Prototype 과 JS통찰하기

자바스크립트의 철학

프로토타입 기반의 자바스크립트

Java와 같은 대부분의 객체지향 언어들은 이와 같은 플라톤의 이데아론의 철학적 관점을 중시합니다. 본질이 존재하고 특정 속성으로 대상을 분류(classification)합니다. 모든 사물이나 대상은 다른 것들로부터 구분할 수 없는 본질이 존재한다는 의견이 강했습니다. 실제 세계는 모든 본질이 구현된 하나의 실체라고 보았습니다. 그래서 우리가 사용하는 class 문법과 class 를 통한 인스턴스의 생성이라는 개념이 보편화된 것입니다.

하지만 자바스크립트는 다릅니다. 클래스 기반의 객체지향을 중시하는 것이 아니라, 프로토타입 이론을 중시합니다. 프로토타입 이론을 기반으로 하는 객체지향 언어의 특징은 다음과 같습니다.

- 개별 객체(instance) 수준에서 메소드와 변수를 추가
- 객체 생성은 일반적으로 복사를 통해 이루어짐
- 확장(extends)은 클래스가 아니라 위임(delegation)
  =>현재 객체가 메시지에 반응하지 못할 때 다른 객체로 메시지를 전달할 수 있게 하여 상속의 본질을 지원
- 개별 객체 수준에서 객체를 수정하고 발전시키는 능력은 선험적 분류의 필요성을 줄이고 반복적인 프로그래밍 및 디자인 스타일을 장려
- 프로토타입 프로그래밍은 일반적으로 분류하지 않고 유사성을 활용하도록 선택
- 결과적으로 설계는 맥락에 의해 평가

프로토타입 이론에 대해 이해하면 해당 특징들은 자연스럽게 받아들일 수 있습니다. 또한 다른 언어와 달리 자바스크립트만이 가지는 다양한 특징들(호이스팅, 렉시컬 스코프, 클로져, this 등)에 대해서 암기가 아닌 본질적인 이해를 통해 자바스크립트 자체에 대한 깊은 통찰이 가능하게 됩니다.

이제 자바스크립트의 철학은 어디서부터 시작되었는지 단계별로 알아보도록 합시다.

비트겐슈타인의 철학

> 공유 속성의 관점에서 정의하기 어려운 개념이 있다(사실상 올바른 분류란 없다) - 비트겐슈타인

현실에서 사용되는 언어에는 어떤 관념론적인 것이 숨어있지 않고, 다만 '사용'에 의해 언어가 규정되는 상황이 있을 뿐이라고 합니다. 즉, 그는 세계에 미리 내재되어서 대상과 언어를 완전히 규정하는 어떤 언어란 존재하지 않음을 말하고 있습니다.

비트겐슈타인은 '게임'을 통해 근거를 제시합니다. 게임은 일반적으로 승리와 패배가 명확합니다. 즉 '승리'와 '패배'라는 속성이 있다고 볼 수 있죠. 하지만 비트겐슈타인은 이에 대한 반론으로 '승리', '패배'가 없는 `ring around a rosy` 를 들고옵니다. 즉 일반적인 '분류'로 규정할 수 없는 대상도 존재한다고 주장합니다. 대신 모든 표현은 '흐름'과 관련이 있다고 주장합니다.

> 표현은 삶의 흐름 속에서만 의미를 갖는다 — 비트겐슈타인

의미사용이론 (The use theory of Meaning)

표현이란 사용(use)에 의해 의미(meaning)가 결정된다는 이론입니다. 단어의 쓰임새가 곧 의미가 됩니다. 즉, 단어의 '진정한 본래의 의미'란 존재하지 않고 '상황과 맥락에 의해서 결정된다'라고 주장하고 있습니다. 어떤 대상의 본질을 하나의 표현과 단어로 규정할 수 없다는 말과 같습니다.

