```
"hello 홍길동"
```

함수 호출이 위에 있고, 함수 정의가 아래 있어도 된다.

함수 호출 #3

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

console.log(add(3, 4));
console.log(add('hello', "world"))
```

```
7
"helloworld"
```

파라미터 값 생략 가능

예 #1

```
function hello(name) {
    console.log("hello " + name);
}

hello();  //hello undefined
```

hello 함수를 호출하면서 파라미터 값을 전달하지 않았다.
값이 전달되지 않은 파라미터 변수의 값은 undefined 이다.

예 #2

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

```
console.log(add(3, 4));  //7
console.log(add(3));     //NaN
```

파라미터 값이 한 개만 전달되기 때문에,
파라미터 변수 a의 값은 3 이고, 파라미터 변수 b의 값은 undefined 가 된다.
3 + undefined 표현식의 값은, 계산할 수 없으므로, NaN 이다.

연습 문제

> 2개의 양수를 파라미터 a와 b로 전달받아 a가 b의 배수면 'a는 b의 배수 입니다'를 출력하고, 배수가 아니면 'a는 b의 배수가 아닙니다'를 출력하는 multipleCheck(a, b) 함수의 정의입니다.
>
> 다음 A와 B를 채워 보세요

```
function multipleCheck(a, b){
    if(  A  ){
        console.log('양수를 입력하세요.');
        (  B  );
    }
    if(a%b == 0)  console.log('a는 b의 배수입니다.');
    else console.log('a는 b의 배수가 아닙니다.');
}
var a = -10, b = 2;
multipleCheck(a,b);
a = 10, b = 2;
multipleCheck(a,b);
```

답

파라미터 값이 전달되었는지 확인

예 #1 -

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

console.log(add(3, 4));  //7
console.log(add(3));     //3
```

`if (b == undefined) b = 0;`

두 번째 파라미터가 전달되지 않은 경우에, b 변수 값은 undefined 이고,
이 if 문이 true 가 된다.

예 #2

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

console.log(add(3, 4));  //7
console.log(add(3));     //3
```

변수 b 값이 undefined 이면,

if (b) 조건식은 false 이고, **if (!b)** 조건식은 true 이다.

if (b == undefined) 대신에 **if (!b)** 를 사용할 수 있다.

예 #5

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

```
console.log(add(3, 4)); //7
console.log(add(3));    //3
```

(b || 0) 표현식의 값
b 값이 숫자이면, 이 표현식의 값은 b 값이 된다.
b 값이 undefined 이면, 이 표현식의 값은 0 이다.

가변 파라미터 ES6 문법

```
function sum(...a) {
    let result = 0;
    for (let i = 0; i < a.length; ++i)
        result += a[i];
    return result;
}

console.log(sum(1, 2, 3, 4));
console.log(sum(4));
console.log(sum());
```

...a
전달된 파라미터 값들을 모아서 배열을 만들어서 a 변수에 대입해라
a 파라미터 변수에 대입된 값은 배열이므로, 배열처럼 사용하면 된다.

...a 부분의 문법의 이름은 spread syntax 이다.

sum(1, 2, 3, 4)
이렇게 호출할 때, a 파라미터 변수에 대입되는 값은 [1, 2, 3, 4] 이다.

sum(4)
이렇게 호출할 때, a 파라미터 변수에 대입되는 값은 [4] 이다.

sum()
이렇게 호출할 때, a 파라미터 변수에 대입되는 값은 [] 이다.

가변 파라미터 옛날 문법

```
function sum() {
    let result = 0;
    for (let i = 0; i < arguments.length; ++i)
        result += arguments[i];
    return result;
}

console.log(sum(1, 2, 3, 4)); //10
console.log(sum(4));          //4
console.log(sum());           //0
```

ES6 이전 옛날 문법에서는, arguments 키워드를 이용해서 구현해야 한다.
arguments 키워드는 모든 함수에서 사용할 수 있다.
arguments 키워드의 값은, 함수를 호출할 때 전달된 파라미터 값 목록이 배열로 모아진 것.

콜백 함수

함수 자료형

javascript 언어에서 함수는 자료형(data type) 중 하나이다.
즉 함수도 자료(data) 이고, 값이다.

```
function1.js
function add(a, b) {
  return a + b;
}

let a = add(3, 4);
console.log(a); //7

let f = add;
console.log(typeof f); //function
```

```
let b = f(3, 4);
console.log(b); //7
```

`let a = add(3, 4);`
변수 a 에는 add(3, 4) 함수의 리턴값 7이 대입된다.

`let f = add;`
변수 f 에 대입되는 값은 add 함수이다.

`typeof f`
변수 f 에 대입된 값은 함수이고, 그 값의 자료형은 function 이다.

`let b = f(3, 4);`
변수 f의 값인 함수를 호출한다. 즉 add 함수가 호출된다.
이 함수의 리턴값 7이 변수 b에 대입된다.

