JavaScript 객체지향의 기본 개념

변 석우 (swbyun@ks.ac.kr)

1 JavaScript 객체지향성의 특징

- Java, C++ 등 기존 널리 사용되는 객체지향 프로그래밍 언어는 클래스 기반(class-based) 언어이다. 이 언어들 객체지향성의 핵심은 클래스이다. 클래스의 템플레이트(template) 기능에 따라 객체를 생성하고, 클래스들 간의 super - subclass 관계를 형성함으로써 상속을 정의한다.
- JavaScript도 객체지향 언어이기는 하지만, JavaScript는 프로토타입 기반 언어 (prototype based language)로서, 클래스 기반의 언어와는 차이가 있다. 프로토타입 언어의 핵심 개념은 객체이다. 모든 것은 객체로 정의되며, 한 객체를 기반으로 새로운 객체를 생성하고, 객체들 간의 프로토타입 체인(prototype chain)을 형성함으로써 상속 관계를 표현한다. 프로토타입 기반 언어에서 클래스는 필수적인 개념이 아니다 (class-less). 그러나 JavaScript의 ES6 에서는 프로토타입 기반 위에 class를 사용할 수 있도록 sugaring 된 환경을 제공하고 있다.
- 클래스 기반 언어의 객체 생성은 템플릿 기능을 하는 class를 정의한 후, 클래스에 new를 적용함으로써 이루어진다. 한 클래스로부터 생성되는 모든 객체들은 동일한 구조를 갖는다. 탬플릿 기반으로 생성되는 객체는 정적인 특성을 가지며 (즉, 생성된 객체의 구조가 변하지 않음), 객체는 클래스의 인스턴스(instance) 이다.
- 프로토타입 기반 언어인 JavaScript의 객체 생성 과 상속(inherit) 과정은 기존 클래스 기반 언어와 다르다. 먼저, 객체를 생성하는 방법에 있어서 일정한 형태가 아닌 다양한 방법이 적용될 수 있다. 특히, 프로토타입 체인에 기반한 상속 과정은 클로저(closure), this 등과 연관되어 있어, 이를 이해하는 데 어려움이 있을 수 있다. 아래의 여러 예를 통하여 좀 더 구체적인 논의를 진행하기로 한다.
- JavaScript 언어는 처음부터 클래스 없이 설계되었으며, ES5까지는 class를 사용하지 않다가, ES6에 class 기능이 추가되었다. 그러나 이 class 때문에 언어의 기본 개념이 바뀐 것은 아니며, 기존 개념을 그대로 유지하면서, class를 sugaring 하는 방법을 사용한다.

2 Java의 객체지향성: 예제 프로그램

- JavaScript의 객체지향성을 이해하는 방법으로서, Java의 객체지향 프로그램 코드가 JavaScript에서 어떤 형태로 표현될 수 있는 지를 설명하고자 한다.
- 다음은 Java에서 `Person` 클래스를 정의한 후, 클래스의 템플릿 기능에 따라, `new` 에 의해서 생성되는 `person` 객체의 생성 및 실행 과정을 보여주는 예이다.
- 클래스 내에 `firstName, lastName, getName, getLastName` 등의 프로퍼티들이 정의되어 있으며, 이 클래스로부터 `new` 를 적용하여 생성되는 객체들은 모두 동일한 구조를 갖는다.

```
public class Person {
  String firstName;   String lastName;
  Person(String firstName, String lastName) {
    this.firstName = firstName;   this.lastName = lastName;
  }
  String getName() { return this.lastName + " " + this.firstName; }
  String getLastName() { return "Mr. " + this.lastName; }

  public static void main(String[] args) {
```

```
    Person person = new Person("Gildong", "Hong");
    System.out.println(person.getName());      // Hong Gildong
    System.out.println(person.getLastName()); // Mr. Hong
  }
}
```

- 다음 절에서는 이와 동등한 의미를 갖는 객체지향 프로그램을 JavaScript로 표현한다. Java와는 달리, JavaScript에서는 여러 다양한 방법이 적용될 수 있다.

