객체

2020-03-16

이승진

**학습목표**

객체의 생성, 비교, 속성 탐색

Object의 메소드

**목차**

[1. 객체 2](#_Toc36559560)

[1) 객체 생성 2](#_Toc36559561)

[2) 객체 비교 7](#_Toc36559562)

[3) 객체 배열 9](#_Toc36559563)

[4) 메소드 13](#_Toc36559564)

[5) 속성 탐색 15](#_Toc36559565)

[6) 멤버 변수 제거 17](#_Toc36559566)

[2. Object의 메소드 18](#_Toc36559567)

[1) Object.assign 메소드 18](#_Toc36559568)

[2) Object.entries 19](#_Toc36559569)

[3) Object.keys, Object.values 20](#_Toc36559570)

[4) Object.freeze 메소드 21](#_Toc36559571)

[3. 고급 주제 22](#_Toc36559572)

[1) 속성 순서 22](#_Toc36559573)

[2) Map 24](#_Toc36559574)

[4. 클래스 27](#_Toc36559575)

[1) 클래스 정의 27](#_Toc36559576)

[2) 메소드 구현 28](#_Toc36559577)

[3) static 메소드 구현 29](#_Toc36559578)

[4) 상속 구현 30](#_Toc36559579)

[5) getter / setter 31](#_Toc36559580)

[5. 요약 33](#_Toc36559581)

# 객체

## 객체 생성

### person1.js

|  |
| --- |
| let person = { name: "홍길동", age: 16 };  console.log(person);  console.log(person.name);  console.log(person.age); |

출력

|  |
| --- |
| { name: '홍길동', age: 16 }  홍길동  16 |

let person = { name: "홍길동", age: 16 };

객체를 생성하고, 그 객체에 대한 참조를 person 변수에 대입한다.

그 객체 내부에 name 멤버 변수와 age 멤버 변수가 만들어진다.

name 멤버 변수에 "홍길동" 문자열이 대입된다.

age 멤버 변수에 16 을 대입된다.



console.log(person);

person 변수가 참조하는 객체를 출력한다.

console.log(person.name);

person 변수가 참조하는 객체 내부의 name 멤버 변수 값을 출력한다.

### 참조형 (reference type)

객체는 참조형이다.

객체 변수에는 그 객체에 대한 참조(reference)만 저장된다.

### 기본 자료형 (primitive type)

기본 자료형은 값형이다. (value type)

기본 자료형 변수에는 값이 저장된다.

### 참고

|  |
| --- |
| Java에서 String은 객체이고 참조형이다.  Javascript에서 문자열은 기본 자료형이고 값형이다.. |

### person2.js

|  |
| --- |
| let person1 = { };  person1.name = "홍길동";  person1.age = 16;  console.log(person1);  let person2 = { name: "임꺽정" }  person2.age = 19;  console.log(person2); |

출력

|  |
| --- |
| { name: '홍길동', age: 16 }  { name: '임꺽정', age: 19 } |

let person1 = { };

객체를 생성하고, 그 객체에 대한 참조를 person1 변수에 대입한다.

그 객체 내부에 멤버 변수가 없다.

person1.name = "홍길동";

person1 변수가 참조하는 객체에, name 맴버 변수가 있다면, 그 변수에 "홍길동" 문자열을 대입한다.

name 멤버 변수가 없다면 만들고, 그 변수에 "홍길동" 문자열을 대입한다.

person1.age = 16;

person1 변수가 참조하는 객체에, age 맴버 변수가 있다면, 그 변수에 16을 대입한다.

agee 멤버 변수가 없다면 만들고, 그 변수에 16을 대입한다.

let person2 = { name: "임꺽정" }

객체를 생성하고, 그 객체에 대한 참조를 person2 변수에 대입한다.

그 객체 내부에 name 멤버가 만들어진다.

name 멤버 변수에 "임꺽정" 문자열이 대입된다.

person2.age = 19;

person2 변수가 참조하는 객체에, age 맴버 변수가 있다면, 그 변수에 19 를 대입한다.

agee 멤버 변수가 없다면 만들고, 그 변수에 19 를 대입한다.

