Chap12 함수

12-1 함수란?

▼ 함수란

- 일련의 과정을 문으로 구현하고 코드 블록으로 감사서 하나의 실행 단위로 정의한 것
  - ▼ 일련의 과정?

    입력을 받아 출력을 내보내는 일련의 과정

    ex) f(x.y)=x+y 이면 함수 내부로 입력을 받아들이는 변수 x,y에 2,5입력하면 일련의과저인 x+y다 깅랜죄너 7이라는 출력값을 반환한다.
- 함수는 함수 정의하는 부분과 함수 호출하는 부분으로 나뉜다.
- 함수 정의는 함수를 호출하기 전에 인수를 전달받을 매개변수와 실행할 문들, 그리고 반환할 값을 지정하는 것(정의 방법은 여러가지가 있다.)
- 함수 호출부분에는 인수를 매개변수를 통해 함수를 실행시킨다.

12-2 함수를 사용하는 이유

▼ 함수를 사용하는 이유

- 재사용할 수 있어 유지보수의 편의성을 높이고 실수를 줄여 코드의 신뢰성을 높인다.
  - ▼ why?

    같은 코드를 중복해서 여러번 작성해야하는 상황에서 함수를 사용하지 않으면 중복된 횟수만큼 코드를 수정해야 한다. 따라서 시간은 오래걸리고 실수할 확률도 높이진다.
- 코드의 가독성을 향상시킨다.
  - ▼ why?
    - 함수는 객체 타입의 값이기 때문에 식별자를 붙일 수 있다. 따라서 적절한 이름은 함수 내부코드를 이해하지 않고도 함수의 역할을 파악할 수 있게 한다.
    - 추가로 함수는 객체 타입의 값이기 때문에 식별자를 붙일 수 있다. 라는 부분에서 값이면 식발자를 붙일 수 있나? 라는 의문이 들 수도 있기에 설명한다. 식별자는 애초의 값의 이름이기 때문에 함수는 식별자를 붙일 수 있다!

12-3 함수 리터럴

▼ 함수 리터럴의 구성

> 💡 리터럴?
> 값을 생성하기 위한 표기법

function 키워드, 함수 이름, 매개 변수 목록, 함수 몸체

▼ 형태

```
//함수 리터럴 : 변수에 할당 된 형태. 함수이름은 생략가능(유무에 따라 기명함수/무명 or  익명함수)
var f= function add(x,y){
```

```
return x+y;
};
```

12-4 함수 정의

▼ 함수 정의

함수를 호출하기 전에 인수를 전달받을 매개변수와 실행할 문들, 그리고 반환할 값을 지정하는 것

▼ 함수를 정의하는 방법

- 함수 선언문
- 함수 표현식(함수 리터럴)
- Function 생성자 함수
- 화살표 함수(ES6)