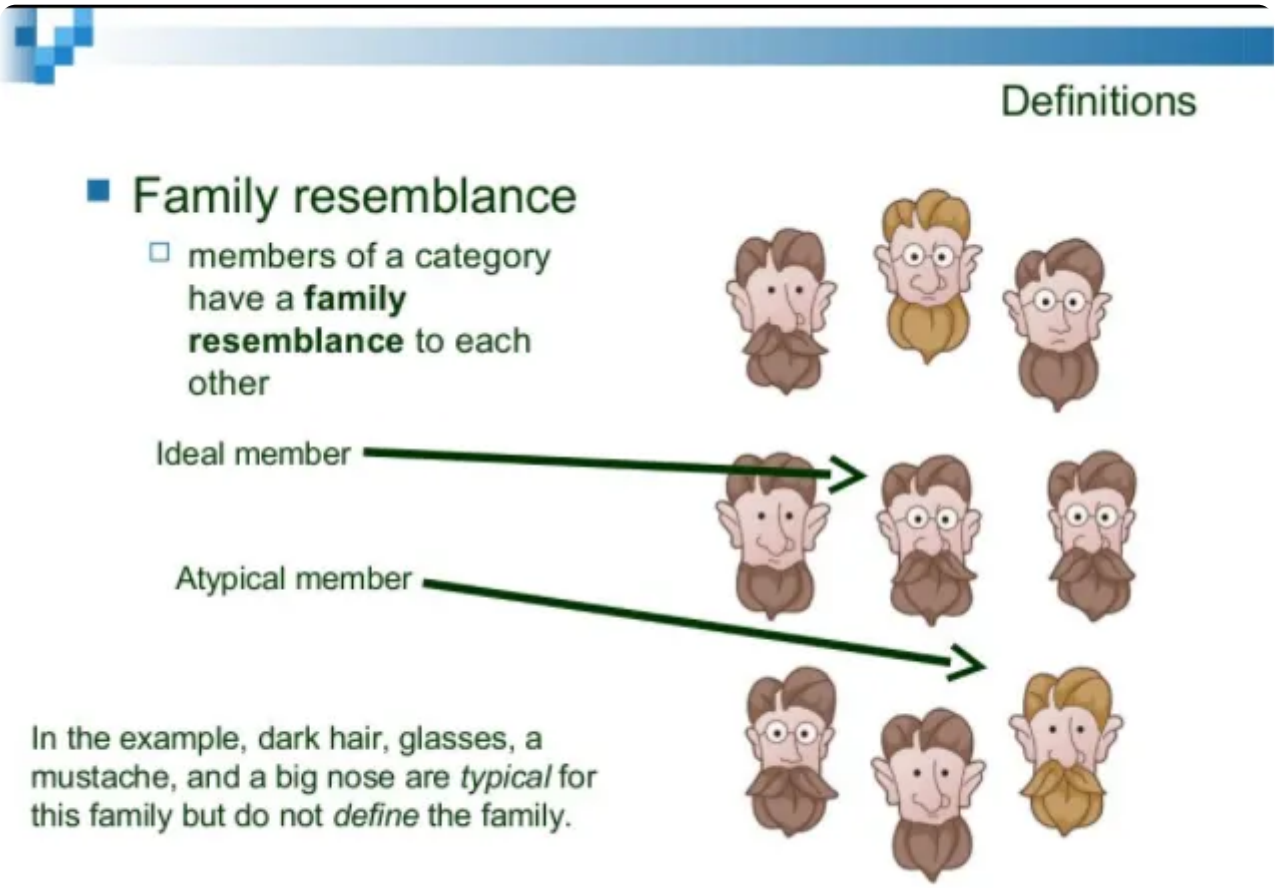
비트겐슈타인은 '벽돌'을 예로 들었습니다. 벽돌을 누가 어떤 상황에서 말하느냐에 따라 벽돌의 의미가 달라집니다.

- (벽돌이 필요할 때) : 벽돌을 달라
- (벽돌로 보수해야 할 때) : 벽돌을 채우라
- (벽돌이 떨어질 때) : 벽돌을 피해라

가족 유사성 (Family Resemblance)

비트겐슈타인은 의미사용이론과 더불어 가족 유사성 이론을 주장합니다.

> 인간이 현실에서 실제로 대상을 분류할 때 속성(전통적인 분류에서의 기준)이 아닌 가족 유사성을 통해 분류하게 된다. - 비트겐슈타인



위 그림처럼 가족이 모두 공유하는 공통된 속성은 없으나, 수염이나 머리색 안경등과 같이 전형적인 특징이 존재합니다. 즉 이런 전형적인 특징을 통해 대상을 분류합니다. 프로토타입 이론은 가족 유사성 이론을 토대로 발전합니다.

