Chap 16 프로퍼티 어트리뷰트

16-1 내부 슬롯과 내부 메서드

내부 슬롯과 내부메서드

- 자바스크립트 엔진의 구현 알고리즘을 설명하기 위해 ECMAScript 사양에서 사용하는 의사 프로퍼티와 의사 메서드다.
- 외부로 공개된 프로퍼티가 아니기때문에 직접적으로 접근하거나 호출할 수 있는 방법을 제공하지 않는다.
- 단, 일부 내부 슬롯과 내부 메서드에 한하여 간접적으로 접근할 수 있는 수단 제공

  ex)[[prototype]]은 `__proto__` 를통해 간접적으로 접근

```
const o={};
//내부 슬롯은 자바스크립트 엔진의 내뮵 로직이므로 직접 접근할 수 없다
o.[[Prototype]]
//단, 일부 내부 슬롯과 내부 메서드에 한하여 간접적으로 접근할 수 있
o.__proto__
```

16-2 프로퍼티 어트리뷰트와 프로퍼티 디스크립터 객체

자바스크립트 엔진은 프로퍼티를 생성할 때 프로퍼티의 상태를 나타내는 프로퍼티 어트리뷰트를 기본값으로 자동 정의한다.

▼ 프로퍼티 어트리뷰트

자바스크립트가 관리하는 내부 상태 값인 내부스롯이다.

- 값 `[[Value]]`

- 갱신 가능 여부 `[[Writable]]`
- 열거 가능 여부 `[[enumerable]]`
- 재정의 가능 여부 `[[configurable]]`

접근 방법 (간접적)

- Object.getOwnPropertyDescriptor

```
const person = {
  name: 'Lee'
};

// 프로퍼티 동적 생성
person.age = 20;

// 모든 프로퍼티의 프로퍼티 어트리뷰트 정보를 제공하는 프로퍼티 디스크립
console.log(Object.getOwnPropertyDescriptors(person));
/*
{
  name: {value: "Lee", writable: true, enumerable: true, c
  age: {value: 20, writable: true, enumerable: true, confi
}
*/
```

첫번째 매개변수에는 객체의 참조를 전달하고 두번째 매개변수에는 프로퍼티 키를 문자열로 전달하여 호출한다. 호출 후에는 프로퍼티 디스크립터 객체를 반환한다.

만약 존재하지 않으면 `undefined`가 반환된다.

`Object.getOwnPropertyDescriptor`는 하나의 프로퍼티에 대한 결과만 반환하지만 ES8에 추가된

`Object.getOwnPropertyDescriptors`메서드는 객체의 모든 프로퍼티 어트리뷰트정보를 제공하는 프로퍼티 디스크립터 객체들을 반환한다.

16-3 데이터 프로퍼티와 접근자 프로퍼티

▼ 프로퍼티(속성)

- property는 해당 object의 특징입니다. property는 보통 데이터 구조와 연관된 속성을 나타냅니다.

- property는 **이름**(string 이나 symbol)과 **값**(원시함수(primitive), 메서드(method) 또는 객체 참조(object reference))을 가지고 있습니다.
- 보통 "프로퍼티가 object를 가지고 있다"라는 것은 "property가 object reference"를 가지고 있다는 것을 줄여서 말한 것이라는 것을 기억하세요.
- property의 값을 변경할 때 기존에 참조된 object는 그대로 남아있기 때문에 이걸 구분하는 것이 중요합니다.

종류

- 데이터 프로퍼티 data property

  키와 값으로 구성된 일반적인 프로퍼티다.

- 접근자 프로퍼티 accessor property

  자체적으로는 값을 갖지 않고 달느 데이터 프로퍼티의 값을 읽거나 저장할 때 호추되는 접근자 함수로 구성된 프로퍼티

▼ 데이터 프로퍼티

데이터 프로퍼티가 갖는 프로퍼티 어트리뷰트

- 값 `[[Value]]`
- 갱신 가능 여부 `[[Writable]]`
- 열거 가능 여부 `[[enumerable]]`
- 재정의 가능 여부 `[[configurable]]`