▼ 함수 선언문

- 함수 리터럴과 형태는 동일하다. 단, 함수 이름 생략 불가능
- 표현식이 아닌 문이다.
  - 완료 값 undefined 출력
  - 변수에 할당 불가능(why? 표현식이 아니라는 것은 값을 나타낼 수 없음. 따라서 변수에 할당 불가능)

```
// 함수 선언문은 표현식이 아닌 문이므로 변수에 할당할 수 없다.
// 하지만 함수 선언문이 변수에 할당되는 것처럼 보인다.
var add = function add(x, y) {
  return x + y;
};

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

위 예제는 함수 선언문을 변수에 할당하는 것처럼 보이는 코드들인데 이는 함수 선언문이 아니라 함수 표현식(함수 리터럴)이다.

자바스크립트 엔진은 코드 문맥에 따라 다르게 해석하기도 하는데 (**중의적 표현**) 함수 리터럴도 이에 해당한다.

함수 리터럴 표현식의 중의적 표현

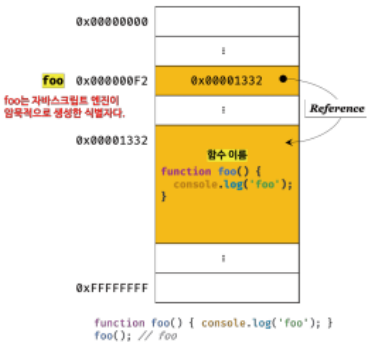
`기명 함수 리터럴` 을 **단독**으로 사용하면 **함수 선언문**으로 해석하고, **변수에 할당하거나 피연산자로 사용**하면 **함수 리터럴 표현식**으로 해석한다.

▼ 함수 선언문과 함수 표현식(함수 리터럴)의 차이

함수 표현식(함수 리터럴)에서 함수 이름은 함수 몸체 내에서만 참조할 수 있는 식별자 이기때문에 함수 몸체 외부에서 함수 이름을 호출할 수 없다. 호출하면 ReferenceError 발생.

함수 선언문으로 정의되는 함수는 몸체 외부에서 함수 이름을 호출할 수 있다.

why??



자바스크립트 엔진은 함수 선언문을 해석해서 함수 객체를 생성합니

이 때 함수 이름은 함수 몸체 내부에서만 유효한 식별자라서 외부에
름에 대한 참조가 불가능합니다.

근데 가능한 이유는 자바스크립트 엔진이 암묵적으로 함수 이름과 :
름의 **식별자**를 생성하고, 거기에 **함수 객체**를 할당하기 때문입니다.

자바스크립트 엔진이 함수 선언문으로 해석한 것은 정상작동

```
//기명 함수 리터럴을 자바스크립트 엔지은 함수 선언문이
function foo(){console.log('foo');}
foo(); //foo
```

자바스크립트 엔진이 함수 리터럴로 해석한 것은 에러발생

```
//자바스크립트 엔진은 이를 함수 표현식으로 해석
(function bar() {console.log('bar');});
bar(); //ReferenceError
```

따라서 함수 선언문인 함수는 함수 이름으로 호출을 해도 이는 함수 이름을 호출한 것이아니라 함수 객체를 가리키는 식별자를 호출한 것이기 때문에 오류가 안난다.

결론적으로 자바스크립트 엔진은 함수 선언문을 함수 표현식으로 변환해 함수 객체를 생성한다고 생각할 수 있다.

▼ 함수 표현식

- 일급 객체인 함수는 함수 리터럴로 생성한 함수 객체를 변수에 할당하는데 이러한 정의 방식을 함수 표현식이라고 한다.
- 함수 표현식의 특징
  - 함수 이름 생략 가능(익명 함수)
  - 함수 식별자로 호출