3 JavaScript에서 객체를 생성하는 다양한 방법

3.1 리터럴(Literal) 문법에 의한 생성

- 자바스크립트의 객체는 key: value 형태로 된 프로퍼티들의 집합이다. 아래의 예는 객체를 직접 코딩으로 생성하는 방법으로서, 가장 단순하면서도 자주 사용되는 방법이다.
- 위의 Java와 동일한 구조의 프로퍼티를 갖는 person 객체를 코딩한다. 여기서는 템플릿 기능 없이 (즉, new를 사용하지 않음) firstName, lastName, getName, getLastName 프로퍼티를 직접 객체로서 표현한다.
- 이러한 리터럴에 의한 객체 생성은 오직 한 개의 객체를 정의하므로, 여러 개의 객체를 생성할 때 번거로움이 있다. 동일한 구조의 여러 객체를 탬플릿과 new 에 의해 쉽게 생성하는 방법과는 대조적이다.

```
// object1.js
  let person = {
    firstName: 'Gildong',
    lastName: 'Hong',
    getName: function() {return this.lastName + ' ' + this.firstName;},
    getLastName: function getLastName() {return 'Mr. ' + this.lastName; }, // ES5
    // getLastName() {return 'Mr. ' + this.lastName; },    // ES6
  };

  console.log(person.getName());      // Hong Gildong
  console.log(person.getLastName());  // Mr. Hong
```

- 위의 예에서, 두 key getName, getLastName 에 해당되는 값(value)이 함수 로 표현된 점에 주목할 필요가 있다. ES6부터 키와 값의 이름이 동일할 경우, exp: exp 형태 대신, 단순히 exp 만으로 표현할 수 있다. 위의 예서는 getLastName 의 정의를 getLastName: getNastName 형태와 단순 표현식의 두 가지 형태로 보여 준다.

3.2 동적으로 변화하는 객체

- 클래스 기반 언어와는 달리, JavaScript의 객체는 동적으로(dynamically) 변화하는 특징을 갖고 있다. 이 기능을 이용하여, 위의 person 객체는 다음과 같이 동적으로 수정되는 형태로 표현될 수도 있다.

```
// object2.js
  // let person = new Object();
  let person = {};
  person.firstName = 'Gildong';
  person.lastName = 'Hong';
  person.getName = function() {return this.lastName + ' ' + this.firstName;};
  person.getLastName = function () {return 'Mr. ' + this.lastName; };

  console.log(person.getName());      // Hong Gildong
  console.log(person.getLastName());  // Mr. Hong
```

- 위의 person 객체는 맨 처음 emtpy 상태로 시작하였으며, 프로그램 실행을 통해 새로운 프로퍼티들이 추가되는 형태로 완성되었다.

3.3 생성자 함수(Constructor Functions)와 new를 적용한 객체 생성

- JavaScript의 함수는 일반 함수의 기능과 함께, 객체 생성(costructor) 기능을 갖는 특징이 있다. 생성자 함수를 정의한 후, 이 함수에 대한 new를 적용하여 객체를 생성할 수 있다. 개념적으로 생성자 함수는 class와 동일하다. 단지 문법(syntax)적으로 class라는 사용하지 않을 뿐이다.

- 일반적인 기능의 함수와 생성자 함수를 구분하여 표현하기 위해서, 관례적으로 일반 함수의 이름은 영문 소문자, 생성자 함수의 이름은 영문 대분자로 시작하는 이름을 사용한다.

```
// object3.js
  function Person(fname, lname) {
    this.firstName = fname;
    this.lastName = lname;
    this.getName = function() {return this.lastName + ' ' + this.firstName;};
    this.getLastName = () => 'Mr. ' + this.lastName; // arrow (lambda) function
  }

  const person = new Person('Gildong', 'Honge');  // instance creation
  console.log(person.getName());      // Hong Gildong
  console.log(person.getLastName());  // Mr. Hong
```

- 생성자 함수에서는 프로퍼티를 정의하기 위해서, this.firstName 처럼 각 프로퍼티에 this 를 사용한다.

