JavaScript

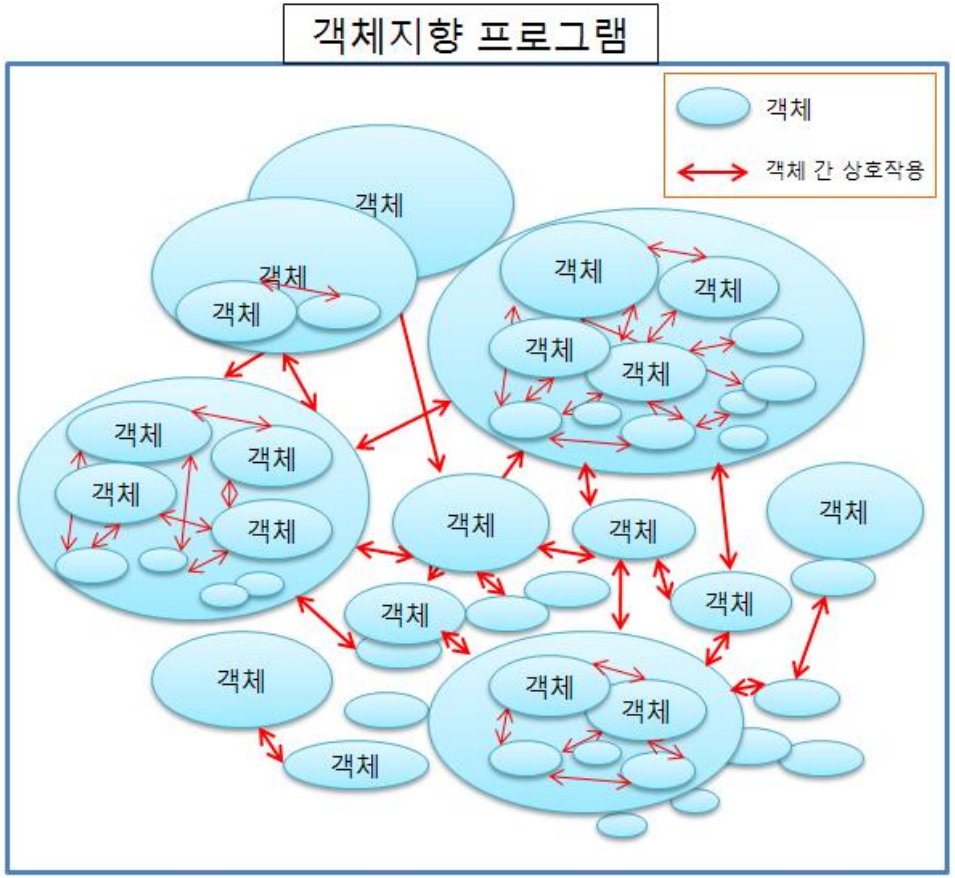
객체 – 김근형 강사

객체

객체(Object)란

객체

- 객체란?
  - **현실의 투영**
  - 현실에 존재하는 명사형으로 지칭될 수 있는 모든 오브젝트들을 코드로 투영하기 위해 사용되는 것.
  - 실제 스크립트에서 관리하기 어려운 변수나 값들의 집합을 객체화 하여 관리하는 경우가 많다.
  - 객체는 객체 간의 상호작용을 통해 복잡한 프로그램을 좀 더 효율적으로 설계할 수 있다.



객체

- 자바 스크립트에서의 객체
  - 자바 스크립트의 모든 참조 타입은 객체가 기본 골조로 이루어져 있다.
  - 객체는 단순히 '이름(key):값(value)' 형태의 프로퍼티들을 저장하는 컨테이너
  - 참조타입인 객체는 여러 개의 프로퍼티들을 포함할 수 있다.
  - 이러한 객체의 프로퍼티는 기본 타입의 값을 포함하거나, 다른 객체를 가리킬 수도 있다.
  - 프로퍼티의 성질에 따라 객체의 프로퍼티는 함수로 표현할 수 있으며 자바스크립트에서는 이러한 프로퍼티를 메서드라고 부른다.