```
//기명 함수 표현식
var add=function foo(x,y){
 return c+y;
}

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

console.log(foo(2,5)); //ReferenceError: foo is not defined
```

▼ 함수 생성 시점과 함수 호이스팅

- 함수 선언문 - 함수 호이스팅 O

  ▼ 함수 선언문이 함수 호이스팅이 일어나는 이유

  var 키워드를 사용한 변수 선언문과 함수 선언문은 런타임 이전에 자바스크립트 엔진에 의해 먼저 실행되어 식별자를 생성한다. 하지만 var선언된 변수는 undefined로 초기화되지만 함수 선언문은 암묵적으로 생성된 식별자는 함수 객체로 초기화되면서 호이스팅이 일어난다.

  한마디로, **암묵적으로 자바스크립트 엔진이 함수 이름과 동일한 이름으로 식별자를 만들어 함수 객체를 참조하기 때문**이다.

- 함수 표현식(리터럴) - 변수 호이스팅 O

  ▼ 함수 표현식이 변수 호이스팅인 이유

  함수 표현식도 호이스팅이 일어난다. 단 변수에 할당되는 값이 함수 리터럴인 문일때.

var로 선언된 변수 할당문의 값은 실행되는 시점, 즉 런타임에 평가되므로 함수 표현식의 함수 리터럴도 할당문이 실행되는 시점에 평가되어 함수 객체가 된다. 따라서 함수 표현식으로 함수를 정의하면 변수 호이스팅이 일어난다.

한마디로, 함수 표현식은 변수에 할당 된 형태이기때문에 var로 선언한 변수일 때에는 런타임 이전에 형성되어 함수 표현식 이전에 참조되면 undefined로 평가된다. 즉 변수 호이스팅이 진행된다.

▼ 호이스팅

인터프리터가 코드를 실행하기 전(런타임 전)에 함수, 변수, 클래스 또는 임포트(import)의 선언문을 해당 범위의 맨 위로 끌어올리는 것처럼 보이는 현상

▼ Function 생성자 함수

빌트인 함수인 Function 생서자 함수에 매개변수 목록과 함수 몸체를 문자열로 전달하면서 new연산자와 함께 호출하여 함수 객체를 생성하는 방법. (new 연산자 없이 호출해도 결과는 동일)

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

▼ 화살표 함수

function 키워드 대신 화살표 ⇒ 를 사용하는 방법

```
const add=(x,y)=>x+y;
console.log(add(2,4)); //6
```

- 생성자 함수로 사용 불가능
- 기존 함수와 this 바인딩 방식 다름
- prototype 프로퍼티 없음
- arguments 객체 생성x

→ 여기나온 단어들에 대해선 뒤 챕터에서 자세히 다룰 예정

12-5 함수 호출

▼ 매개변수란(인자)?

매개변수(parameter)란 함수의 정의에서 전달받은 인수를 함수 내부로 전달하기 위해 사용하는 변수를 의미합니다.

▼ 인수란?

인수(argument)란 함수가 호출될 때 함수로 값을 전달해주는 값을 말합니다.

▼ 함수 호출

- 매개변수는 함수 몸체 내부에서만 참조 가능

```
function add(x, y) {
  console.log(x, y); // 2 5
  return x + y;
}

add(2, 5);

// add 함수의 매개변수 x, y는 함수 몸체 내부에서만 참조할 수 있다.
console.log(x, y); // ReferenceError: x is not defined
```

- 매개변수의 개수와 인수의 개수가 일치하는지 체크하지 않는다.

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

```
console.log(add(2)); // NaN = 2 + undefined
console.log(add(2,3,4)) // 5
```

초과되는 경우에는 버려지는 것이 아니라 arguments 객체의 프로퍼티에 보관된다.

```
function add(x, y) {
  console.log(arguments);
  // Arguments(3) [2, 5, 10, callee: ƒ, Symbol(Symbol.iterator): ƒ]

  return x + y;
}

add(2, 5, 10);
```

▼ 인수 확인

- 자바스크립트 함수는 매개변수와 인수의 개수가 일치하는지 확인하지 않는다.
- 자바스크립트는 동적 타입 언어다. 따라서 자바스크립트 함수는 매개변수의 타입을 사전에 지정할 수 없다.

위와 같은 이유로 함수를 정의할 때 적절한 인수가 전달되었는지 확인 필요

방법 1. if문을 이용하여 전달된 인수의 타입이 부적절한 경우 에러를 발생시키도록 한다.

방법 2. 정적 타입을 선언할 수 있는 타입스크립트를 이용한다.

▼ 매개변수의 최대 개수

- ECMAScript 사양에는 매개변수의 최대 개수에 대해 명시적으로 제한하지 않는다.
- 가장 이상적인 매개변수의 개수는 0개 이다.
- 매개변수는 순서에 의미가 있다.
- 개수가 너무 많으면 유지보수성이 나빠진다.
- 최대 3개 이상을 넘지 않는 것을 권장한다.
- 많은 매개변수가 필요하다면 객체를 활용하는 것이 유리하다.

▼ 반환문

return 키워드와 표현식으로 이뤄진 반환문을 사용해 실행결과를 함수 외부로 반환할 수 있다.

▼ 반환문의 역할

- 함수의 실행을 중단하고 함수 몸체를 빠져나간다.