### person3.js

|  |
| --- |
| let person = { };  person["name"] = "홍길동";  person["age"] = 16;  console.log(person["name"]);  console.log(person["age"]); |

출력

|  |
| --- |
| 홍길동  16 |

person["name"] = "홍길동";

위 코드는 아래 코드와 동일하다.

person.name = "홍길동";

person["age"] = 16;

위 코드는 아래 코드와 동일하다.

person.age = 16;

console.log(person["name"]);

위 코드는 아래 코드와 동일하다.

console.log(person.name);

console.log(person["age"]);

위 코드는 아래 코드와 동일하다.

console.log(person. age);

### person4.js

|  |
| --- |
| function createPerson(s, i) {    return { name: s, age: i };  }  let person1 = createPerson("홍길동", 16);  let person2 = createPerson("임꺽정", 18);  console.log(person1);  console.log(person2); |

출력

|  |
| --- |
| { name: '홍길동', age: 16 }  { name: '임꺽정', age: 18 } |

function createPerson(s, i) {

  return { name: s, age: i };

}

createPerson 함수는 객체를 생성하여 리턴한다.

그 객체 내부에는 name 멤버 변수와 age 멤버 변수가 만들어진다.

name 멤버 변수에 s 파라미터 변수의 값이 대입되고,

age 멤버 변수에 i 파라미터 변수의 값이 대입된다.

let person1 = createPerson("홍길동", 16);

createPerson 함수가 리턴하는 객체에 대한 참조를 person1 변수에 대입한다.

### person5.js

|  |
| --- |
| function createPerson() {    return { name: "홍길동", age: 16 }  }  let person1 = createPerson()  let person2 = createPerson()  person1.name = "임꺽정"  person2.age = 20  console.log(person1)  console.log(person2) |

출력

|  |
| --- |
| { name: '임꺽정', age: 16 }  { name: '홍길동', age: 20 } |

createPerson 함수는 호출될 때 마다 새 객체를 생성해서 리턴한다.

그래서 person1 변수와 person2 변수가 참조하는 객체는 서로 다른 두 객체이다.

## 객체 비교

### compare1.js

|  |
| --- |
| let person1 = { name: "홍길동", age: 16 };  let person2 = { name: "홍길동", age: 16 };  let p = person1;  console.log(person1 == person2);  console.log(person1 == p); |

출력

|  |
| --- |
| false  true |

let person1 = { name: "홍길동", age: 16 };

객체를 한 개 생성하고, 그 객체에 대한 참조를 person1 변수에 대입한다.

let person2 = { name: "홍길동", age: 16 };

또 객체를 한 개 생성하고, 그 객체에 대한 참조를 person2 변수에 대입한다.

let p = person1;

person1 변수의 값을 변수 p에 대입한다.

person1 변수의 값은 객체에 대한 참조이다. 그 참조를 변수 p에 대입한다.



### 참조 비교 (== 연산자)

참조형인 객체를 == 연산자로 비교하면, 객체 내부 값을 비교하지 않고,

두 참조를 비교한다. (두 참조가 동일한 객체를 참조하는지 비교한다.)

위 그림에서

person1 변수에 p 변수는 같은 객체를 참조하고 있다.

따라서 person1 == p 표현식의 값은 true 이다.

person1 변수와 person2 변수는 서로 다른 객체를 참조하고 있다.

따라서 person1 == person2 표현식의 값은 false 이다.

### 참고

|  |
| --- |
| 참조를 비교하는 것은 identity를 비교하는 것이다.  내부 값을 비교하는 것은 equality를 비교하는 것이다. |

### compare2.js

|  |
| --- |
| let s1 = "hello world";  let s2 = "hello " + "world";  console.log(s1 == s2); |

출력

|  |
| --- |
| true |

javascript에서 문자열은 기본 자료형이다.

기본 자료형을 == 연산자로 비교하면, 값이 같은지를 비교한다.

s1 변수의 값과 s2 변수의 값이 같으므로, s1 == s2 표현식의 값은 true 이다.

### compare3.js

|  |
| --- |
| let person1 = { name: "홍길동", age: 16 };  let person2 = { name: "홍길동", age: 16 };  function equals(p1, p2) {    return p1.name == p2.name &&           p1.age == p2.age;  }  console.log(equals(person1, person2)); |

출력

|  |
| --- |
| true |

객체의 내부 값을 비교하려면, 위와 같은 비교 함수를 구현해야 한다.

### 주의

|  |
| --- |
| java에서는 이름이 같은 메소드가 여러 개일 수 있다. (method overloading)  javscript에서는 이름이 같은 함수가 여러 개일 수 없다. |

## 객체 배열

### arr1.js

|  |
| --- |
| let person1 = { name: "홍길동", age: 16 };  let person2 = { name: "임꺽정", age: 18 };  let person3 = { name: "전우치", age: 19 };  let persons = [ person1, person2, person3 ];  console.log(persons); |

출력

|  |
| --- |
| [  { name: '홍길동', age: 16 },  { name: '임꺽정', age: 18 },  { name: '전우치', age: 19 }  ] |

배열도 참조형이다.

