JavaScript 101 - (4) 함수와 객체

1. 함수란 무엇인가?

함수(function)란 작업을 수행하거나 값을 계산하는 명령문의 집합입니다. 함수는 재사용 가능한 영역을 정의하고, 정보영역을 캡슐화하는데 사용되는 구문이기도 합니다. 일반적으로 함수는 입력(input)과 출력(output)을 갖습니다. 입력은 함수에 전달되는 값이고, 출력은 함수가 반환하는 값입니다.

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

이 함수는 `add`라는 이름을 가지고 있으며, `a`와 `b`라는 두 개의 입력(매개변수)을 받습니다. 이 함수는 입력된 두 값을 더한 결과를 반환합니다. 따라서 이 함수의 출력(반환값)은 `a + b`와 같습니다.

함수를 사용하려면 **호출(call)**해야 합니다. 호출이란 함수의 이름과 필요한 입력값을 명시하여 실행하는 것입니다.

예를 들어, 다음과 같이 `add`함수를 호출할 수 있습니다.

```
let result = add(3, 5); // result에 8이 저장됩니다.
```

2. 함수의 선언식과 표현식

JavaScript에서는 **함수 선언문(function declaration)**과 **함수 표현식(function expression)** 두 가지 방법으로 함수를 정의할 수 있습니다.

2-1. 함수 선언문

함수 선언문은 다음과 같은 형태로 작성합니다.

```
function functionName(parameter1, parameter2, ...) {
  // function body
}
```

- `function` 키워드로 시작합니다.
- 그 다음에 **함수 이름**을 씁니다. 보통 동사 형태로 지어줍니다.
- 그 다음에 괄호(`()`) 안에 **매개변수(parameter)**들을 쉼표(`,`)로 구분하여 씁니다. 매개변수는 입력값으로 전달되는 변수입니다.
- 그 다음에 중괄호(`{}`) 안에 **함수 본문(function body)**을 씁니다. 여기에 실행할 명령문들을 작성합니다.

예를 들어, 다음과 같이 `sayHello`라는 이름의 함수를 선언할 수 있습니다.

```
function sayHello(name) {
  console.log("Hello, " + name);
}
```

이 함수는 `name`이라는 매개변수 하나를 받아서 `"Hello, "`와 결합하여 콘솔에 출력합니다.

2-2. 함수 표현식

함수 표현식은 다음과 같은 형태로 작성합니다.

```
let functionName = function(parameter1, parameter2, ...) {
  // function body
};
```

- `let` 또는 `const` 키워드로 변수를 선언하고 그 변수에 **익명(anonymous)함수**를 할당합니다.

- 익명함수란 이름이 없는 함수입니다.
- 익명함수도 `function` 키워드로 시작하고 괄호 안에 매개변수들을 씁니다. 그 다음에 중괄호 안에 함수 본문을 씁니다.

예를 들어, 다음과 같이 `sayHello`라는 변수에 익명함수를 할당할 수 있습니다.

```
let sayHello = function(name) {
  console.log("Hello, " + name);
};
```

이 함수도 `name`이라는 매개변수 하나를 받아서 `"Hello, "`와 결합하여 콘솔에 출력합니다.

함수 표현식은 변수에 할당되기 때문에 **값(value)**으로 취급됩니다. 따라서 함수 표현식은 다른 값처럼 인자로 전달하거나 반환할 수 있습니다.

예를 들어, 다음과 같이 `sayHi`라는 함수가 `sayHello`함수를 인자로 받아서 실행하는 경우가 있습니다.

```
function sayHi(func) {
  func("World");
}

sayHi(sayHello); // Hello, World
```

2-3. 함수 선언문과 표현식의 차이점

함수 선언문과 표현식의 가장 큰 차이점은 **호이스팅(hoisting)**입니다. 호이스팅이란 자바스크립트 엔진이 코드를 해석할 때 선언된 변수나 함수를 코드 상단으로 끌어올리는 현상입니다.

함수 선언문은 전체가 호이스팅되기 때문에 정의하기 전에 호출할 수 있습니다.

예를 들어, 다음과 같은 코드가 정상적으로 작동합니다.

```
greet("World"); // Hello, World

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

하지만 함수 표현식은 변수만 호이스팅되고 값(익명함수)은 호이스팅되지 않기 때문에 정의하기 전에 호출하면 에러가 발생합니다.

예를 들어, 다음과 같은 코드는 에러가 발생합니다.