객체

- 객체 생성
  - 자바 스크립트는 클래스라는 개념이 없고 객체 리터럴이나 생성자 함수 등 별도의 생성 방식이 존재한다.
  - 자바 스크립트에서 객체를 생성하는 방식
    - 기본 제공 Object() 객체 생성자 함수를 이용하는 방법
    - 객체 리터럴을 이용하는 방법
    - 생성자 함수를 이용하는 방법

객체

- Object() 생성자 함수 이용

```
// Object()를 이용해서 foo 빈 객체 생성
var foo = new Object();

// foo 객체 프로퍼티 생성
foo.name = 'foo';
foo.age = 30;
foo.gender = 'male';

console.log(typeof foo); // object
console.log(foo); // { name: 'foo', age: 30, gender: 'male' }
```

```
object
▶ {name: "foo", age: 30, gender: "male"}
```

객체

- 객체 리터럴 방식

```
// 객체 리터럴 방식으로 foo 객체 생성
var foo = {
    name : 'foo',
    age : 30,
    gender: 'male'
};

console.log(typeof foo); // object
console.log(foo); // { name: 'foo', age: '30', gender: 'male' }
```

```
object
▶ {name: "foo", age: 30, gender: "male"}
```

객체

- 객체 프로퍼티 읽기/쓰기/갱신
  - 객체 프로퍼티에 접근하려면 다음과 같은 방법을 사용한다.
    - 대괄호([]) 표기법

      objectName[property name]

    - 마침표(.) 표기법

      objectName.property name

```
// 객체 리터럴 방식을 통한 foo 객체 생성
var foo = {
    name : 'foo',
    major : 'computer science'
};

// 객체 프로퍼티 읽기
console.log(foo.name); // foo
console.log(foo['name']); // foo
console.log(foo.nickname); // undefined

// 객체 프로퍼티 갱신
foo.major = 'electronics engineering';
console.log(foo.major); //electronics engineering
console.log(foo['major']); //electronics engineering

// 객체 프로퍼티 동적 생성
foo.age = 30;
console.log(foo.age); // 30

// 대괄호 표기법만을 사용해야 할 경우
foo['full-name'] = 'foo bar';
console.log(foo['full-name']); // foo bar
console.log(foo.full-name); // NaN
console.log(foo.full); // undefined
console.log(name); // undefined
```

객체 대상 명령문

- for ~ in
  - for ~ in 문은 해당 객체의 모든 열거할 수 있는 프로퍼티(enumerable properties)를 순회할 수 있도록 도와주는 명령어이다.
  - 열거할 수 있는 프로퍼티란 내부적으로 enumerable 플래그가 true 로 설정된 프로퍼티를 의미한다.

```
[구문]
For ( variable in enumerableObject ) { 코드 }
```

객체 대상 명령문

- for ~ in
  - for in 문을 사용하면 객체에 포함된 모든 프로퍼티에 대해 루프를 수행할 수 있다.

```
// 객체 리터럴을 통한 foo 객체 생성
var foo = {
    name: 'foo',
    age: 30,
    major: 'computer science'
};

// for in문을 이용한 객체 프로퍼티 출력
var prop;
for (prop in foo) {
    console.log(prop, foo[prop]);
}
```

```
name foo
age 30
major computer science
```

객체 대상 명령문

- in
  - in 키워드는 특정 속성 또는 메소드가 그 객체에 있는지 판별한다.

```
var product = {
    name: '7D 건조 망고',
    kind: '당절임',
    ingredient: '망고, 설탕, 메타중아황산나트륨, 치자황색소',
    origin: '필리핀',

    getName: function() {
        return this.name;
    }
};

console.log('name' in product); // true
console.log('shop' in product); // false
console.log('getName' in product); // true
```

객체 대상 명령문

- with
  - with 키워드는 product.name 처럼 써야할 것을 name만 사용할 수 있게 도와준다.

```
var product = {
    name: '7D 건조 망고',
    kind: '당절임',
    ingredient: '망고, 설탕, 메타중아황산나트륨, 치자황색소',
    origin: '필리핀',

    getName: function() {
        return this.name;
    }
};

with(product) {
    console.log(name); // 7D 건조 망고
    console.log(kind); // 당절임
    console.log(ingredient); // 망고, 설탕, 메타중아황산나트륨, 치자황색소
    console.log(origin); // 필리핀
    console.log(getName()); // 7D 건조 망고
}
```