접근 방법

Object.getOwnPropertyDescriptor 메서드

▼ 접근자 프로퍼티

자체적으로 값을 갖지 않고 다른 데이터 프로퍼티의 값을 읽거나 저장할 때 사용하는 접근자 함수로 구성된 프로퍼티

프로퍼티 어트리 뷰트

- `[[get]]` : 데이터 프로퍼티 값읽을때 호출

- `[[set]]` : 데이터 프로퍼티 값 저장할때 호출
- `[[enumerable]]` : 데이터 프로퍼티 enumerable과 일치
- `[[configurable]]` : 데이터 프로퍼티 configurable과 일치

```
const person = {
  // 데이터 프로퍼티
  firstName: 'Ungmo',
  lastName: 'Lee',

  // fullName은 접근자 함수로 구성된 접근자 프로퍼티다.
  // getter 함수
  get fullName() {
    return `${this.firstName} ${this.lastName}`;
  },
  // setter 함수
  set fullName(name) {
    // 배열 디스트럭처링 할당: "31.1 배열 디스트럭처링 할당" 참고
    [this.firstName, this.lastName] = name.split(' ');
  }
};

// 데이터 프로퍼티를 통한 프로퍼티 값의 참조.
console.log(person.firstName + ' ' + person.lastName); //

// 접근자 프로퍼티를 통한 프로퍼티 값의 저장
// 접근자 프로퍼티 fullName에 값을 저장하면 setter 함수가 호출된다.
person.fullName = 'Heegun Lee';
console.log(person); // {firstName: "Heegun", lastName: "L

// 접근자 프로퍼티를 통한 프로퍼티 값의 참조
// 접근자 프로퍼티 fullName에 접근하면 getter 함수가 호출된다.
console.log(person.fullName); // Heegun Lee

// firstName은 데이터 프로퍼티다.
// 데이터 프로퍼티는 [[Value]], [[Writable]], [[Enumerable]],
let descriptor = Object.getOwnPropertyDescriptor(person, '
console.log(descriptor);
```

```
// {value: "Heegun", writable: true, enumerable: true, con

// fullName은 접근자 프로퍼티다.
// 접근자 프로퍼티는 [[Get]], [[Set]], [[Enumerable]], [[Confi
descriptor = Object.getOwnPropertyDescriptor(person, 'full
console.log(descriptor);
// {get: f, set: f, enumerable: true, configurable: true}
```

위 예제에서 접근자 프로퍼티 `fullName`으로 프로퍼티 값에 접근하면 내부적으로 `[[Get]]` 내부 메서드가 호출되어 다음과 같이 동작한다.

1. 프로퍼티 키가 유효한지 확인한다.(프로퍼티 키는 문자열 또는 심벌이어야 한다.)
2. 프로토타입 체인에서 프로퍼티를 검색한다.(`person` 객체에 `fullName` 프로퍼티가 존재한다.)
3. 검색된 `fullName` 프로퍼티가 데이터 프로퍼티인지 접근자 프로퍼티인지 확인한다.(접근자 프로퍼티임)
4. 접근자 프로퍼티 `fullName`의 프로퍼티 어트리뷰트 `[[Get]]`의 값, 즉 `getter` 함수를 호출하여 그 결과를 반환한다.

접근자 프로퍼티와 데이터 프로퍼티 구별 방법

- `Object.getOwnPropertyDescriptor`의 결과로 구분 가능하다.

```
// 일반 객체의 __proto__는 접근자 프로퍼티다.
Object.getOwnPropertyDescriptor(Object.prototype, '__proto
// {get: f, set: f, enumerable: false, configurable: true}

// 함수 객체의 prototype은 데이터 프로퍼티다.
Object.getOwnPropertyDescriptor(function() {}, 'prototype'
// {value: {...}, writable: true, enumerable: false, confi
```