```
let f = function(a, b) {
  return a + b;
}

console.log(typeof f); //function

let a = f(3, 4);
console.log(a); //7
```

`let f = function(a, b) {`
`  return a + b;`
`}`

변수 f 에 대입되는 값은, 노란색으로 칠한 부분이다. 즉 함수가 변수 f 에 대입된다.

`typeof f`
변수 f 에 대입된 값은 함수이고, 그 값의 자료형은 function 이다.

`let b = f(3, 4);`
변수 f의 값인 함수를 호출한다. 즉 add 함수가 호출된다.
이 함수의 리턴값 7이 변수 b에 대입된다.

동일한 코드

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

let f = add;
```

위 코드와 아래 코드는 동일하다.
변수 f에 함수가 대입된다.

```
let f = function(a, b) {
  return a + b;
}
```

동일한 코드

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

위 코드와 아래 코드는 동일하다.
변수 add에 함수가 대입된다.

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

콜백 함수 전달(숫자는 코드 실행 순서)

```
//1 test1 함수 생성
function test1(f) { //5 test1함수가 호출된다. 매개변수 f에 add함
수가 전달된다.
                                        //9 test1 함수가 호출
```

```
된다. 매개변수 f에 multiply함수가 전달된다
  let result = f(3, 4); //6 add함수가 호출되고, 그 리턴 값이 res
ult에 대입된다.
                                                    //10 multip
ly함수가 호출되고, 그 리턴 값이 result에 대입된다
  console.log(result); //7 콘솔에 값이 출력 (7)
                                                  //11 콘솔에 값이
출력 (12) - 끝
}

//2 add 함수 생성
function add(a, b) {
  return a + b;
}

//3 multiply 함수 생성
function multiply(a, b) {
  return a * b;
}

test1(add); //4 test1 함수를 호출한다, 이때 파라미터 값으로 add 함
수가 전달된다.
test1(multiply); //8 test1 함수를 호출한다, 이때 파라미터 값으로 m
ultiply 함수가 전달된다.
```

콜백 함수란?

콜백 함수(callback function)는 다른 함수의 파라미터 값으로 전달되어 호출되는 함수이다.

add 함수가 test1 함수의 파라미터(매개변수) 값으로 전달되어, test1 함수 내부에서 호출되었다.
multiply 함수가 test1 함수의 파라미터(매개변수) 값으로 전달되어, test1 함수 내부에서 호출되었다.

여기서 add 함수와 multiply 함수가 콜백 함수이다.

아래 코드는 위 코드와 동일함

```
function test1(f) {
  let result = f(3, 4);
  console.log(result);
}

const add = function(a, b) {
  return a + b;
}

const multiply = function(a, b) {
  return a * b;
}

test1(add);
test1(multiply);
```

아래 코드는 위 코드와 동일함

```
function test1(f) {
  let result = f(3, 4);
  console.log(result);
}

test1(function(a, b) {
  return a + b;
});

test1(function(a, b) {
  return a * b;
});
```

화살표 함수

화살표 함수는 콜백 함수를 좀 더 간결하게 구현하기 위한 문법다.

```
arrow1.js
function test1(f) {
  let result = f(3, 4);
  console.log(result);
}

const add = (a, b) => {
  return a + b;
}

const multiply = (a, b) => {
  return a * b;
}

test1(add);
test1(multiply);
```

위 콜백함수 예제를 위와 같이 구현할 수 있다.
실행 결과는 동일하다.

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

위 코드와 아래 코드는 동일한 함수를 구현한다.

```
(a, b) => {
  return a + b;
}
```

타이머

setTimeout 함수

```
let id = setTimeout(callback, delay, [arg1], [arg2], ...)
```

delay

밀리초 후에 callback 함수가 호출된다.

arg1, arg2 ...

callback 함수를 호출할 때 파라미터 값들이다.

리턴값 :

등록된 callback 함수의 id 이다.
callback 함수 등록을 취소할 때, 이 id 값을 사용한다.

```
let id = setTimeout(...생략...);
clearTimeout(id);                    // 취소
```

```
function printTime(msg) {
  console.log(msg, new Date());
}

setTimeout(printTime, 1000, "1초 후"); //1초 후 2020-03-29T1
3:37:09.514Z
setTimeout(printTime, 2000, "2초 후"); //2초 후 2020-03-29T1
3:37:10.515Z
setTimeout(printTime, 3000, "3초 후"); //3초 후 2020-03-29T1
3:37:11.515Z
```

setInterval 함수

```
let id = setInterval(callback, delay, [arg1], [arg2], ...)
```

delay

밀리초 간격으로 callback 함수가 반복 호출된다.

arg1, arg2 ...

callback 함수를 호출할 때 파라미터 값들이다.