- 위의 예에서 getLastName 을 화살표 함수(arrow function, lambda function) 로 정의하였다. 함수를 정의하는 방법은 기존의 function 키워드를 사용하는 방법과 화살표 함수로 정의하는 두 가지 방법이 있는데, 상황에 따라 원하는 방식을 적용하면 된다.

- 화살표 함수는 표현이 간결하여 일반 함수 대신 널리 사용되고 있으나, 화살표 함수의 this 값의 영역(scope) 은 일반 함수와 다르므로, 이 점에 유의해야 한다 (다른 관련 자료 참고)

- 화살표 함수는 람다 식(lambda expression) 이라고도 부르는데, 이는 함수를 수식(expression) 으로 표현할 수 있음을 의미한다. 개념적으로, 함수형 프로그래밍에서는 함수도 값 으로서, 문법적으로 값은 수식으로 표현된다. 언어 문법의 계층 구조 관점에서 볼 때, 수식(expressions)은 가장 낮은 계층에 속하는 단위이다. 일반적인 프로그래밍 언어 문법 구조에서, 수식들이 모여서 문장(statements)을 이루고, 문장들이 모여서 함수(functions)를 구성한다. 수식은 하위 계층에, 함수는 상위 계층에 속하는 구조이다. 그런데, 함수를 화살표 함수로 표현한다는 것은, 구문적으로, 함수를 수식으로 표현함을 의미한다. 이 수준은 변수 등과 같은 수준으로서, 마치 변수처럼, 함수가 배열 등에 저장될 수 있고, 다른 함수의 인수로 사용될 수도 있고, 복귀 값(return value) 으로 이용될 수 있다. 문법적으로 함수의 표현이 제약없이 매우 자유롭게 이루어질 수 있음을 의미한다. 이런 개념을 함수가 일등급 시민(functions are first-class citizens) 이라고 부른다. 또한 이런 형태로 표현되는 함수를 고차함수(higher-order functions)라고도 한다. JavaScript는 고차함수를 이용하고 있으며, JavaScript에서 함수는 일등급 시민이다.

3.4 프로토타입(Prototype) 적용

- JavaScript에서 모든 것은 객체로 표현된다. 함수도 객체로 표현되는데, 특이한 점은, 함수에 대한 객체는 한 개가 아닌 두 개의 객체로 표현된다는 것이다 (이 함수는 일반 함수를 의미하며, 화살표 함수는 여기에 해당되지 않음). 함수 그 자체를 표현하는 객체 (Fuction 타입)와 더불어, 그 함수의 프로타입(Prototype) 기능을 표현하는 Prototype 객체이다.

- 아래 Person 함수에 대해서 Person 객체 (Function 타입) 와 Person.prototype 객체 (Prototype 타입) 두 객체가 생성된다.

- 대략적으로, 함수에 대한 객체를 다룰 때는 Function 타입의 객체가 아닌, Prototype 객체를 중심으로 프로그래밍이 이루어진다. 객체의 생성 및 상속 등이 모두 프로토타입 개념을 위주로 진행되기 때문이다.

