2026.01.06.화

생성일	@2026년 1월 6일 오전 8:52
태그	JavaScript

목차

1. 함수 알아보기
2. 객체 알아보기
3. 생성자 함수
4. 클래스와 상속
5. 접근자 프로퍼티
6. 캡슐화
7. 프로퍼티 어트리뷰트
8. 스프레드
9. 디스트럭쳐링

▼ 1) 함수 알아보기

- 일급 객체: 함수를 변수와 같이 다루는 개념
  - 상수, 변수에 함수 할당 가능
  - 다른 함수의 인자로 전달 가능
    - 고차함수(high-order funcion): 전달받는 함수
    - 콜백함수(callback functino): 전달되는 함수
  - 다른 함수의 결과값으로 반환 가능
- 상수, 변수 할당 예시

```
const greet = function(name) {
   return "Hello, " + name + "!";
};
const greetAgain = greet;

console.log(greet("kim"));
// Hello, kim!
console.log(greetAgain("kim"));
// Hello, kim!
```

```
const person = {
   age: 25;
   greet function(name) {
   return "안녕하세요. " + name + "님!";
   }
};
console.log(person.greet("홍길동"));
```

```
const function = [
 function(name) { return "안녕하세요, " + name + "님!"; },
 function(name) { return "반갑습니다, " + name + "님!"; }
];
// 배열의 첫 번째 함수를 호출
console.log(functions("홍길동")); // "안녕하세요, 홍길동님!" 출력
```

```
// 배열의 두 번째 함수를 호출
console.log(functions[1]("이순신")); // "반갑습니다, 이순신님!" 출력
```

- 다른 함수의 인자로 사용 예시

```
// 인자로 전달될 함수
function greet(name) {
 return "안녕하세요, " + name + "님!";
}
// 함수를 인자로 받는 함수
function processUserInput(callback) {
 const name = "홍길동";
 console.log(callback(name));
}
// 함수 `greet`를 인자로 전달
processUserInput(greet); // "안녕하세요, 홍길동님!" 출력
```

```
// calculate
function add(a, b) { return a + b; }
function subtract(a, b) { return a - b; }
function multiply(a, b) { return a * b; }
// evaluate
function isOdd(number) { return !!(number % 2); }
function isPositive(number) { return number > 0; }
function calcAndEval(calc, eval, x, y) {
 return eval(calc(x, y));
}
console.log(
 calcAndEval(add, isOdd, 3, 9), // false (12 짝수)
 calcAndEval(subtract, isPositive, 3, 9), // false (-6 음수)
 calcAndEval(multiply, isOdd, 3, 9) // true (27 홀수)
);
```

- 중첩 함수: 함수 블록 안에 다른 함수가 포함되는 경우

```
function outerFunction() {
 let functionName = "외부 함수";
 console.log(functionName, "입니다.");
 function innerFunction() {
 let functionName = "내부 함수";
 console.log(functionName, "입니다.");
 }
 innerFunction(); // 내부 함수 호출
}
outerFunction(); // 외부 함수 호출
```

- 재귀함수
  - base case: 재귀 호출을 중단하는 조건 (무한 루프 방지)
  - recursive case: 함수가 자기 자신을 호출하여 반복하는 부분

```
function sum(n) {
 // 기본 사례: n이 1이면, 합은 1입니다.
 if (n === 1) {
 return 1;
 }
 // 재귀 사례: n + sum(n-1)을 계산합니다.
 return n + sum(n - 1);
```

```
}
console.log(sum(5)); // 15 출력 (5 + 4 + 3 + 2 + 1)
```

▼ 2) 객체 알아보기

- 객체에 무언가를 선언하거나 접근할 때 객체를 넣으면 object Object 문자열로 치환되어 접근됨.
- console.log(obj['object Object']);형태로 접근하여도 “객체 키” 라는 값으로 나오게 됨.
- 정리하자면, 실제로 해당 객체나 배열의 내용이나 참조값이 키가 되는 것이 아닌 것.
- 따라서, 객체에는 객체나 배열을 키로 사용하시면 안됨.
- Delete 연산자: 객체 프로퍼티 삭제
  - 삭제 성공 실패 여부를 나타내는 boolean 값을 반환
  - 전역 객체나 함수 인수와 같은 몇 가지 특정 상황에서는 사용이 제한 될 수 있음

```
let person = {
 name: "Alice",
 age: 30,
 job: "Developer"
};
console.log(person); // { name: "Alice", ... }
delete person.job; // job 프로퍼티 삭제
console.log(person); // { name: "Alice", age: 30 }
console.log(person.job); // undefined 출력
```