```
function multiply(x, y) {
  return x * y; // 반환문
  // 반환문 이후에 다른 문이 존재하면 그 문은 실행되지 않고 무시된다.
  console.log('실행되지 않는다.');
}

console.log(multiply(3, 5)); // 15
```

- return 키워드 뒤에 오는 표현식을 평가해 반환한다.

```
function foo () {
  return;
}

console.log(foo()); // undefined
```

- 반환문 생략 가능

```
function foo () {
//반환문 생략하면 암묵적으로 undefined가 반환된다.
}

console.log(foo()); // undefined
```

- return 키워드와 반환 값 사이에 줄바꿈이 있으면 반환 값은 무시된다.

```
function multiply(x, y) {
  // return 키워드와 반환값 사이에 줄바꿈이 있으면
  return // 세미콜론 자동 삽입 기능(ASI)에 의해 세미콜론이 추가된다.
  x * y; // 무시된다.
}

console.log(multiply(3, 5)); // undefined
```

- 반환문은 함수 몸체 내부에서만 사용 가능

```
<!DOCTYPE html>
<html>
<body>
  <script>
    return; // SyntaxError: Illegal return statement
  </script>
</body>
</html>
```

12-6 참조에 의한 전달과 외부 상태의 변경

원시값은 값에 의한 전달, 객체는 참조에 의한 전달 방식으로 동작한다. 매개변수도 함수 몸체 내부에서 변수와 동일하게 취급되므로 매개변수 타입에 따라 원시값이 될 수 있고 객체가 될 수 있다.

```
// 매개변수 primitive는 원시값을 전달받고, 매개변수 obj는 객체를 전달받는다.
function changeVal(primitive, obj) {
  primitive += 100;
  obj.name = 'Kim';
}

// 외부 상태
var num = 100;
var person = { name: 'Lee' };

console.log(num); // 100
console.log(person); // {name: "Lee"}

// 원시값은 값 자체가 복사되어 전달되고 객체는 참조값이 복사되어 전달된다.
changeVal(num, person);

// 원시값은 원본이 훼손되지 않는다.
console.log(num); // 100
```

```
// 객체는 원본이 훼손된다.
console.log(person); // {name: "Kim"}
```

위 예제와 같이 `person` 객체를 매개변수로 전달하면 함수 내부에서 이를 수정할 경우 원본 객체가 변경되는 부수 효과(`side effect`)가 발생한다.

원시값은 값을 복사해 전달하기 때문에 영향 없음.

12-7 다양한 함수형태

▼ 즉시실행함수

함수의 정의와 동시에 즉시 호출되는 함수(`IIFE Immediately Invoked Function Expression`)

단 한 번만 호출되며 다시 호출할 수 없다.

- 함수 이름이 없는 익명 함수를 사용하는 것이 일반적이지만 기명 즉시 실행 함수도 사용 가능.
- 즉시 실행 함수는 반드시 그룹 연산자 `()`로 감싸야 한다.
- 그룹연산자`()`내의 기명함수는 함수 선언문이 아니라 함수 리터럴로 평가되며 함수 이름은 함수 몸체에서만 참조가능하여 호출할 수 없다.

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

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

▼ 그룹 연산자 `()` 로 감싸는 이유

그룹 연산자의 피연산자는 값으로 평가되기 때문에 기명 또는 무명 함수를 그룹 연산자를 감싸면 모두 함수 리터럴로 평가된다.

감싸지 않으면 함수 선언문이 되버리는데 이는 함수정의와 동시에 호출이 되지않기때문에 그룹 연산자 `()` 로 감싸서 표현식으로 만들어 사용하는 것이다. 그룹연산자 자체가 즉시실행함수의 형태라고 보면 된다.

▼ 정리

따라서 그룹 연산자를 사용하는 이유는 함수를 함수 리터럴로 평가하도록 하기 위해서이다. 함수 리터럴로 평가하면 변수에 할당할 수 있어 일반 함수처럼 값을 반환할 수 있고 인수를 전달할 수도 있다.

▼ 재귀 함수

- 함수가 자기 자신을 호출하는 것
- 팩토리얼 구현 가능
- 반복문 없이 구현할 수 있다. 그러나 무한 반복에 빠질 위험이 있고, 스택 오버플로 에러를 발생시킬 수 있으므로 주의해야한다.

▼ 스택 오버 플로우

반복적인 함수 호출로인해 메모리 스택이 넘치는 것

▼ 중첩 함수

- 함수 내부에 정의된 함수
- 중첩 함수(`nested function`)또는 내부 함수(`inner function`)라 한다. 그리고 중첩 함수를 포함하는 함수는 외부 함수(`outer function`)라 한다.