배열 변수에는 배열에 대한 참조만 저장된다.

let persons = [ person1, person2, person3 ];

배열이 생성되고, 이 배열에 대한 참조가 persons 변수에 대입된다.

persons[0]에 person1 변수의 값이 대입된다. 즉 person1 변수가 참조하는 객체에 대한 참조가 대입된다.

마찬가지로 person[1], person[2]에도, person2, person3 변수의 값이 각각 대입된다.



### arr2.js

|  |
| --- |
| let persons = [    { name: "홍길동", age: 16 },    { name: "임꺽정", age: 18 },    { name: "전우치", age: 19 }  ];  console.log(persons);  for (let i = 0; i < persons.length; ++i)    console.log(persons[i]); |

출력

|  |
| --- |
| [  { name: '홍길동', age: 16 },  { name: '임꺽정', age: 18 },  { name: '전우치', age: 19 }  ]  { name: '홍길동', age: 16 }  { name: '임꺽정', age: 18 }  { name: '전우치', age: 19 } |



### arr3.js

|  |
| --- |
| let persons = [    { name: "홍길동", age: 16 },    { name: "임꺽정", age: 18 },    { name: "전우치", age: 19 }  ];  console.log(persons);  persons[2] = persons[1];  persons[1].age = 20;  console.log(persons); |

출력

|  |
| --- |
| [  { name: '홍길동', age: 16 },  { name: '임꺽정', age: 18 },  { name: '전우치', age: 19 }  ]  [  { name: '홍길동', age: 16 },  { name: '임꺽정', age: 20 },  { name: '임꺽정', age: 20 }  ] |

let persons = [

  { name: "홍길동", age: 16 },

  { name: "임꺽정", age: 18 },

  { name: "전우치", age: 19 }

];



persons[2] = persons[1];

persons[1].age = 20;



### arr4.js

|  |
| --- |
| let persons = [    { name: "홍길동", age: 16 },    { name: "임꺽정", age: 18 }  ];  console.log(persons);  persons = [    { name: "전우치", age: 19 },    { name: "이몸룡", age: 16 }  ];  console.log(persons); |

출력

|  |
| --- |
| [ { name: '홍길동', age: 16 }, { name: '임꺽정', age: 18 } ]  [ { name: '전우치', age: 19 }, { name: '이몸룡', age: 16 } ] |

let persons = [

  { name: "홍길동", age: 16 },

  { name: "임꺽정", age: 18 }

];



persons = [

  { name: "전우치", age: 19 },

  { name: "이몸룡", age: 16 }

];



## 메소드

### method1.js

|  |
| --- |
| let rectangle = {    width: 5,    height: 7,    area : function() { return this.width \* this.height; }  };  console.log(rectangle.area()); |

실행

|  |
| --- |
| 35 |

let rectangle = {

  width: 5,

  height: 7,

  area : function() { return this.width \* this.height; }

};

객체가 한 개 생성되고, 그 객체에 대한 참조가 rectangle 변수에 대입된다.

이 객체에는 width, height, area 멤버 변수가 만들어진다.

width 멤버 변수에 5가 대입되고, height 멤버 변수에 7이 대입된다.

area 멤버 변수에 함수가 대입된다.

객체의 멤버 변수에 대입된 함수는, 그 객체의 메소드가 된다.

rectangle.area()

rectangle 객체의 area 멤버 변수의 값인 함수를 호출한다.

이때 이 함수 본문에서 this는 rectangle 객체를 참조한다.

console.log(rectangle.area());

rectangle 객체의 area 메소드를 호출하고, 그 리턴값을 출력한다.

### 주의

java 메소드에서 this는 생략 가능하다.

javascript 메소드에서 this는 생략할 수 없다.

### method2.js

|  |
| --- |
| let rectangle = {    width: 5,    height: 7  };  rectangle.area = function () {    return this.width \* this.height;  }  console.log(rectangle.area()); |

실행

|  |
| --- |
| 35 |

rectangle.area = function () {

  return this.width \* this.height;

}

reactangle 객체에 area 멤버 변수가 있다면, 그 멤버 변수에 함수를 대입한다.

area 멤버 변수가 없다면 만들고, 그 멤버 변수에 함수를 대입한다.

객체의 멤버 변수에 대입된 함수는, 그 객체의 메소드가 된다.