- 아래 프로그램에서는 `Person` 생성자가 정의된 후, `getName, getLastName` 이라는 두 프로퍼티(메소드)가 객체에 추가되는 예를 보여 준다. 이때 사용되는 객체는 `Person` 이 아닌, Persion.prototype 인 것에 유의해야 한다.

```
// object4.js
    function Person(fname, lname) {
      this.firstName = fname;
      this.lastName = lname;
    }

  // adding properties: getName, getLastName
    Person.prototype.getName = function() {
      return this.lastName + ' ' + this.firstName;
    };
    Person.prototype.getLastName = function () { return 'Mr. ' + this.lastName; }

    const person = new Person('Gildong', 'Honge');
    console.log(person.getName());      // Hong Gildong
    console.log(person.getLastName());  // Mr. Hong
```

3.5 Class 적용 (ES6)

- 2015년 ES6가 발표되면서, JavaScript에 Java와 유사한 기능의 class가 추가되었다. class가 추가되었다고 하더라도 프로토타입 기반 언어의 개념이 바뀐 것은 아니다. 위에서 언급한 대로, JavaScript는 이미 Java의 class와 유사한 기능을 하는 생성자 함수(constructor functions)를 가지고 있었으며, ES6의 class는 새로운 개념이라기 보다는 생성자 함수의 sugaring인 셈이다. 사용의 편리함을 위해 생성자 함수를 class 구문으로 포장한(wrap) 것에 불과하다. 실제로 class는 컴파일 과정에서 생성자 함수로 변환된다.

- 다음은 위의 생성자 함수와 프로토타입으로 코딩된 예를 class 문법으로 표현한 예이다.

```
// object5.js
  class Person{
    constructor(fname, lname) {
      this.firstName = fname;
      this.lastName = lname;
    }
    getName () {return this.lastName + ' ' + this.firstName;};
    getLastName () {return 'Mr. ' + this.lastName; }
  }

  const person = new Person('Gildong', 'Honge');
  console.log(person.getName());      // Hong Gildong
  console.log(person.getLastName());  // Mr. Hong
  person.newproperty = 1234;
  console.log(person.newproperty);  // 1234
```

- 생성자 함수와 비교하면, 명시적으로 `constructor` 함수를 사용하고 있으며, 프로퍼티를 정의할 때 `this` 를 생략하고 있다. 또한, class 내에 메소드를 내포하고 있으며, 이 메소드 또한 `this` 를 생략한다 (생성자 함수 또한 메소드들을 내포할 수 있는데, 이 경우 메소드 이름에 `this` 를 붙여야 한다).

- 위의 예에서, person 객체는 Person 클래스의 템플릿으로 생성되었지만, 새로운 프로퍼티 newproperty 가 추가 될 수 있음을 보여 준다.
- 실용적으로, 생성자 함수나 프로토타입 방식보다 class에 의한 표현이 간결하므로, class 사용을 추천한다. 그렇지만, 프로토타입 기술이 적용되는 경우가 있으므로, JavaScript에 대한 기본 개념을 충실히 이해하는 것이 필요하다.

4 __proto__ 객체 상속

4.1 객체들 사이의 상속

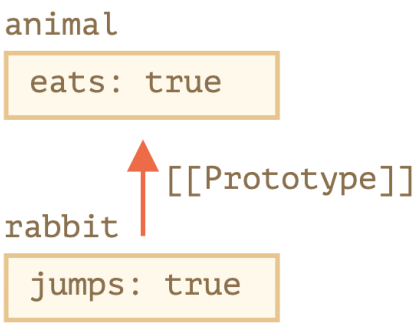
- 참고 문헌: https://javascript.info/prototype-inheritance
- 객체는 내부적으로 감추어진 특별한 프로퍼티인 [[Prototype]] 를 가지고 있는데, 프로그램에서는 이를 __proto__ 로서 접근할 수 있다.
- __proto__ 는 두 객체 사이의 상속 관계를 설정한다. 다음은 animal 객체가 rabbit 객체를 상속하는 상황을 코딩한 것이다.

```
// protoEx1.js
  let animal = { eats: true };
  let rabbit = { jumps: true };

  // rabbit object inherits animal object
  rabbit.__proto__ = animal;

  console.log(rabbit.eats); // true
  console.log(rabbit.jumps); // true
```

- 위의 예에서, rabbit 은 eats 프로퍼티를 갖지 않지만 animal 객체를 상속받아 그 객체에 있는 eats 를 공유할 수 있다. 이 상황은 다음 그림으로 표현될 수 있다.