```
greet("World"); // TypeError: greet is not a function

let greet = function(name) {
  console.log("Hello, " + name);
};
```

함수와 객체

JavaScript에서 함수는 **일급 객체**입니다. 일급 객체란 다음과 같은 조건을 만족하는 객체를 말합니다.

- 무명의 리터럴로 생성할 수 있다.
- 변수나 자료구조에 저장할 수 있다.
- 함수의 인자로 전달할 수 있다.
- 함수의 반환값으로 사용할 수 있다.

예를 들어, 다음과 같은 코드가 가능합니다.

```
// 무명의 리터럴로 생성한 함수를 변수에 저장
const add = function(a, b) {
  return a + b;
};

// 배열의 요소로 사용
const arr = [add, function(c) { return c * c; }];

// 함수의 인자로 전달
function apply(func, x) {
```

```
  return func(x);
}

console.log(apply(arr[0], 3, 4)); // 7
console.log(apply(arr[1], 5)); // 25

// 함수의 반환값으로 사용
function makeFunc(n) {
  return function(x) {
    return x + n;
  };
}

const plusOne = makeFunc(1);
const plusTwo = makeFunc(2);

console.log(plusOne(10)); // 11
console.log(plusTwo(10)); // 12
```

스코프

스코프란 **변수에 접근할 수 있는 범위**를 말합니다. JavaScript에서는 **함수 스코프**와 **블록 스코프**가 있습니다.

함수 스코프

함수 스코프란 **함수 내에서 정의된 변수는 해당 함수 내에서만 사용할 수 있고 유효하다**는 것을 의미합니다. 함수 밖에서 정의된 변수는 전역 변수로서 어디서든 접근할 수 있습니다.

예를 들어, 다음과 같은 코드가 가능합니다.

```
var x = 'global'; // 전역 변수

function foo() {
  var x = 'local'; // 지역 변수
  console.log(x); // local
}

foo(); // local
console.log(x); // global
```

블록 스코프

블록 스코프란 **중괄호(`{}`)로 둘러싸인 영역 내에서 정의된 변수는 해당 영역 내에서만 사용할 수 있고 유효하다**는 것을 의미합니다. ES6 이전에는 블록 스코프가 존재하지 않았으나, ES6부터 `let` 과 `const` 키워드를 사용하여 블록 스코프를 만들 수 있습니다.

예를 들어, 다음과 같은 코드가 가능합니다.

```
let x = 'global'; // 전역 변수

if (true) {
  let x = 'local'; // 지역 변수
  console.log(x); // local
}

console.log(x); // global
```

클로저

클로저는 **함수와 함수가 선언된 어휘적 환경**의 조합입니다. 이 환경은 클로저가 생성된 시점의 유효 범위 내에 있는 모든 지역 변수로 구성됩니다.

예를 들어, 다음과 같은 코드가 가능합니다.

```
function makeCounter() {
  let count = 0; // 지역 변수

  return function() { // 클로저 (count를 기억함)
    return ++count;
  };
}

const counter1 = makeCounter(); // 카운터 생성
const counter2 = makeCounter(); // 다른 카운터 생성
```

```
console.log(counter1()); // 1 (counter1의 count 값)
console.log(counter1()); // 2 (counter1의 count 값 증가)
console.log(counter2()); // 1 (counter2의 count 값)
console.log(counter2()); // 2 (counter2의 count 값 증가)
```

고차함수

고차함수는 **함수를 인자로 받거나 반환하는 함수**입니다. 고차함수는 일급 객체인 함수를 활용하여 **추상화**하거나 **재사용**할 수 있습니다.

예를 들어, 다음과 같은 코드가 가능합니다.

```
// 배열을 받아 각 요소에 적용할 콜백함수와 초기값을 인자로 받아 배열을 축약하는 고차함수 reduce 정의
function reduce(array, callback, initialValue) {

  let accumulator = initialValue; // 누적값 초기화

  for (let i = 0; i < array.length; i++) {
    accumulator = callback(accumulator, array[i]); // 콜백함수 실행하여 누적값 갱신
  }

return accumulator; // 최종 누적값 반환

}

// reduce 고차함수 사용 예시

// 배열 요소들을 모두 더하는 콜백함수 정의
function sum(a, b) {
return a + b;
}

// 배열 요소들을 모두 곱하는 콜백함수 정의
function product(a,b) {
return a * b;
}

const numbers = [1,2,3,4];

console.log(reduce(numbers,sum ,0)); // 10 (1 + 2 + 3 + 4)
console.log(reduce(numbers, product, 1)); // 24 (1 * 2 * 3 * 4)
```