리턴값

등록된 callback 함수의 id 이다.
callback 함수 등록을 취소할 때, 이 id 값을 사용한다.

```
let id = setInterval(...생략...);
clearInterval(id);                    // 취소
```

```
function printTime(msg) {
  console.log(msg, new Date());
}

setInterval(printTime, 1000, "1초 간격"); //1초 간격으로 출력이
계속된다
```

```
setInterval(function (msg) {
  console.log(msg, new Date());
}, 1000, "1초 간격");
```

노란색으로 칠한 부분이 setInterval 함수의 첫째 파라미터 값이다.

```
setInterval((msg) => console.log(msg, new Date()), 1000, "1
초 간격");
```

노란색으로 칠한 부분이 setInterval 함수의 첫째 파라미터 값이다.

지역 함수

```
function outterFunc() {
  print("hello")

    function print(msg) {
    console.log(msg)
  }
}

outterFunc() //hello
// print("world")  여기서 호출할 수 없다.
```

outterFunc 함수 내부에 print 함수가 선언되었다.
print 함수는 outterFunc 함수 내부에서만 호출 할 수 있고, 외부에서는 호출할 수 없다.

```
function outterFunc() {

  function print(msg) {
    console.log(msg)
  }

  print("hello")
}

outterFunc()
```

outterFunc 함수 내부에 print 함수가 선언되었다.
print 함수 선언 위치와 print 함수 호출 위치의 순서는 상관없다.

이름 없는 함수

```
const f = function (msg) {
  console.log(msg)
}
```

```
f("hello")
```

노란색으로 칠한 부분은 함수이다.
그 함수를 변수 f에 대입한다.

하늘색으로 칠한 부분은 함수 호출을 위한 괄호이고, "hello" 문자열은 파라미터 값이다.
변수 f의 값인 함수이다. 이 함수가 호출된다.

위 코드에서 f 변수를 제거하면 아래의 코드가 된다.

```
(function (msg) {
  console.log(msg)
})("hello")
```

노란색으로 칠한 부분은 함수이다.
이 함수에는 이름이 없다.

하늘색으로 칠한 부분은 함수 호출을 위한 괄호이고, "hello" 문자열은 파라미터 값이다.
노란색으로 칠한 부분은 함수이다. 이 함수가 호출된다.

함수를 리턴

```
function factory() {
  return function (msg) {
    console.log(msg)
  }
}

const f = factory()
f("hello")
```

factory 함수의 리턴 값은 함수이다. (노란색으로 칠한 부분)
이 리턴값을 변수 f에 대입한다.

하늘색으로 칠한 부분은 함수 호출을 위한 괄호이고, "hello" 문자열은 파라미터 값이다.
변수 f의 값은 함수이다. 이 함수가 호출된다.

```
function factory() {
  return function (msg) {
    console.log(msg)
  }
}

factory()("hello")
```

하늘색으로 칠한 부분은 함수 호출을 위한 괄호이고, "hello" 문자열은 파라미터 값이다.
factory 함수의 리턴 값은 함수이다. 이 함수가 호출된다.

아래 코드는 화살표 함수 문법을 이용해서, 코드를 간결하게 조금씩 수정해 나가는 과정이다.
즉 아래 박스의 코드들이 조금씩 간결해질 뿐 동일한 코드이다.

리액트 공부가 어려운 이유는, 아래와 같은 고급 자바스크립트 문법이 예제 코드에도 들어있기 때문.

```
function factory() {
  return function (msg) {
    console.log(msg)
  }
}

factory()("hello")
```

factory 함수를 화살표 함수 문법으로

```
const factory = () => {
    return function (msg) {
        console.log(msg)
```

```
    }
}

factory()("hello")
```

리턴되는 함수도 화살표 함수 문법으로

```
const factory = () => {
    return (msg) => {
        console.log(msg)
    }
}

factory()("hello")
```

factory 화살표 함수 본문이 return 문 한 개이므로...

```
const factory = () =>
    (msg) => {
        console.log(msg)
    }

factory()("hello")
```

리턴되는 함수도 본문이...

```
const factory = () => (msg) => console.log(msg)

factory()("hello")
```

연습 문제

체온계 만들기

체온이 37.0 보다 크면 “고온” 이 출력 되고,

체온이 35.0 보다 작으면 “저온” 이 출력 되고,

체온이 35.0 ~ 37.0 면 “정상” 이 출력 되는 함수를 작성하세요.

답