## 속성 탐색

### for1.js

|  |
| --- |
| let person = { name: "홍길동", age: 16 };  for (let key in person) {    let value = person[key];    console.log("%s: %s", key, value);  } |

출력

|  |
| --- |
| name: 홍길동  age: 16 |

for (let key in person) {

person 객체의 멤버 변수 각각에 대해서, 그 이름이 key 변수에 대입되고 반복문이 실행된다.

person 객체의 멤버 변수는 두 개 이므로, 위 for 문은 두 번 반복한다.

첫째 반복에서 key 변수에 "name" 문자열이 대입된다.

둘째 반복에서 key 변수에 "age" 문자열이 대입된다.

let value = person[key];

key 변수의 값이 "name" 이면, person 객체의 name 멤버 변수 값이 value 변수에 대입된다.

key 변수의 값이 "age" 이면, person 객체의 age 멤버 변수 값이 value 변수에 대입된다.

let value = person.name;

위 코드와 아래 코드는 동일하다.

let value = person["name"];

위 코드와 아래 코드는 동일하다.

let key = "name";

let value = person[key];

### for2.js

|  |
| --- |
| let rectangle = {    width: 5,    height: 7,    area : function() { return this.width \* this.height; }  };  for (let key in rectangle) {    let value = rectangle[key];    console.log("%s: %s", key, value);  } |

출력

|  |
| --- |
| width: 5  height: 7  area: function() { return this.width \* this.height; } |

rectangle 객체 내부의 멤버 변수는 세 개이다. (width, height, area)

area 멤버 변수의 값은 함수이다.

## 멤버 변수 제거

### delete1.js

|  |
| --- |
| let person = { name: "홍길동", age: 16, department: "소프" };  console.log(person);  delete person.department;  console.log(person); |

출력

|  |
| --- |
| { name: '홍길동', age: 16, department: '소프' }  { name: '홍길동', age: 16 } |

delete person.department;

person 객체의 department 멤버 변수를 제거한다.

# Object의 메소드

## Object.assign 메소드

### Object.assign(to, from)

from 객체의 모든 멤버 변수 값을, to 객체에 복사한다.

from 객체의 멤버 변수와 같은 이름의 멤버 변수가 to 객체 있다면,

to 객체의 멤버 변수의 값이 from 객체의 멤버 변수 값으로 바뀐다.

from 객체의 멤버 변수와 같은 이름의 멤버 변수가 to 객체에 없다면,

그 멤버 변수가 to 객체에 새로 만들어지고 값이 대입된다.

assign 메소드는 to 객체를 리턴한다.

### assign1.js

|  |
| --- |
| let person = { name: "홍길동", age: 16 };  let info = { age: 20, department: "소프", year: 2 };  Object.assign(person, info);  console.log(person); |

출력

|  |
| --- |
| { name: '홍길동', age: 20, department: '소프', year: 2 } |

### assign2.js - 객체 복제

|  |
| --- |
| let person1 = { name: "홍길동", age: 16 };  let person2 = Object.assign({ }, person1);  console.log(person2); |

출력

|  |
| --- |
| { name: '홍길동', age: 16 } |

### 객체 복제

let person2 = Object.assign({ }, person1);

person1 객체의 모든 멤버 변수값을 { } 빈 객체에 복사한다.

그렇게 값이 채워진 객체가 리턴되어, person2 변수에 대입된다.

즉 person1 객체를 복제한 새 객체가 만들어져서, person2 변수에 대입되었다.

## Object.entries

### Object.entries(객체)

객체의 모든 멤버 변수 값이 들어있는 2차원 배열을 리턴한다.

리턴되는 배열은 다음과 같은 형태이다.

[ [ 멤버변수이름1: 값], [ 멤버변수이름2: 값], [ 멤버변수이름3: 값], ... ]

### entries1.js

|  |
| --- |
| let person = { name: "홍길동", age: 16 };  console.log(Object.entries(person)); |

출력

|  |
| --- |
| [ [ 'name', '홍길동' ], [ 'age', 16 ] ] |

### entries2.js

|  |
| --- |
| let person = { name: "홍길동", age: 16 };  for (let [key, value] of Object.entries(person))    console.log("%s: %s", key, value); |

출력

|  |
| --- |
| name: 홍길동  age: 16 |

Object.entries(person) 메소드가 리턴하는 배열의 원소 각각에 대해서,

그 원소를 [key, value]에 대입하여 (destructing assignment)

for 문의 본문을 반복 실행한다.

Object.entries(person) 메소드가 리턴하는 배열의 원소는

[멤버변수이름, 값] 형태의 배열이다.