Rosch 의 프로토타입 이론(Prototype Theory)

1975년에 Rosch 는 한 가지 실험을 합니다.

- 실험 참가자들에게 여러 범주 구성원(사과, 코코넛, 오렌지)의 속성을 적어보라고 함
- 각 범주 구성원에 대해 범주의 다른 구성원과 공유하는 속성의 개수를 도출함
- 사과, 오렌지 : 2점(둥글다. 즙이 있다.)
- 코코넛 : 1점(둥글다)

점수가 높을수록 '가족 유사성'이 높다고 볼 수 있습니다. 전통적인 분류에선 모두 과일로 볼 수 있지만, 프로토타입 이론에서는 사과와 오렌지가 가장 전형적인 무언가라고 볼 수 있습니다. 반면에 코코넛은 저 중에서 가장 비전형적인 것으로 볼 수 있습니다.

이 실험을 통해 로쉬는 "인간은 '등급의 구조(graded structure)'를 가진다"라고 주장합니다. 인간은 사물을 분류할 때 자연스럽게 가장 유사성 높은 것 순서대로 등급을 매긴다는 의미로 볼 수 있습니다. 이렇게 분류했을 때 가장 높은 등급을 가진 녀석이 나올 텐데요, 이것이 바로 원형(Prototype)입니다.

즉, 객체는 '정의'로부터 분류되는 것이 아니라 가장 좋은 보기(prototype, exemplar)로부터 범주화된다고 합니다.

이런 분류체계는 매우 효율적입니다. 기존에 없던 새로운 대상을 발견했을 때, 새로운 대상의 몇 가지 특징을 파악하고, 기존에 있던 원형과 비교하여 범주화를 시도하면 됩니다.

프로토타입 이론은 또한, 의미사용이론을 지지합니다. 즉 동일한 대상에 대해 원형과 범주화를 시도할 때, 누가 어떤 상황(context)에서 대상을 접하냐에 따라 그 결과가 달라질 수 있다는 것입니다.

예를 들면 아이가 생각하는 새의 범주에서 '참새'는 명확하게 새에 속하지만 '펭귄' 은 해당 범주에 속하지 못할 수도 있습니다. 아이가 생각할 땐 펭귄이 매우 비전형적이기 때문이죠. 하지만 조류학자가 생각할 때 '참새'와 '펭귄'은 명확하게 유사한 새의 범주에 속할 수 있습니다. 같은 단어여도 어떤 상황에서 접했나에 따라 범주는 크게 달라집니다.

결국 Rosch의 프로토타입 이론이 자바스크립트와 같은 프로토타입 기반의 객체지향 언어들의 토대가 됩니다.

자바스크립트는 프로토타입을 어떻게 구현했는가?

결국 프로토타입에서는 기존의 객체지향과 달리 분류(classification)을 지향하지 않습니다. 다만 원형이 존재하고, 생성된 객체들의 특징에 따라 유사성을 정의하고 범주화할 뿐입니다. 해당 과정에서 결국은 객체(대상)은 상황(Context)에 따라 결정되고 평가됩니다.

자바스크립트의 Context, Scope, Closure, this, Hoisting 등의 개념은 이러한 프로토타입 이론의 'context' 를 표현하기 위한 수단으로서 활용되는 것입니다.

Prototype vs Class

클래스 기반의 객체지향 설계는 일반적인 부모 클래스(Task)와 유사한 태스크의 공통 작동을 정의하는 것부터 시작합니다. 그리고 나서 Task를 상속받은 자식 클래스를 정의한 후, 이들에 특화된 작동은 두 클래스에 각각 추가합니다.

상속의 진가를 발휘하기 위해 가능 하면 메서드를 오버라이드(다형성) 할 것을 권장하고, 작동 추가뿐 아니라 때에 따라서 오버라이드 이전 원본 메서드를 super 키워드로 호출할 수 있게 지원합니다.

공통 요소는 추상화하여 부모 클래스의 일반 메서드로 구현하고 자식 클래스는 이를 더 세분화(오버라이드)합니다. 클래스는 인스턴스화 하여 모든 요소를 복사하여 사용합니다.

즉 클래스라는 원형과 그 실체인 인스턴스에 충실합니다.