▼ 3) 생성자 함수

- 객체를 생성하는데 사용되는 함수
  - 만들어진 객체를 인스턴스라고 부름
  - this 키워드 사용
  - new 키워드 호출로 새로운 객체 생성
  - 생성자 함수 자체는 객체가 아니기에 메서드 정의 못함
  - function은 기본적으로 생성자 함수의 기능을 가짐

```
function Car(model, year) {
 this.model = model;
 this.year = year;
 this.getInfo = function() {
 return `${this.year} ${this.model}`;
 };
}
const car1 = new Car("Toyota", 2021);
const car2 = new Car("Honda", 2020);
console.log(car1.getInfo()); // 2021 Toyota
console.log(car2.getInfo()); // 2020 Honda
```

- 객체 반환 함수 (Object Returning Function)

```
function createAnimal(type, sound) {
 return {
 type: type,
 sound: sound,
 makeSound: function() {
 return `${this.type} says ${this.sound}!`;
 }
 };
}
const animal2 = createAnimal("Cat", "Meow");
```

```
const animal2 = createAnimal("Cat", "Moo");
console.log(animal2.makeSound()); // Cat says Meow!
console.log(animal3.makeSound()); // Cat says Moo
```

- 생성자 함수 (Constructor Function)

```
function Animal(type, sound) {
 this.type = type;
 this.sound = sound;
}
Animal.prototype.makeSound = function() {
 return `${this.type} says ${this.sound}!`;
};
const animal4 = new Animal("Lion", "Roar");
const animal5 = new Animal("Sheep", "Baa");
console.log(animal4.makeSound()); // "Lion says Roar"
console.log(animal5.makeSound()); // "Sheep says Baa"
```

▼ 4) 클래스와 상속

- 클래스 선언: class
  - constructor 메서드: 인스턴스 초기화 담당
  - 메서드: 클래스 내에서 정의되며 인스턴스에서 사용할 수 있음

```
class Animal {

 constructor(type, sound) {
    this.type = type;
    this.sound = sound;
 }

 makeSound() {
    return `${this.type} says ${this.sound}!`;
    }
 }

const dog = new Animal('Dog', 'Woof');
const dog = new Animal('Cat', 'Meow');
console.log(dog.makeSound());
console.log(cat.makeSound());
```

- 상속
  - extends: 자식 클래스가 부모 클래스를 상속받을 때 사용
  - super: 자식 클래스의 생성자에서 부모 클래스의 생성자를 호출

```
Animal {
   constructor(name) { this.name = name; }
   speak() { console.log(`${this.name} makes a sound.`); }
}

class Dog extends Animal {
   constructor(name, breed) {
      super(name);
      this.breed = breed;
   }
   speak() {
      console.log(`${this.name} barks.`);
   }
```

```
}

const myDog = new Dog(`바둑이', '진돗개');

myDog.Speak();
```

▼ 5) 접근자 프로퍼티

- 객체의 속성에 대한 접근을 제어하기 위해 사용되는 프로퍼티
- 데이터 프로퍼티와 다르게 값을 직접 저장하지 않고 getter와 setter 함수를 사용하여 속성 값을 가져오거나 변경
  - getter: 속성을 읽을 때
  - setter: 속성을 쓸 때

```
onst inventory = {
    quantity: 10,
    get totalQuantity() {
        return this.quantity;
    },
    set totalQuantity(value) {
        if (value < 0) console.log("음수 불가");
    } else { this.quantity = value; }
};

console.log(inventory.totalQuantuty); // 10 출력

inventory.totalQuantuty = 15;
console.log(inventory.totalQuantity); // 15 출력

inventory.totalQuantuty = -5; // 음수 불가 출력
console.log(inventory.totalQuantity); // 15 출력
```