객체 대상 명령문

- 객체 프로퍼티 삭제
  - 자바 스크립트에서 delete 연산자를 이용해 객체 프로퍼티를 삭제할 수 있다.
  - 단 delete는 프로퍼티를 삭제할 수 있을 뿐 객체 자체를 삭제할 순 없다.

```
// 객체 리터럴을 통한 foo 객체 생성
var foo = {
    name: 'foo',
    nickname: 'babo'
};

console.log(foo.nickname);  // babo
delete  foo.nickname;  // nickname 프로퍼티 삭제
console.log(foo.nickname);  // undefined

delete foo;  // foo 객체 삭제 시도
console.log(foo.name);  // foo
```

기본형과 참조형

- 기본형과 참조형
  - 기본형과 참조형은 메모리 내에서 상당히 다른 움직임을 보여준다.
  - 기본형은 값의 복사가 일어나지만 참조형은 메모리의 참조가 일어나므로 잘 구분해야 한다.

기본형과 참조형

- 기본형

```
var a;
a = 10;
```

변수명	a			
주소	@313			

주소	...	313	314	315	316	317	...
데이터		10					

```
var a;   var b = 'abc';
a = 10;
```

변수명	a	b		
주소	@313	@314		

주소	...	313	314	315	316	317	...
데이터		10	'abc'				

```
var a;   var b = 'abc';
a = 10;  b = false;
```

변수명	a	b		
주소	@313	@314		

주소	...	313	314	315	316	317	...
데이터		10	false				

```
var a;   var b = 'abc';  var c = b;
a = 10;  b = false;
```

변수명	a	b	c	
주소	@313	@314	@315	

주소	...	313	314	315	316	317	...
데이터		10	false	false			

```
var a;   var b = 'abc';  var c = b;
a = 10;  b = false;      c = 20;

         b ≠ c
```

변수명	a	b	c	
주소	@313	@314	@315	

주소	...	313	314	315	316	317	...
데이터		10	false	20			

기본형과 참조형

- 참조형

```
var obj = {
  a: 1,
  b: 'b'
};
```

변수명	obj
주소	@413

주소	...	413	414	...	1011	1012	1013	...
데이터		@1011			{ a: @1012, b: @1013 }	1	'b'	

```
var obj = {
  a: 1,
  b: 'b'
};
var obj2 = obj;
```

변수명	obj	obj2
주소	@413	@414

주소	...	413	414	...	1011	1012	1013	...
데이터		@1011	@1011		{ a: @1012, b: @1013 }	1	'b'	

```
var obj = {
  a: 1,
  b: 'b'
};
var obj2 = obj;
obj2.a = 10;
console.log(obj2.a);    // 10
console.log(obj.a);     // 10
```

변수명	obj	obj2
주소	@413	@414

주소	...	413	414	...	1011	1012	1013	...
데이터		@1011	@1011		{ a: @1012, b: @1013 }	10	'b'	

기본형과 참조형

- 기본형과 참조형 예제 실습

```
var objA = {
    val : 40
};
var objB = objA;

console.log(objA.val);  // 40
console.log(objB.val);  // 40

objB.val = 50;
console.log(objA.val);  // 50
console.log(objB.val);  // 50
```

```
var a = 100;
var b = 100;

var objA = { value: 100 };
var objB = { value: 100 };
var objC = objB;

console.log(a == b);  // true
console.log(objA == objB); // false
console.log(objB == objC); // true
```

기본형과 참조형

- 참조형에서 기본형 데이터를 밀어넣을 경우

```
var obj3 = {
  a: [4, 5, 6]
};
```

변수명	obj3			
주소	@547			

주소	...	547	...	1184	1185	...	1326	1327	1328	...
데이터		@1184		{ a: @1185 }	[@1326, @1327, @1328]		4	5	6	

```
var obj3 = {
  a: [4, 5, 6]
};
obj3.a = 'new';
```