프로토타입 기반의 자바스크립트는 그 설계가 다릅니다. 먼저 클래스와 같이 Task 객체를 정의하는 것 자체는 동일합니다. 그러나 이 객체에는 다양한 태스크에서 사용할 유틸리티 메서드가 포함된 구체적인 작동이 기술됩니다. 최대한 일반화와 추상화를 통해 원형을 클래스로 설계하려는 것과 상반됩니다.

여러 가지 테스크가 존재할 때, 해당 테스크들의 공통된 속성과 행위를 찾아 테스크 클래스를 정의하는 것이 아니라, 각 태스크별 객체를 정의하여 고유한 데이터와 작동을 정의하고 Task 유틸리티 객체에 연결해 필요할 때 특정 태스크 객체가 Task에 작동을 위임하도록 작성한다.

여기서 중요한 것은 연결과 위임입니다. 자바스크립트의 Prototype 체계는 객체를 다른 객체에 연결합니다. 클래스 같은 추상화 체계는 존재하지 않습니다.

이처럼 객체와 객체간 연결이라는 개념에서 등장한 것이 '작동 위임' 개념입니다.

작동 위임 (Behavior Delegation)

작동 위임이란 찾으려는 프로퍼티/메서드 레퍼런스가 객체에 없으면 다른 객체로 수색 작업을 위임하는 것을 의미합니다. 부모/자식 클래스, 상속, 다형성은 전혀 상관없고 객체들이 수평적으로 배열된 상태에서 링크가 체결되는 모습을 떠올리면 됩니다.

동일한 동작의 코드를 각각 class 기반과 위임 기반의 방식으로 작성해 봅시다.

class 지향

```javascript
function Foo(who) {
  this.me = who;
}

Foo.prototype.identify = function() {
  return `I am ${this.me}`;
}

function Bar(who) {
  Foo.call(this, who);
}
Bar.prototype = Object.create(Foo.prototype);
Bar.prototype.speak = function() {
  alert(`Hello, ${this.identify()}.`);
}

var b1 = new Bar('b1');
var b2 = new Bar('b2');

b1.speak();
b2.speak();
```

작동 위임 지향

```javascript
const Foo = {
  init: function init(who) {
    this.me = who;
  },
  identify: function identify() {
    return `I am ${this.me}`;
  }
```

```
}

const Bar = Object.create(Foo);
Bar.speak = function speak() {
  alert(`Hello, ${this.identify()}.`);
}

const b1 = Object.create(Bar);
b1.init('b1');
const b2 = Object.create(Bar);
b2.init('b2');

b1.speak();
b2.speak();
```

클래스 지향은 객체의 생성과 동시에 초기화를 해줘야 하는 반면에, 위임 지향의 경우 객체의 생성과 초기화가 자연스럽게 분리가 됩니다. 결국 관심사의 분리 관점을 더 잘 반영할 수 있습니다. 또한 상속이나 추상화를 신경쓰기 위한 복잡한 구조 설계보다 더 간단하고 직관적인 코드의 작성이 가능해집니다.

자바스크립트에서 이런 작동 위임을 프로토타입 체이닝을 통해 구현합니다.

사용자가 특정 객체의 프로퍼티에 접근하려 시도하거나, 메소드를 호출하려는 상황에는 다음과 같은 전제 중 하나 이상이 기저에 깔려있습니다.

1. 사용자는 해당 객체의 원형을 알고 있다.
2. 사용자는 해당 객체의 특성을 알고 있다.

해당 객체의 원형 혹은 특성을 안다는 말은 사용자가 해당 객체를 통해 수행하고 싶은 작업이 무엇인지 알 수 있고, 해당 객체가 어떤식으로 동작해야 하는지 어느정도 예측하고 있다는 말입니다.

즉 프로토타입을 이해하는 사용자는 "프로토타입 기반인 자바스크립트에서 분명 이 객체는 x객체로 부터 위임받았으니 내가 명령한 y 함수도 정상적으로 수행할 수 있을거야!" 처럼 기대하게 됩니다.

다음과 같은 상황에 대해, 자바스크립트가 정상적으로 유저의 기대 사항이 수행하도록 하기 위해 프로토타입 체이닝이 존재한다고 볼 수 있습니다.