## Object.keys, Object.values

### Object.keys(객체)

객체의 멤버 변수 이름 목록을 배열로 리턴하다.

### keys1.js

|  |
| --- |
| let person = { name: "홍길동", age: 16, department: "속프" };  console.log(Object.keys(person)); |

출력

|  |
| --- |
| [ 'name', 'age', 'department' ] |

### Object.values(객체)

객체의 멤버 변수 이름 목록을 배열로 리턴하다.

### keys2.js

|  |
| --- |
| let person = { name: "홍길동", age: 16, department: "속프" };  console.log(Object.values(person)); |

출력

|  |
| --- |
| [ '홍길동', 16, '속프' ] |

## Object.freeze 메소드

### Object.freeze(객체)

객체를 수정할 수 없는 상태로 변경한다.

객체 내부 값을 변경할 수 없게 된다.

### freeze1.js

|  |
| --- |
| let person1 = { name: "홍길동", age: 16 };  person1.age = 20;  person1.department = "소프";  console.log(person1);  let person2 = { name: "홍길동", age: 16 };  Object.freeze(person2);  person2.age = 20;  person2.department = "소프";  console.log(person2); |

출력

|  |
| --- |
| { name: '홍길동', age: 20, department: '소프' }  { name: '홍길동', age: 16 } |

Object.freeze(person2);

person2 객체를 수정할 수 없게 된다.

멤버 변수 값을 변경할수 없고, 새 멤버 변수를 추가할 수도 없고, 멤버 변수를 제거할 수도 없다.

### Object.isFrozen(객체)

객체가 freeze 되었는지 여부를 리턴한다.

### freeze2.js

|  |
| --- |
| let person = { name: "홍길동", age: 16 };  console.log(Object.isFrozen(person));  Object.freeze(person);  console.log(Object.isFrozen(person)); |

출력

|  |
| --- |
| false  true |

# 고급 주제

## 속성 순서

사실 javascript 객체의 멤버 변수란 것은 정확한 표현이 아니다.

멤버 변수가 아니고 속성(property)라고 표현하는 것이 맞다.

단순한 변수가 아니기 때문이다.

편의상 멤버 변수와 호칭을 섞어서 사용하도록 하겠다.

### property1.js

숫자 속성명이 가능하다.

|  |
| --- |
| let person = { name: "홍길동", age: 16 }  person[0] = "호형호제"  person[1] = "도술"  console.log(person)  console.log(person[0]) |

출력

|  |
| --- |
| { '0': '호형호제', '1': '도술', name: '홍길동', age: 16 }  호형호제 |

person 객체는 배열이 아니다.

person[0] 이것은 배열의 0 번째 원소로 해석할 수 없고,

person 객체의 '0' 속성으로 해석해야 한다.

### property2.js

|  |
| --- |
| let person = { name: "홍길동", age: 16, 1: "도술", 0: "호형호제"  }  console.log(person)  console.log(person[0]) |

출력

|  |
| --- |
| { '0': '호형호제', '1': '도술', name: '홍길동', age: 16 }  호형호제 |

### property3.js

|  |
| --- |
| let person = { name: "홍길동", age: 16, 1: "도술", 0: "호형호제"  }  for (let key in person)      console.log("%s: %s", key, person[key]) |

출력

|  |
| --- |
| 0: 호형호제  1: 도술  name: 홍길동  age: 16 |

객체의 속성을 for in 반복문으로 탐색할 수 있다.

속성의 탐색 순서는 다음과 같다.

- 속성명이 숫자인 속성들이 먼저, 속성명이 문자열인 속성들이 뒤

- 속성명이 숫자인 속성들의 순서는, 그 숫자의 오름차순

- 속성명이 문자열인 속성들의 순서는, 속성이 만들어진 순서

이 순서는, Object.keys(...), Object.values(...), Object.entries(...) 메소드가 리턴하는 배열의 순서이기도 하다.

## Map

javascript 객체를 Java의 Map 처럼 사용할 수 있다.

1 이상 1000 이하의 수를 랜덤하게 100 개를 생성했을 때,

두 번 이상 생성된 수를 출력하는 코드를 구현하자.

### count1.js

|  |
| --- |
| const MAX = 1000, MIN = 1;  let count = [];  for (let i = 0; i < 100; ++i) {    let a = Math.floor(Math.random() \* (MAX - MIN + 1) + MIN);    if (typeof count[a] == "undefined") count[a] = 1;    else count[a] += 1;  }  for (let i = 0; i < count.length; ++i)    if (count[i] >= 2)      console.log("%d %d", i, count[i]);  console.log("length = ", count.length); |

count 배열을 이용하여, 각 수의 생성된 횟수를 센다.