G.C 대상

변수명	obj3			
주소	@547			

주소	...	547	...	1184	1185	...	1326	1327	1328	...
데이터		@1184		{ a: @1185 }	'new'		4	5	6	

Property name

- 프로퍼티 이름 조합
  - 문자열과 변수를 조합하여 오브젝트의 프로퍼티 이름으로 사용할 수 있다.
  - 이를 프로퍼티 이름 조합(Computed property name)이라고 부른다.
  - 문자열 조합은 아래와 같이 사용이 가능하다.

```
let item = {
  ["one" + "two"]: 12
};
console.log(item.onetwo);
```

```
12
```

Property name

- 프로퍼티 이름 조합
  - 변수 값과 문자열 조합

```
let item = "tennis";
let sports = {
  [item]: 1,
  [item + "Game"]: "윔블던",
  [item + "Method"](){
    return this[item];
  }
};
console.log(sports.tennis);
console.log(sports.tennisGame);
console.log(sports.tennisMethod());
```

```
1
윔블던
1
```

Property name

- 프로퍼티 이름 조합
  - Distructuring과 property 이름 조합

```
let one = "sports";
let {[one]: value} = {sports: "농구"};
console.log(value);
```

농구

Computed Property Name

- Computed Property Name
  - computed property name은 표현식(expression)을 이용해 객체의 key 값을 정의하는 문법이다.
  - 속성명에 표현식을 사용하기 위해 대괄호(" [ ] ") 안에 표현식을 쓴다.
  - 표현식은 변수가 들어올 수도 있고, 함수가 들어올 수도 있다.

```
var i = 0;
var a = {
  ["foo" + ++i]: i,
  ["foo" + ++i]: i,
  ["foo" + ++i]: i
};

console.log(a);
```

```
▼ {foo1: 1, foo2: 2, foo3: 3}
    foo1: 1
    foo2: 2
    foo3: 3
  ▶ __proto__: Object
```

Computed Property Name

- Computed Property Name
  - 프로퍼티 이름 예제

```
var name1 = "장동건";
var name2 = "원빈";
var num = 1;

var obj = {
    [num+") "+name1]:"반장",
    [num+1+") "+name2]:"부반장"
};

console.log(obj);
```

```
▼{1) 장동건: "반장", 2) 원빈: "부반장"}
    1) 장동건: "반장"
    2) 원빈: "부반장"
  ▶ __proto__: Object
```

Computed Property Name

- Computed Property Name
  - 함수 이름 예제

```
function add(a, b){
    return a+b;
}
function conf(a, b){
    return `${a} + ${b}`;
}
var obj = {
    ["addNumber "+conf(5, 10)]:add(5, 10),
    [add(10, 20)+" is Result by"]:"10+20"
}
console.log(obj);
```

```
▼{addNumber 5 + 10: 15, 30 is Result by: "10+20"}
    addNumber 5 + 10: 15
    30 is Result by: "10+20"
  ▶__proto__: Object
```

Shorthand Property and Method Names

- 프로퍼티 이름 축약(Shorthand Property Names)
  - 일반 변수에서 객체로 값을 할당하고자 할 경우 변수의 이름과 객체 프로퍼티의 이름이 똑같다면 이름에 대한 축약이 가능해진다.

```
// ES5
// var x = 10; var y = 20;
// var obj = { x: x, y: y };

// ES6
var x = 10; var y = 20;
var obj = { x, y };

console.log(obj); // {x: 10, y: 20}
```

Shorthand Property and Method Names

- 메서드 이름 축약(Shorthand Method Names)
  - 일반적인 메서드는 프로퍼티 명 옆에 반드시 function(){} 을 써 주어야만 하였다.
  - 하지만 ES6 이후에 function 키워드와 : 를 축약할 수 있게 되었다.

```
// ES5
var obj = {
    a : function(){
        console.log("method a");
    },
    b : function(){
        console.log("method b");
    }
}
```

```
// ES6
var obj = {
    a (){ console.log("method a"); },
    b (){ console.log("method b"); }
}
obj.a();
obj.b();
```

Getter Setter

- Getter Setter
  - ECMA 2015 버전이 등장하면서 객체에 Getter와 Setter라는 개념이 등장하였다.
  - Getter는 객체의 프로퍼티를 가져오는 함수를 말하며 Setter는 객체의 프로퍼티를 설정하는 함수를 말한다.
  - Getter와 Setter를 사용하는 이유는 다음과 같다.
    - getter와 setter 를 사용해 정보 은닉이 가능하다.
    - 직접적인 접근의 방지가 가능하다.
    - 일관성 있는 코드가 만들어지며 코드의 가독성이 좋아진다.