프로토타입 체이닝은 특정 객체에 사용자가 원하는 메서드나 프로퍼티가 존재하지 않는다면 해당 객체에 대한 위임 주체가 되는 상위 프로토타입으로 되돌아가 원하는 요소를 탐색합니다. 이는 해당 객체의 원형에 도달할 때 까지 반복되며, 원형에 도달해서 정상적인 반환값이 존재하지 않는다면 그제서야 에러를 발생시킵니다.

Type Introspection

타입 인트로스펙션은 인스턴스를 조사해 객체 유형을 거꾸로 유추하는 방식입니다. 클래스 인스턴스에서 타입 인트로스펙션은 주로 인스턴스가 생성된 소스 객체의 구조와 기능을 추론하는 용도로 사용됩니다.

그러나 자바스크립트에서는 클래스 인스턴스라는 개념이 존재하지 않습니다. 즉 자바스크립트의 타입 인트로스펙션은 해당 객체가 어떤 객체에 의해 위임되었는가, 해당 객체와 링크된 객체가 무엇인지를 판단하는 것입니다.

다음은 자바스크립트에서 타입 인트로스펙션을 구현하는 몇가지 코드 입니다.

1. instanceof

```javascript
const arr = [1, 2, 3];

console.log(arr instanceof Array); // true

function Person(name) {
    this.name = name;
}
const person = new Person('John');

console.log(person instanceof Person); // true
```

자바와 같은 클래스 기반의 언어에서 작동하는 instanceof 키워드와는 달리, 자바스크립트의 instanceof 키워드로 얻은 결과는 단지 대상 객체와 위임, 연결된 상태임을 나타냅니다.

객체 인스턴스와 클래스 사이의 관계를 조사하는게 아니라 객체 및 해당객체와 연결된(혹은 위임의 주체)인 prototype 사이의 관계를 알려주는 역할에 불과합니다.

1. typeof - 원시값의 type introspection

```javascript
console.log(typeof 123); // "number"
console.log(typeof 'Hello, world!'); // "string"
console.log(typeof true); // "boolean"
console.log(typeof null); // "object" (This is a known issue in JavaScript
```

원시값은 그 자체로는 객체가 아닙니다. 그러나 자바스크립트의 철학 상,프로토타입을 기반으로하는 자바스크립트에서 원형이 되는 객체가 존재하지 않는 것은 없어야 한다. 그래서 자바스크립트는 원시값도 원형이 되는 객체를 지원한다. 각 원시값을 래핑하는 Wrapper 객체들이다. 예로 어느 원시값의 데이터 타입이 string일 때, 접두사가 대문자인 String이라는 String 객체가 존재한다.

결국 typeof 도 자바스크립트 내에서 타입 인트로스펙션을 구현하려면 객체의 링크를 통해 원형을 찾아가야 한다. 이 때 사용되는 것이 각 원시 데이터 형에 대한 Wrapper 객체들이다. 그러나 이 객체들은 항상 메모리에 상주하는 것이 아니라, 메서드 호출로 인해 해당 래퍼 객체가 필요해지는 상황에만 일시적으로 생성되었다가 폐기된다.