▼ 6) 캡슐화 (Encapsulation)

- 인스턴스의 프로퍼티 값을 함부로 열람하거나 수정하지 못하도록 함.
- private 필드는 # 기호를 사용해 정의

```
class BankAccount {
  // Private 필드 선언 (클래스 본문 최상단)
  #balance = 0;
  #accountNumber;
  #transactionHistory = [];

  constructor(accountNumber, initialBalance = 0) {
    this.#accountNumber = accountNumber;
    this.#balance = initialBalance;
  }

  // Public 메서드 - 외부 인터페이스
  deposit(amount) {
    if (amount <= 0) {
      throw new Error('입금액은 0보다 커야 합니다');
    }
    this.#balance += amount;
    this.#recordTransaction('입금', amount);
    return this.#balance;
  }

  withdraw(amount) {
    if (amount > this.#balance) {
```

```
      throw new Error('잔액이 부족합니다');
    }
    this.#balance -= amount;
    this.#recordTransaction('출금', amount);
    return this.#balance;
  }

  // Private 메서드
  #recordTransaction(type, amount) {
    this.#transactionHistory.push({
      type,
      amount,
      timestamp: new Date(),
      balance: this.#balance
    });
  }

  // Getter - 읽기 전용 접근
  get balance() {
    return this.#balance;
  }

  get accountNumber() {
    return this.#accountNumber.replace(/(\d{4})/g, '$1-').slice(0, -1);
  }
}

const account = new BankAccount('1234567890', 1000);
console.log(account.balance); // 1000
account.deposit(500); // 1500

// ❌ 직접 접근 불가 - SyntaxError
// console.log(account.#balance);
// account.#recordTransaction('해킹', 1000000);
```

▼ 7) 프로퍼티 어트리뷰트

- 프로퍼티의 메타정보
- Data Property
  - value: 실제값
  - writable: 프로퍼티 값의 수정 가능 여부
  - enumerable: 프로퍼티 열거, 나열 가능 여부
  - configurable: 메타 정보를 변경하거나 프로퍼티 삭제 가능 여부
- Accessor Property
  - get: 프로퍼티 값 접근 함수
  - set: 프로퍼티 값 변경 함수
  - enumerable: 프로퍼티 열거 가능 여부
  - configurable: 메타 정보를 변경하거나 프로퍼티 삭제 가능 여부

▼ 8) 스프레드

- "..." 연산자를 사용하여 배열이나 객체와 같은 이터러블 객체를 개별 요소로 분리하거나, 새로운 배열 또는 객체를 만들 때 기존의 요소들을 간편하게 복사하고 추가하는 데 사용되는 문법
  - 배열 복사 (옅은 복사)

```
const original = [1, 2, 3];
const copied = [...original];

// 얕은 복사의 한계
const nested = [1, [2, 3], 4];
const shallowCopy = [...nested];
shallowCopy[1][0] = 999;
console.log(nested[1][0]); // 999 (원본도 변경됨!)

// 깊은 복사가 필요한 경우
const deepCopy = JSON.parse(JSON.stringify(nested));
// 또는 structuredClone (최신 방법)
const deepCopy2 = structuredClone(nested);
```

- 배열 결합 및 삽입

```
const arr1 = [1, 2];
const arr2 = [3, 4];
const arr3 = [5, 6];

// 여러 배열 결합
const combined = [...arr1, ...arr2, ...arr3];

// 중간에 요소 삽입
const inserted = [...arr1, 'middle', ...arr2];
// [1, 2, 'middle', 3, 4]

// 배열 앞뒤에 요소 추가
const wrapped = [0, ...arr1, 5];
// [0, 1, 2, 5]
```

- 함수 호출 시 인자로 사용

```
// 배열을 개별 인자로 전달
const numbers = [1, 5, 3, 9, 2];
console.log(Math.max(...numbers)); // 9

// 기존 방식 (apply)
console.log(Math.max.apply(null, numbers)); // 9

// 여러 배열 결합하여 전달
const arr1 = [1, 2];
const arr2 = [3, 4];
someFunction(...arr1, 'middle', ...arr2, 'end');

// 생성자 함수 호출
const dateFields = [2024, 0, 1]; // 2024-01-01
const date = new Date(...dateFields);

// 가변 인자 함수 활용
function sum(...args) {
  return args.reduce((acc, val) => acc + val, 0);
}

const nums = [1, 2, 3, 4];
console.log(sum(...nums)); // 10
```