Getter Setter

- Getter 사용 방법

```
{get prop() { ... } }
{get [expression]() { ... } }  // ES6에서 추가
```

  - 동적으로 계산이 필요한 프로퍼티 값을 가져와야 할 때, getter를 사용한다면 별도의 함수를 만들 필요가 없다.
  - getter를 사용할 때 유의점
    - 식별자로 숫자와 문자를 모두 사용할 수 있다.
    - 파라미터가 없어야 한다.
    - 리터럴 객체의 같은 이름의 get이나 동일한 이름의 프로퍼티를 가질 수 없습니다. ({ get x() {}, get x() {} }, { x : ..., get x() {} } 불가능)

Getter Setter

- Getter 특징
  - getter는 객체를 초기화 할 때 선언 할 수 있다.

```
var log = ['test'];
var obj = {
    get latest () {
        if (log.length == 0) return undefined;
        return log[log.length - 1]
    }
}
console.log(obj.latest); // Will return "test".
```

Getter Setter

- Getter 특징
  - getter는 삭제가 가능하다.

```
var log = ['test'];
var obj = {
    get latest () {
        if (log.length == 0) return undefined;
        return log[log.length - 1]
    }
}
console.log(obj.latest);
delete obj.latest;
console.log(obj.latest);
```

Getter Setter

- Getter 특징
  - 객체가 이미 존재 할 때, defineProperty 메소드로 getter를 정의할 수 있다.

```
var o = { a:0 }

Object.defineProperty(o, "b", { get: function () { return this.a + 1; } });

console.log(o.b) // Runs the getter, which yields a + 1 (which is 1)
```

  - 동적으로 getter 이름을 정의 할 수 있다.

```
var expr = "foo";

var obj = {
    get [expr]() { return "bar"; }
};

console.log(obj.foo); // "bar"
```

Getter Setter

- Getter 장점
  - 계산 미루기 (Lazy getter)
    - getter의 값 계산은 실제 값이 필요할 때 이루어지고, 값이 필요하지 않다면, 계산을 하지 않는다. 즉 값이 필요하지 않다면 쓸데없는 계산을 하지 않아 cpu를 낭비하지 않게 된다.
  - 캐싱 (Smart/Memorized getter)
    - 값은 getter가 호출될 때 처음 계산되며 캐싱된다. 이후의 호출은 다시 계산하지 않고 이 캐시 값을 반환한다.
    - 이러한 캐싱은 다음과 같은 경우에 유용하다.
      - 값의 계산 비용이 큰 경우. (RAM이나 CPU 시간을 많이 소모하거나, worker thread를 생성하거나, 원격 파일을 불러오는 등)
      - 값이 당장 필요하지 않지만 나중에 사용되어야 할 경우(혹은 이용되지 않을 수도 있는 경우)
      - 값이 여러 번 이용되지만 변경되지 않아 매번 계산할 필요가 없는 경우

Getter Setter

- Getter 사용 시 유의사항
  - getter은 값을 캐싱하고 있기 때문에 아래와 같은 경우, 사용에 유의해야 한다.

```
var o = {
  set foo (val) {
    delete this.foo;
    this.foo = val;
  },
  get foo () {
    delete this.foo;
    return this.foo = 'something';
  }
};

o.foo = "test";
console.log(o.foo);
```

```
test
```

Getter Setter

- Setter 사용 방법

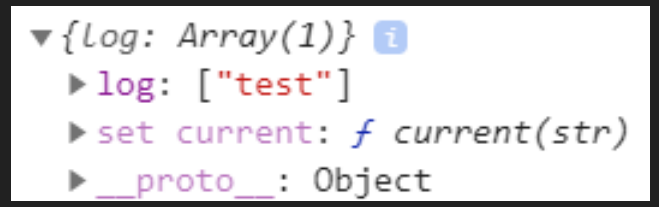
```
{set prop(val) { ... } }
{set [expression](val) { ... } }  // ES6에서 추가
```