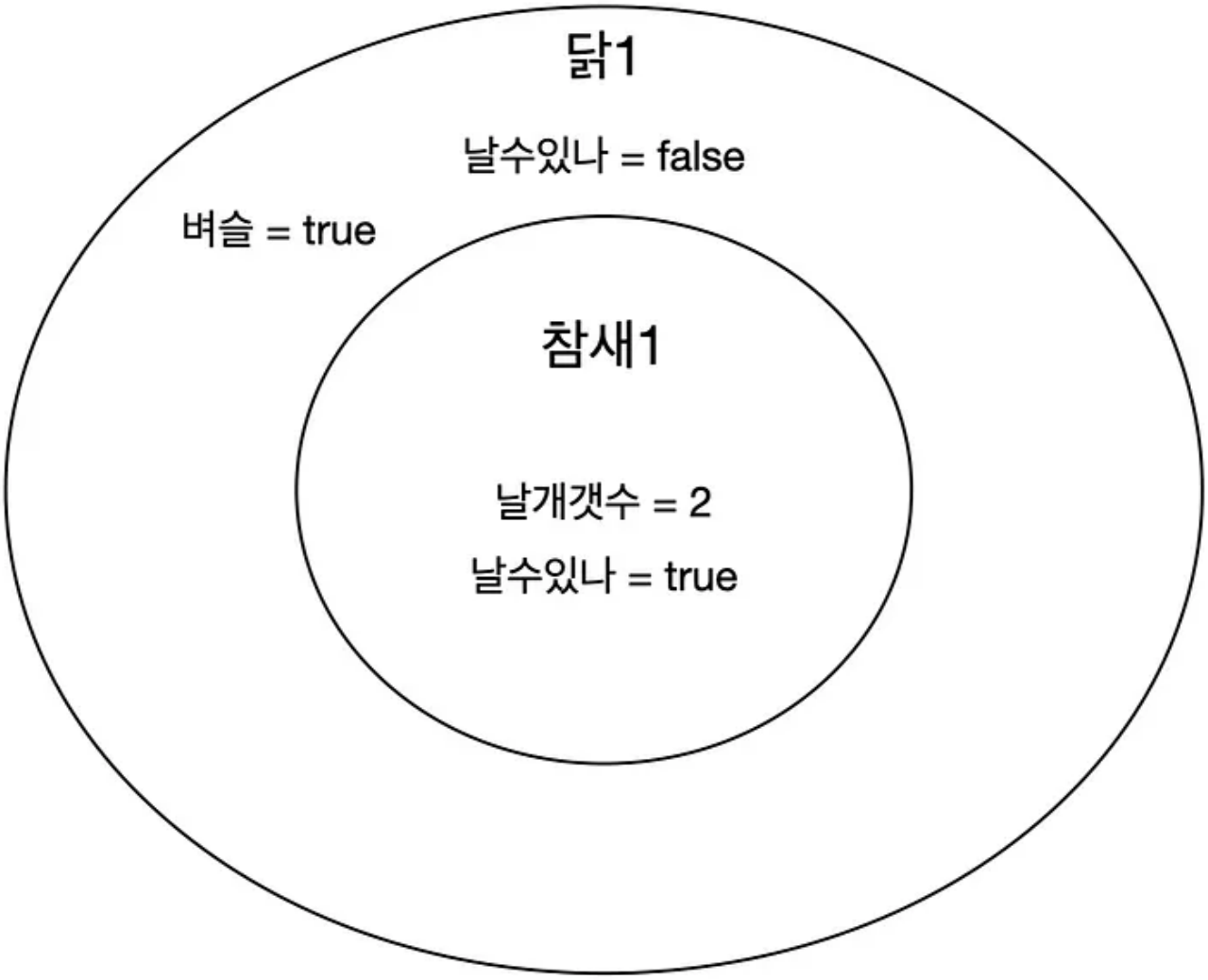
객체는 객체일 뿐이다

- 프로토타입의 참조: 프로토타입 이론은 이미 존재하는 사물을 통해 범주화한다
  특정 객체의 프로토타입은 이미 존재하는 객체이다.
  기존 생성자 함수로 생성된 객체 인스턴스와 유사성이 있는 다른 객체에 대한 프로토타입을 정의할 때, 생성자 함수 자체가 아니라, 이미 존재하는 사물인 객체 그 자체를 기반으로 설정한다.
- 객체 속성의 변경 : 각 객체들은 본질과 실체가 구분되는것이 아니라 그 자체로 하나의 대상이자 의미를 가진다.
  즉 특정 객체를 원형으로 가지는 다른 객체에서, 원형 객체와 공통점이 되는 특정 프로퍼티에 변경이 있다면, 그것은 해당 객체가 원형에서 조금 더 멀어진 범주화에 속하는 것이지, 하나의 본질 자체가 변경되는 것이 아니다. 따라서 프로퍼티 변경이 일어난 객체에서만 수정이 이루어지고, 원형 객체의 속성은 변하지 않는다.