마지막으로 count 배열의 크기를 출력한다.

출력

|  |
| --- |
| 676 2  840 2  length = 994 |

count 변수가 배열이고, 랜덤하게 생성된 값이 배열의 인덱스로 사용되기 때문에,

count 배열의 크기는, 랜덤하게 생성된 값의 최대값 + 1 이다.

### count2.js

|  |
| --- |
| const MAX = 1000, MIN = 1;  let count = { };  for (let i = 0; i < 100; ++i) {    let a = Math.floor(Math.random() \* (MAX - MIN + 1) + MIN);    if (typeof count[a] == "undefined") count[a] = 1;    else count[a] += 1;  }  for (let key in count)    if (count[key] >= 2)      console.log("%d %d", key, count[key]);  console.log("length = ", Object.keys(count).length); |

count 객체를 Map 처럼 활용하여, 각 수의 생성된 횟수를 센다.

마지막으로 count 객체의 크기를 출력한다.

객체의 속성명이 Map의 key 역할을 하고, 속성값이 Map의 value 역할을 한다.

출력

|  |
| --- |
| 10 2  175 2  207 2  630 2  890 2  length = 95 |

for (let key in count)

count 객체의 속서 각각에 대해서 반복한다.

반복할 때 마다 속성명이 key 변수에 대입된다.

if (count[key] >= 2)

count 객체의 key 속성값이 2 이상이면 true

### count3.js

|  |
| --- |
| const MAX = 1000, MIN = 1;  let count = { };  for (let i = 0; i < 100; ++i) {    let a = Math.floor(Math.random() \* (MAX - MIN + 1) + MIN);    count[a] = count[a] ? count[a] + 1 : 1;  }  for (let key in count)    if (count[key] >= 2)      console.log("%d %d", key, count[key]);  console.log("length = ", Object.keys(count).length); |

좀 더 간결하게 구현하였다.

count[a] ? count[a] + 1 : 1

count[a] 값이 undefined 이거나 0 이면,

조건식이 false가 되고, 이 표현식의 값이 1 이다.

count[a] 값이 0보다 큰 숫자이면

조건식이 true가 되고, 이 표현식의 값은 count[a] + 1 이다.

### count4.js

|  |
| --- |
| const MAX = 1000, MIN = 1;  let count = { };  for (let i = 0; i < 100; ++i) {    let a = Math.floor(Math.random() \* (MAX - MIN + 1) + MIN);    count[a] = (count[a] || 0) + 1;  }  for (let key in count)    if (count[key] >= 2)      console.log("%d %d", key, count[key]);  console.log("length = ", Object.keys(count).length); |

좀 더 간결하게 구현하였다.

(count[a] || 0)

count[a] 값이 false에 해당하는 값이면 (undefined 이거나 0 이면),

이 표현식의 값은 0 이다.

count[a] 값이 true에 해당하는 값이면 (0보다 큰 수이면),

이 표현식의 값이 count[a] 이다.

# 클래스

## 클래스 정의

### person1.js

|  |
| --- |
| class Person {    constructor(s, i) {      this.name = s;      this.age = i;    }  }  let person = new Person("홍길동", 16);  console.log(person); |

class Person {

Java 언어와 같은 방법으로 객체를 생성할 수 있다.

Person 클래스를 정의하고, Person 클래스로부터 객체를 생성할 수 있다.

constructor(s, i) {

생성자의 이름은 언제나 constructor 이다.

파라미터 변수 타입은 언제나 생략한다.

this.name = s;

this.age = i;

멤버 변수를 따로 선언할 필요 없다.

생성자에서 값을 대입할 때 멤버 변수가 생성된다.

let person = new Person("홍길동", 16);

Person 클래스의 객체를 생성한다.

### this 생략 불가능

javascript 메소드에서 this를 생략할 수 없다.

### 모두 public

javascript 멤버 변수는 모두 public 이다.

private 멤버 변수를 선언하는 문법이 없다.

## 메소드 구현

### rectangle1.js

|  |
| --- |
| class Rectangle {    constructor(w, h) {      this.width = w;      this.height = h;    }    area() {      return this.width \* this.height;    }  }  let rectangle = new Rectangle(10, 15);  console.log(rectangle);  console.log(rectangle.area()); |

area() {

return this.width \* this.height;

}

Rectangle 클래스에 area 메소드를 구현한다.