  - 동적으로 계산이 필요한 프로퍼티 값을 가져와야 할 때, setter를 사용한다면 별도의 함수를 만들 필요가 없다.
  - setter를 사용할 때 유의점
    - 식별자로 숫자와 문자를 모두 사용할 수 있다.
    - 한개의 파라미터만 가질 수 있습니다.
    - 리터럴 객체의 같은 이름의 set이나 동일한 이름의 프로퍼티를 가질 수 없다. ({ set x() {}, set x() {} }, { x : ..., set x() {} } 불가능)

Getter Setter

- Setter 특징
  - setter는 객체를 초기화 할 때 선언 할 수 있다.

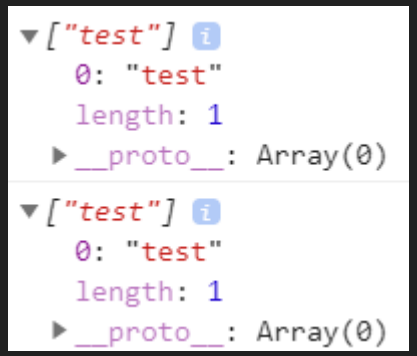
```
var o = {
    set current (str) {
        this.log[this.log.length] = str;
    },
    log: []
}
o.current = "test";
console.log(o);
```



Getter Setter

- Setter 특징
  - setter는 삭제가 가능하다.

```
var o = {
    set current (str) {
        this.log[this.log.length] = str;
    },
    log: []
}
o.current = "test";
console.log(o.log);
delete o.current;
o.current = "test2";
console.log(o.log);
```



Getter Setter

- Setter 특징
  - 객체가 이미 존재 할 때, defineProperty 메소드로 setter를 정의 할 수 있다.

```
var o = { a:0 };

Object.defineProperty(o, "b", { set: function (x) { this.a = x / 2; } });

o.b = 10; // Runs the setter, which assigns 10 / 2 (5) to the 'a' property
console.log(o.a) // 5
```

Getter Setter

- Setter 특징
  - 동적으로 setter 이름을 정의 할 수 있다.

```
var expr = "foo";

var obj = {
  baz: "bar",
  set [expr](v) { this.baz = v; }
};

console.log(obj.baz); // "bar"
obj.foo = "baz";      // run the setter
console.log(obj.baz); // "baz"
```

Getter Setter

- defineProperty로 Getter, Setter 한꺼번에 선언하기

```
const person = {};

Object.defineProperty(person, 'name', {
    set(value) { this.name1 = `${value}-set`; },
    get() { return `get-${this.name1}`; },
});

person.name = 'sungin'; // set을 통해 sungin-set 으로 설정
console.log(person.name);  // get을 통해 get-sungin-set 으로 설정
```

Optional Chaining

- Optional Chaining
  - 객체 내에 없는 정보를 참조하려고 할 경우 에러가 날 수 있다.
  - 보통 이러한 에러를 해결하기 위해 기존에는 && 연산자를 사용하여 해결하였지만 로직이 길어지는 단점이 존재했다.

```
let user = {}; // 주소 정보가 없는 사용자

alert(user.address.street); // TypeError: Cannot read property 'street' of undefined

alert( user && user.address && user.address.street ); // undefined, 에러가 발생 안함
```

Optional Chaining

- Optional Chaining
  - 옵셔널 체이닝을 활용한 메서드 접근

```
let user1 = {
    admin() {
        alert("관리자 계정입니다.");
    }
}

let user2 = {};

user1.admin?.(); // 관리자 계정입니다.
user2.admin?.();
```

Optional Chaining

- Optional Chaining
  - 옵셔널 체이닝과 []를 활용한 프로퍼티 접근

```
let user1 = {
    firstName: "Violet"
};

let user2 = null;

let key = "firstName";

alert(user1?.[key]); // Violet
alert(user2?.[key]); // undefined

alert(user1?.[key]?.something?.not?.existing); // undefined
```