- 객체 사이의 상속 관계를 객체 내에 표현할 수 있으며, 또한 하위 객체가 상위 객체에 내장된 함수를 호출할 수 있다.

```
// protoEx2.js
  let animal = {
    eats: true,
    walk() { console.log("Animal walk"); }
  };

  let rabbit = {
    jumps: true,
    __proto__: animal
```

```
};

rabbit.walk(); // Animal walk
```

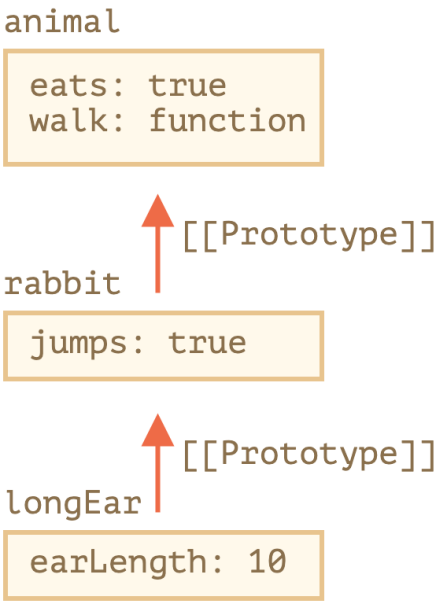
- 프로토타입에 의한 상속은 여러 객체들 사이에서 구성될 수 있으며, 이 상속 관계를 프로토타입 체인 (prototype chain) 이라 부른다.

```
// protoEx3.js
let animal = {
  eats: true,
  walk() { console.log("Animal walk"); }
};

let rabbit = {
  jumps: true,
  __proto__: animal
};

let longEar = {
  earLength: 10,
  __proto__: rabbit
};

longEar.walk(); // Animal walk
console.log(longEar.jumps); // true (from rabbit)
```



- 프로토타입 체인에서는 overrding이 적용된다. 다음 예에서 `rabbit` 의 `walk` 는 `animal` 의 `work` 를 overrding한다. 이때 선택 원리는, 현 객체로부터 시작하여 위쪽 방향으로, 가장 가까운 객체에 소속된 것이 선택된다.

```
// protoEx4.js
let animal = {
  eats: true,
  walk() { console.log("Animal walk") }
};
```

```
let rabbit = {
  __proto__: animal
};

rabbit.walk = function() {
  console.log("Rabbit! Bounce-bounce!");
};

animal.walk(); // Animal walk
rabbit.walk(); // Rabbit! Bounce-bounce!
```

4.2 프로토타입 체인에서 this 의 값(value)

- 객체지향에서 this 는 현재(current) 객체, 혹은 컨텍스트(context) 객체 이다. 즉, this 는 실행 환경의 객체를 의미하는데, 이 실행 환경은 동적으로 변화한다.
- 아래 예에서, walk(), sleep() 메소드 안에 있는 this.isSleeping 은 walk() 를 호출하는 객체가 animal 객체일 경우, 이는 animal.isSleeping 이며, rabbit 일 경우 rabbit.isSleeping 이 된다.
- rabbit.sleep() 이 실행 순서는 다음과 같다. rabbit 객체 내에 sleep() 메소드를 검색하는데, 이것이 없으므로 이 객체의 프로토타입 체인을 따라 위의 객체 animal 에서 sleep() 을 찾는데, 여기에 이 메소드가 존재하므로 이를 실행한다. 이때 중요한 점은 sleep() 이 animal 객체 안에 속해 있다고 하더라도, 이 메소드의 컨텍스트 객체는 rabbit 이다. 따라서 sleep() 메소드 안의 this.isSleeping = true; 는 rabbit.isSleeping = true; 이 된다.
- rabbit.isSleeping = true; 는 rabbit 객체 내에 isSleeping: true 형태의 프로퍼티를 추가함을 의미한다. 원래는 이 프로퍼티가 없었지만, rabbit.sleep() 을 실행함으로써 새로운 프로퍼티가 추가되었다.
- 비슷한 방법으로, animal.sleep() 을 실행하면 animal 객체에 isSleeping 프로퍼티가 추가된다. 아래 예에서, 이 메소드의 실행 전과 후에 animal.isSleeping 의 값이 변화되는 상황을 확인할 수 있다.

```
// protoEx5.js
let animal = {
  walk() {
    if (!this.isSleeping) { console.log(`I walk`); }
  },
  sleep() { this.isSleeping = true; }
};

let rabbit = {
  name: "White Rabbit",
  __proto__: animal
};

rabbit.sleep();  // new property added: rabbit.isSleeping = true

console.log(rabbit.isSleeping); // true
console.log(animal.isSleeping); // undefined (no such property in the prototype)

console.log(animal); // { walk: [Function: walk], sleep: [Function: sleep] }
animal.sleep();
console.log(animal.isSleeping); // true
```

```
  console.log(rabbit);   // { name: 'White Rabbit', isSleeping: true }
  console.log(animal);   // { walk: [Function], sleep: [Function], isSleeping: true }
```