메소드의 리턴 타입은 언제나 생략한다.

javascript 언어는 변수의 타입, 파라미터 변수의 타입, 리턴 타입을 명시할 수 없다.

console.log(rectangle.area());

area 메소드의 리턴 값을 출력한다.

### 모두 public

javascript 메소드는 모두 public 이다.

private 멤버 메소드를 선언하는 문법이 없다.

## static 메소드 구현

### static1.js

|  |
| --- |
| class Point {    constructor(x, y) {      this.x = x;      this.y = y;    }    static distance(point1, point2) {      let dx = point1.x - point2.x;      let dy = point1.y - point2.y;      return Math.hypot(dx, dy);    }  }  let p1 = new Point(10, 15);  let p2 = new Point(25, 40);  console.log(Point.distance(p1, p2)); |

static distance(point1, point2) {

메소드 앞에 static 키워드를 붙이면, 그 메소드는 static 메소드가 된다.

static 메소드의 특징은, 현재 객체를 의미하는 this 키워드를 사용할 수 없다는 점이다.

this 키워드를 사용할 수 없기 때문에,

this 객체의 멤버 변수나, this 객체의 메소드를 호출할 수 없다.

this 객체가 아닌 다른 객체의 멤버 변수나 메소드를 사용하는 것은 가능하다.

## 상속 구현

### figure2.js

|  |
| --- |
| class Figure {    constructor(x, y) {      this.x = x;      this.y = y;    }    move(dx, dy) {      this.x += dx;      this.y += dy;    }  }  class Rectangle extends Figure {    constructor(x1, y1, x2, y2) {      super(x1, y1);      this.x2 = x2;      this.y2 = y2;    }    move(dx, dy) {      super.move(dx, dy);      this.x2 += dx;      this.y2 += dy;    }  }  let rect = new Rectangle(10, 10, 20, 25);  rect.move(5, 10);  console.log(rect); |

class Rectangle extends Figure {

Figure 클래스를 상속하여 Rectangle 클래스를 구현하였다.

부모 클래스의 멤버 변수와 메소드가 자식 클래스에 상속된다.

super(x1, y1);

부모 클래스의 생성자를 호출한다.

super.move(dx, dy);

부모 클래스의 move 메소드를 호출한다.

## getter / setter

### rectangle2.js

|  |
| --- |
| class Rectangle {    constructor(x1, y1, x2, y2) {      this.x1  = x1;      this.y1  = y1;      this.x2  = x2;      this.y2  = y2;    }    getWidth() {      return Math.abs(this.x2 - this.x1);    }    setWidth(width) {      this.x2 = this.x1 + width;    }    getHeight() {      return Math.abs(this.y2 - this.y1);    }    setHeight(height) {      this.y2 = this.y1 + height;    }  }  let rectangle = new Rectangle(5, 5, 20, 25);  rectangle.setWidth(30);  rectangle.setHeight(30);  console.log(rectangle);  console.log("%d %d", rectangle.getWidth(), rectangle.getHeight()); |

Rectangle 클래스에는 x1, y1, x2, y2 멤버 변수만 정의되었다.

getWidth, setWidth, getHeight, setHeight 메소드는

이 x1, y1, x2, y2 멤머 변수를 사용하여 구현되었다.

위와 같이 get 메소드와 get 메소드를 구현하는 방법 보다 더 세련된 문법이 제공된다.

### rectangle3.js

|  |
| --- |
| class Rectangle {    constructor(x1, y1, x2, y2) {      this.x1  = x1;      this.y1  = y1;      this.x2  = x2;      this.y2  = y2;    }    get width() {      return Math.abs(this.x2 - this.x1);    }    set width(width) {      this.x2 = this.x1 + width;    }    get height() {      return Math.abs(this.y2 - this.y1);    }    set height(height) {      this.y2 = this.y1 + height;    }  }  let rectangle = new Rectangle(5, 5, 20, 25);  rectangle.width = 30;  rectangle.height = 30;  console.log(rectangle);  console.log("%d %d", rectangle.width, rectangle.height); |

getter / setter 매소드를 구현하여 width 속성과 height 속성을 정의하였다.

width, height 속성은 멤버 변수가 아니고, getter / setter 메소드로 구현되었지만,

외부에서 사용할 때, 마치 멤버 변수인 것 처럼 사용할 수 있다.

# 요약

### 객체 생성

{ name: "홍길동", age: 16 };

### 빈 객체 생성

{ };

### 참조형 (reference type)

객체는 참조형이다.

객체 변수에는 그 객체에 대한 참조(reference)만 저장된다.