이러한 특징들은 자바스크립트가 클래스를 지향하는 다른 언어와 달리 클래스/상속 패턴이 아니라 작동 위임의 패턴을 중심으로 작동한다고 사고할 수 있도록 합니다.

```javascript
function 참새(){
    this.날개갯수 = 2;
    this.날수있나 = true;
}

const 참새1 = new 참새();
console.log("참새의 날개 갯수 : ", 참새1.날개갯수); // 2

function 닭(){
    this.벼슬 = true;
}

닭.prototype = 참새1; // reference(오른쪽이 인스턴스인 점 주목)
const 닭1 = new 닭();

console.log("닭1 날개 : ", 닭1.날개갯수, ", 날수있나? ", 닭1.날수있나); // 2, true

닭1.날수있나 = false;

console.log("다시 물어본다. 닭1은 날 수 있나? :", 닭1.날수있나); // false
```

위 코드는 아래 도식과 같다.



호이스팅과 렉시컬 스코프는 왜 존재하는가?

의미사용이론에 따르면 표현은 누가 어떤 맥락에서 사용하느냐에 따라 달라집니다. 자바스크립트의 렉시컬 스코프와 호이스팅은 이 부분을 뒷받침하는 가장 강력한 도구중 하나입니다.

> 변수의 이미는 그 어휘적인 실행 문맥에서의 의미가 규정된다.

그래서 실행 문맥의 모든 선언을 참고하여 변수의 의미를 정해야 합니다.즉 실행 문맥에서 선언이 최상단으로 끌어올려지는 호이스팅은 각 변수의 의미를 규정하기 위해 해당 컨텍스트가 어떤지 파악하기 위한 개념입니다.

자바스크립트는 '단어의 의미가 사용되는 근처 환경' 에서의 '근처'를 어휘적인 범위(Lexical Scope)로 정의했습니다. 자바스크립트 엔진은 코드가 로드될 때 실행 컨텍스트를 생성하고 그 안에 선언된 변수, 함수를 실행 컨텍스트 최상단으로 호이스팅 합니다. 이러한 범위를 렉시컬 스코프라 합니다.

```javascript
// 전역 실행문맥 생성. 전체 정의(name, init) 호이스팅
var name = 'Kai';
init(); // init 실행문맥 생성. 내부 정의(name, displayName) 호이스팅

function init() {
    var name = "Steve";
    function displayName() {
        console.log(name); // 현재 실행문맥 내에 정의된게 없으니 outer 로 chain
        // var name = 'troll?'; // 주석 해제되면 호이스팅
    }

    displayName(); // displayName 실행문맥 생성. 내부 정의 호이스팅.
}
```

결국 자바스크립트의 실행 문맥', '렉시컬 스코프', '호이스팅'과 같은 시스템은 어휘가 문맥에서만 의미를 가진다는 핵심을 구현하기 위한 자연스러운 특징일 뿐입니다.

this 는 특별한 무언가가 아니다.

비트겐슈타인은 어휘는 누가 어떤 상황에서 쓰냐에 따라 그 의미가 달라지는 것을 강조합니다.
그는 이를 '발화' 라고 규정합니다. 위에서 예시 들었던 것 처럼 '벽돌!'이라고 크게 외칠 때, 그것이 어디서 '발화'되느냐에 따라서 단어의 의미가 달라집니다. 좀더 쉽게 얘기하면 받아들이는 '대상'에 따라서 같은 단어도 의미가 달라진다는 얘기입니다.

이것이 바로 프로토타입과 클래스의 대표적인 차이라고 볼 수 있습니다. 전혀 다르게 단어를 보는 방식이고 중요한 세계관의 차이입니다. 미리 분류하고 정의한 클래스를 가장 중요하게 여기는 전통적인 방식과는 달리, 프로토타입에서는 받아들이는 주체와 문맥이 가장 중요한 것이죠. 프로그래밍으로 보자면 실행(invoke)하는 '객체' 가 중요하다는 의미입니다.

이것이 바로 프로토타입 기반 언어인 자바스크립트에서 this 가 클래스 기반 언어들과 다르게 동작하는 이유입니다.

프로토타입 기반 언어에서는 this 가 정의된 함수가 어떻게 발화(invoke) 되었는지에 따라 가리키는 값이 달라집니다. 정확히는 받아들이는 대상의 컨텍스트를 가리킵니다.

이를 이해하려면 먼저 메소드와 메시지를 명확하게 알아야 합니다.