```
function outer() {
  var x = 1;

  // 중첩 함수
  function inner() {
```

```
    var y = 2;
    // 외부 함수의 변수를 참조할 수 있다.
    console.log(x + y); // 3
  }

  inner();
}

outer();
```

ES6부터는 함수 선언문을 `if` 문이나 `for` 문 등의 코드 블록 내에서도 정의할 수 있다.

단, 호이스팅으로 인해 혼란이 발생할 수 있으므로 바람직하지 않다. 중첩 함수는 `클로저`와 관련이 있다.

▼ 콜백 함수

- 함수의 매개변수를 통해 다른 함수의 내부로 전달되는 함수. **(repeat함수는 고차함수)**
- 매개변수를 통해 함수의 외부에서 콜백 함수를 전달받은 함수를 **고차함수**라고 한다.**(logAll함수와 logOdds 함수는 콜백함수)**

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

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

// 반복 호출할 함수를 인수로 전달한다.
repeat(5, logAll); // 0 1 2 3 4

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

// 반복 호출할 함수를 인수로 전달한다.
repeat(5, logOdds); // 1 3
```

▼ 순수함수와 비순수 함수

함수형 프로그래밍에서는 어떤 외부 상태에 의존하지도 않고 변경하지도 않는, 즉 부수 효과가 없는 함수를 **순수 함수**라고 합니다. 또한 외부 상태에 의존하거나 외부 상태를 변경하는, 즉 부수 효과가 있는 함수를 **비순수 함수**라고 합니다.

순수함수

```
var count = 0; // 현재 카운트를 나타내는 상태

// 순수 함수 increase는 동일한 인수가 전달되면 언제나 동일한 값을 반환한다.
function increase(n) {
  return ++n;
}

// 순수 함수가 반환한 결과값을 변수에 재할당해서 상태를 변경
count = increase(count);
console.log(count); // 1
```

```
count = increase(count);
console.log(count); // 2
```

비순수 함수

```
var count = 0; // 현재 카운트를 나타내는 상태: increase 함수에 의해 변화한다.

// 비순수 함수
function increase() {
  return ++count; // 외부 상태에 의존하며 외부 상태를 변경한다.
}

// 비순수 함수는 외부 상태(count)를 변경하므로 상태 변화를 추적하기 어려워진다.
increase();
console.log(count); // 1

increase();
console.log(count); // 2
```

매개변수를 통해 객체를 전달받으면 비순수 함수가 된다. 순수 함수를 사용하는 것이 좋다.

함수형 프로그래밍은 외부 상태를 변경하는 부수효과를 최소화해서 불변성(immutable)을 지향하는 프로그래밍 패러다임이다.

함수형 프로그래밍은 순수 함수를 통해 부수 효과를 최대한 억제하여 오류를 피하고, 프로그램의 안정성을 높이는 노력의 일환이다.