### 기본 자료형 (primitive type)

기본 자료형은 값형이다. (value type)

기본 자료형 변수에는 값이 저장된다.

### person1.age = 16;

person1 변수가 참조하는 객체에, age 맴버 변수가 있다면, 그 변수에 16을 대입한다.

agee 멤버 변수가 없다면 만들고, 그 변수에 16을 대입한다.

person["name"] = "홍길동";

위 코드는 아래 코드와 동일하다.

person.name = "홍길동";

function createPerson(s, i) {

  return { name: s, age: i };

}

createPerson 함수는 객체를 생성하여 리턴한다.

### 참조 비교 (== 연산자)

참조형인 객체를 == 연산자로 비교하면, 객체 내부 값을 비교하지 않고,

두 참조를 비교한다. (두 참조가 동일한 객체를 참조하는지 비교한다.)

javascript에서 문자열은 기본 자료형이다.

기본 자료형을 == 연산자로 비교하면, 값이 같은지를 비교한다.

### 배열도 참조형이다.

배열 변수에는 배열에 대한 참조만 저장된다.

### 메소드

객체의 멤버 변수에 대입된 함수는, 그 객체의 메소드가 된다.

let rectangle = {

  width: 5,

  height: 7,

  area : function() { return this.width \* this.height; }

};

rectangle.area = function () {

  return this.width \* this.height;

}

### 속성 탐색

for (let key in person) {

person 객체의 멤버 변수 각각에 대해서, 그 이름이 key 변수에 대입되고 반복문이 실행된다.

### 멤버 변수 제거

delete person.department;

person 객체의 department 멤버 변수를 제거한다.

### Object.assign(to, from)

from 객체의 모든 멤버 변수 값을, to 객체에 복사한다.

### Object.entries(객체)

객체의 모든 멤버 변수 값이 들어있는 2차원 배열을 리턴한다.

리턴되는 배열은 다음과 같은 형태이다.

[ [ 멤버변수이름1: 값], [ 멤버변수이름2: 값], [ 멤버변수이름3: 값], ... ]

### 객체 복제

let person2 = Object.assign({ }, person1);

즉 person1 객체를 복제한 새 객체가 만들어져서, person2 변수에 대입된다.

### Object.keys(객체)

객체의 멤버 변수 이름 목록을 배열로 리턴하다.

### Object.freeze(객체)

객체를 수정할 수 없는 상태로 변경한다.

객체 내부 값을 변경할 수 없게 된다.

### 클래스 선언

생성자의 이름은 언제나 constructor 이다.

javascript 메소드에서 this를 생략할 수 없다.

javascript 멤버 변수와 메소드는 모두 public 이다.

private 선언 문법이 없다.

javascript 언어는 변수의 타입, 파라미터 변수의 타입, 리턴 타입을 명시할 수 없다.

### static 메소드

메소드 앞에 static 키워드를 붙이면, 그 메소드는 static 메소드가 된다.

static 메소드의 특징은, 현재 객체를 의미하는 this 키워드를 사용할 수 없다는 점이다.

this 키워드를 사용할 수 없기 때문에,

this 객체의 멤버 변수나, this 객체의 메소드를 호출할 수 없다.

this 객체가 아닌 다른 객체의 멤버 변수나 메소드를 사용하는 것은 가능하다.

class Rectangle extends Figure {

Figure 클래스를 상속하여 Rectangle 클래스를 구현하였다.

부모 클래스의 멤버 변수와 메소드가 자식 클래스에 상속된다.

super(x1, y1);

부모 클래스의 생성자를 호출한다.

super.move(dx, dy);

부모 클래스의 move 메소드를 호출한다.

### getter / setter

getter / setter 매소드를 구현하여 속성을 정의한다.

getter / setter 매소드로 구현된 속성을 외부에서 사용할 때,

마치 멤버 변수인 것 처럼 사용할 수 있다.

# 과제

### personArray1.js

비어있는 persons 배열을 생성한다.

for 루프의 본문에서

persons 배열의 끝에 { name: "홍길동", age: 16 + i } 객체를 추가한다.

for 루프를 10번 반복한다.

persons 배열을 출력한다.

### personArray2.js

personArray1.js와 동일한 방법으로 persons1 배열을 생성한다.

persons1 배열을 복제해서 persons2 배열을 생성한다.

persons2 배열을 새로 배열을 만들어서, persons1 배열의 값들을 복사.

deep copy가 되도록 복제해야 한다.

즉 배열만 복제 되는 것이 아니고, 배열이 참조하는 객체들도 복제 되어야 한다.

persons2 배열을 출력한다.