4.3 객체지향 메소드의 파라메터 추론하기

- Java, JavaScript 등의 객체지향 프로그래밍에서는 `obj.method(x)` 형태의 메소드를 호출한다. 메소드란 함수의 변형된 표현으로서 객체(`obj`)를 중심으로 표현된 함수라고 생각할 수 있다. 이 표현은 객체지향이 아닌 일반 함수적인 표현에서는 `method(obj, x)` 형태의 이진 함수 호출에 해당된다. 만약 `obj.method()` 이라면 `method(obj)` 에 해당된다.
- 위의 예에서 `rabbit.sleep()` 형태의 메소드 호출은 함수의 관점에서 볼 때 `sleep(rabbit)` 에 해당된다.
- 함수적 관점에서 볼 때, `sleep() { this.isSleeping = true;}` 형태의 메소드 정의는 숨겨진 객체 파라메터를 명시적으로 표현한 `sleep(this) { this.isSleeping = true;}` 형태로 이해할 수 있다. 따라서 `sleep(rabbit)` 의 호출 결과는 `rabbit.isSleeping = true;` 가 됨을 쉽게 추론할 수 있다.
- JavaScript 프로그래밍에서는 종종 `this` 를 이해하는 데 어려움을 겪는다. 이때, 메소드를 함수적 관점으로 해석하면 이해하는 데 도움이 될 수 있다.
- 여기서 논의한 내용은 Java와 Python 등에서도 유사하게 적용될 수 있다.

5 예제 : Prototype (__proto__) 에 의한 상속

http://dmitrysoshnikov.com/ecmascript/javascript-the-core/

5.1 예제 프로그램 코드

```
'use strict';

function Foo(y) { this.y = y; }

Foo.prototype.x = 10;
Foo.prototype.calculate = function (z) {
  return this.x + this.y + z;
};

var b = new Foo(20);
var c = new Foo(30);

b.calculate(30); // 60
c.calculate(40); // 80

console.log(
  b.__proto__ === Foo.prototype, // true
  c.__proto__ === Foo.prototype, // true

  // also "Foo.prototype" automatically creates
  // a special property "constructor", which is a
  // reference to the constructor function itself;
  // instances "b" and "c" may found it via
  // delegation and use to check their constructor

  b.constructor === Foo, // true
```

```
  c.constructor === Foo, // true
  Foo.prototype.constructor === Foo, // true

  b.calculate === b.__proto__.calculate, // true
  b.__proto__.calculate === Foo.prototype.calculate // true
);
```

5.2 프로토타입 체인 다이어그램

- 위의 프로그램 코드에 해당하는 프로토타입 체인은 다음과 같은 다이어그램으로 나타낼 수 있다.