- 객체 복사/병합

```
const obj1 = { a: 1, b: 2 };
const obj2 = { c: 3, d: 4 };

// 객체 병합
const merged = { ...obj1, ...obj2 };
// { a: 1, b: 2, c: 3, d: 4 }

// 프로퍼티 오버라이드 (나중 것이 우선)
const override = {
  ...obj1,
  ...{ a: 999, b: 888 }
};
// { a: 999, b: 888 }

// 얕은 복사 주의사항
const original = {
  name: 'John',
  address: { city: 'Seoul', country: 'Korea' }
};
const copied = { ...original };
copied.address.city = 'Busan';
console.log(original.address.city); // 'Busan' (원본 변경!)
```

▼ 9) 디스트럭쳐링

- 구조 분해 할당을 통해 데이터를 쉽게 추출

```
// 기존 방식
const arr = [1, 2, 3];
const first = arr[0];
const second = arr[1];

const obj = { name: 'John', age: 30 };
const name = obj.name;
const age = obj.age;

// 디스트럭처링
const [first, second] = [1, 2, 3];
const { name, age } = { name: 'John', age: 30 };
```

- 배열 디스트럭쳐링

```
// 기본 할당
const [a, b, c] = [1, 2, 3];
console.log(a, b, c); // 1 2 3

// 일부만 추출
const [first, second] = [1, 2, 3, 4, 5];
console.log(first, second); // 1 2

// 요소 건너뛰기
const [, , third, fourth] = [1, 2, 3, 4, 5];
console.log(third, fourth); // 3 4

// 변수 교환 (swap)
let x = 1, y = 2;
[x, y] = [y, x];
console.log(x, y); // 2 1
```

```
// 함수 반환값 분해
function getCoordinates() {
  return [100, 200];
}
const [x, y] = getCoordinates();
```

- 객체 디스트럭쳐링

```
// 기본 할당
const { name, age } = { name: 'John', age: 30, city: 'Seoul' };
console.log(name, age); // 'John' 30

// 변수명 변경
const { name: userName, age: userAge } = { name: 'John', age: 30 };
console.log(userName, userAge); // 'John' 30

// 존재하지 않는 속성
const { x, y, z } = { x: 1, y: 2 };
console.log(x, y, z); // 1 2 undefined

// 계산된 속성명
const key = 'dynamicKey';
const { [key]: value } = { dynamicKey: 'dynamicValue' };
console.log(value); // 'dynamicValue'

// 여러 변수명 변경
const person = { firstName: 'John', lastName: 'Doe', age: 30 };
const {
  firstName: first,
  lastName: last,
  age: years
} = person;
console.log(first, last, years); // 'John' 'Doe' 30
```

- 함수 매개변수에서의 사용

```
// 배열 매개변수 디스트럭처링
function printCoordinates([x, y]) {
  console.log(`X: ${x}, Y: ${y}`);
}
printCoordinates([100, 200]); // X: 100, Y: 200

// 객체 매개변수 디스트럭처링
function greet({ name, age }) {
  console.log(`Hello, ${name}! You are ${age} years old.`);
}
greet({ name: 'John', age: 30 }); // Hello, John! You are 30 years old.

// 기본값 설정
function createUser({
  name = 'Anonymous',
  age = 18,
  role = 'user'
}) {
  return { name, age, role };
}

console.log(createUser({ name: 'John' }));
// { name: 'John', age: 18, role: 'user' }
```

```
// 매개변수 자체의 기본값과 속성 기본값
function config({
  host = 'localhost',
  port = 8080
} = {}) {
  return { host, port };
}

console.log(config()); // { host: 'localhost', port: 8080 }
console.log(config({})); // { host: 'localhost', port: 8080 }
console.log(config({ port: 3000 }));
// { host: 'localhost', port: 3000 }
```