- 메소드 : 객체의 함수
- 메시지 : 메소드를 실행하라는 메시지 전달

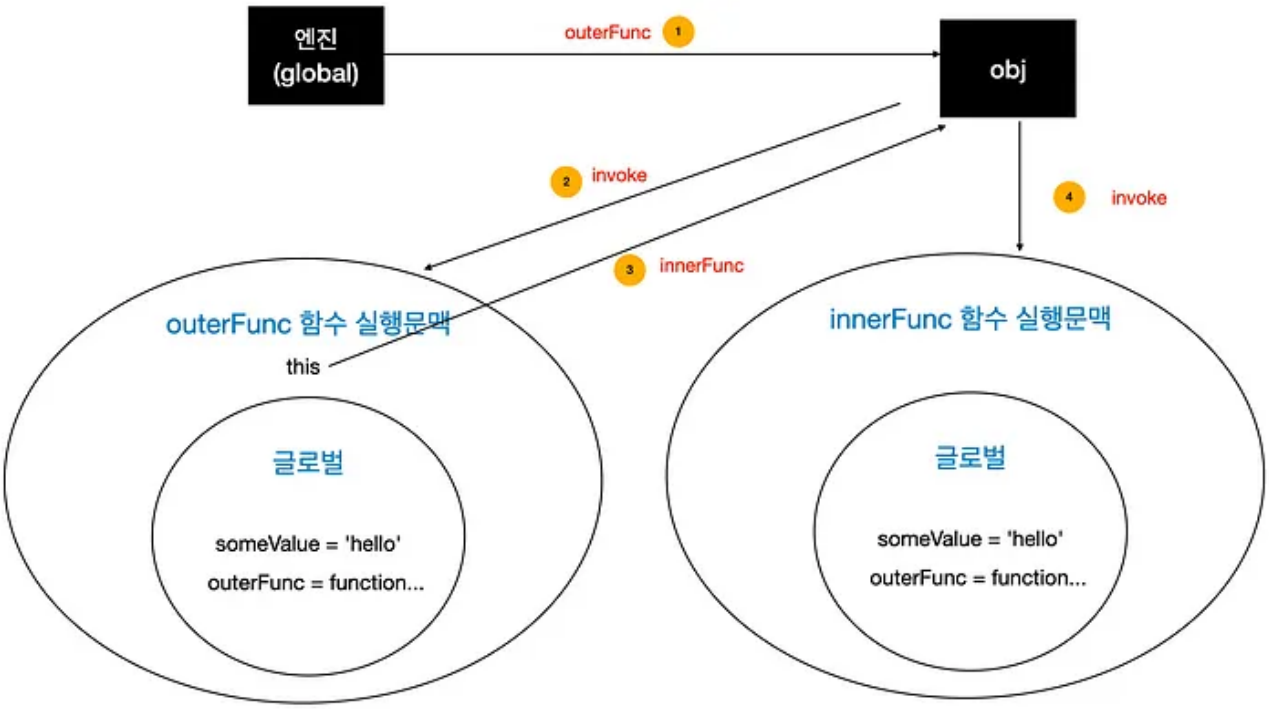
자바에서는 클래스의 메소드를 호출하는 행위를 메시지라 합니다. 자바스크립트 개발자에게는 위 용어가 익숙하지 않을 수도 있을 것 같은데, 메시지로 이해하는 것이 앞으로의 내용을 이해하는데 도움이 될 것 같습니다. 자바스크립트를 예로 들면 foo 라는 객체가 있고 그 내부에 bar() 라는 함수가 있을 때 다음처럼 발화(invoke) 할 객체를 지정할 수 있습니다.

- foo.bar()
- bar.call(foo)
- var boundBar = bar.bind(foo)

위처럼 foo 객체를 통해 발화한 함수는 내부 this 가 무조건 foo 를 가리킵니다. 만약 아무것도 지정되어있지 않으면 글로벌(브라우져라면 window)을 가리킵니다.

```javascript
var someValue = 'hello';

function outerFunc() {
    console.log(this.someValue); // 첫번째 : ?, 두번째 : ?
    this.innerFunc();
}

const obj = {
    someValue : 'world',
    outerFunc,
    innerFunc : function() {
        console.log("innerFunc's this : ", this); // 첫번째 : ?, 두번째 : ?
    }
}

obj.outerFunc(); // 첫번째
outerFunc(); // 두번째
```

첫 번째 호출(obj가 발화 주체일 때)



두 번째 호출(글로벌 문맥이 발화 주체일 때)