16-4 프로퍼티 정의

새로운 프로퍼티를 추가하면서 프로퍼티 어트리뷰트를 명시적으로 정의하거나, 기존 프로퍼티의 프로퍼티 어트리뷰트를 재정의하는 것

정의 방법

- `Object.definedProperty` 메서드

```
const person = {};

// 데이터 프로퍼티 정의
Object.defineProperty(person, 'firstName', {
  value: 'Ungmo',
  writable: true,
  enumerable: true,
  configurable: true
});

Object.defineProperty(person, 'lastName', {
  value: 'Lee'
});

let descriptor = Object.getOwnPropertyDescriptor(person, 'fir
console.log('firstName', descriptor);
// firstName {value: "Ungmo", writable: true, enumerable: tru

// 디스크립터 객체의 프로퍼티를 누락시키면 undefined, false가 기본값이디
descriptor = Object.getOwnPropertyDescriptor(person, 'lastNam
console.log('lastName', descriptor);
// lastName {value: "Lee", writable: false, enumerable: false

// [[Enumerable]]의 값이 false인 경우
// 해당 프로퍼티는 for...in 문이나 Object.keys 등으로 열거할 수 없다.
// lastName 프로퍼티는 [[Enumerable]]의 값이 false이므로 열거되지 않
console.log(Object.keys(person)); // ["firstName"]

// [[Writable]]의 값이 false인 경우 해당 프로퍼티의 [[Value]]의 값을
// lastName 프로퍼티는 [[Writable]]의 값이 false이므로 값을 변경할 수
// 이때 값을 변경하면 에러는 발생하지 않고 무시된다.
person.lastName = 'Kim';

// [[Configurable]]의 값이 false인 경우 해당 프로퍼티를 삭제할 수 없다
// lastName 프로퍼티는 [[Configurable]]의 값이 false이므로 삭제할 수
// 이때 프로퍼티를 삭제하면 에러는 발생하지 않고 무시된다.
```

```
delete person.lastName;

// [[Configurable]]의 값이 false인 경우 해당 프로퍼티를 재정의할 수 없
// Object.defineProperty(person, 'lastName', { enumerable: tr
// Uncaught TypeError: Cannot redefine property: lastName

descriptor = Object.getOwnPropertyDescriptor(person, 'lastNam
console.log('lastName', descriptor);
// lastName {value: "Lee", writable: false, enumerable: false

// 접근자 프로퍼티 정의
Object.defineProperty(person, 'fullName', {
  // getter 함수
  get() {
    return `${this.firstName} ${this.lastName}`;
  },
  // setter 함수
  set(name) {
    [this.firstName, this.lastName] = name.split(' ');
  },
  enumerable: true,
  configurable: true
});

descriptor = Object.getOwnPropertyDescriptor(person, 'fullNam
console.log('fullName', descriptor);
// fullName {get: ƒ, set: ƒ, enumerable: true, configurable:

person.fullName = 'Heegun Lee';
console.log(person); // {firstName: "Heegun", lastName: "Lee"
```

`Object.defineProperty`메서드는 한번에 하나의 프로퍼티만 정의할 수 있다. 그리고 `Object.defineProperties`메서드를 사용하면 여러개의 프로퍼티를 한 번에 정의할 수 있다.

```
const person = {};

Object.defineProperties(person, {
  // 데이터 프로퍼티 정의
```

```
  firstName: {
    value: 'Ungmo',
    writable: true,
    enumerable: true,
    configurable: true
  },
  lastName: {
    value: 'Lee',
    writable: true,
    enumerable: true,
    configurable: true
  },
  // 접근자 프로퍼티 정의
  fullName: {
    // getter 함수
    get() {
      return `${this.firstName} ${this.lastName}`;
    },
    // setter 함수
    set(name) {
      [this.firstName, this.lastName] = name.split(' ');
    },
    enumerable: true,
    configurable: true
  }
});

person.fullName = 'Heegun Lee';
console.log(person); // {firstName: "Heegun", lastName: "Lee"
```

16-5 객체 변경 방지

▼ 객체 확장 금지

- 프로퍼티 추가 금지를 의미. 즉 프로퍼티 동적 추가와 `Object.defineProperty` 메서드가 금지된다.
- `Object.preventExtensions` 메서드사용하여 객체 확장을 금지한다.

▼ 객체 밀봉

- 프로퍼티 추가 및 삭제와 프로퍼티 어트리뷰트 재정의를 금지한다. 즉 밀봉된 객체는 읽기와 쓰기만 가능하다.
- `Object.seal` 메서드 사용하여 객체를 밀봉하고 `Object.sealed` 메서드로 밀봉된 객체인지 확인할 수 있다.

▼ 객체동결

- 프로퍼티 추가 및 삭제와 프로퍼티 어트리뷰트 재정의 금지, 프로퍼티 값 갱신 금지를 의미. 즉 동결된 객체는 읽기만 가능
- `Object.freeze` 메서드로 객체를 동결시킬 수 있다. `Object.isFrozen` 메서드로 동결된 객체인지 확인할 수 있다.

▼ 불변 객체

지금까지 살펴본 변경 방지 메서드들 (`Object.preventExtensions`, `Object.seal`, `Object.freeze`)은 얕은 변경 방지(`shallow only`)이기 때문에 직속 프로퍼티만 변경이 방지되고 중첩 객체까지는 영향을 주지 못한다.

따라서 객체 동결로 예를들어보자면

```
const person = {
  name: 'Lee',
  address: { city: 'Seoul' }
};

// 얕은 객체 동결
Object.freeze(person);

// 직속 프로퍼티만 동결한다.
console.log(Object.isFrozen(person)); // true
// 중첩 객체까지 동결하지 못한다.
console.log(Object.isFrozen(person.address)); // false

person.address.city = 'Busan';
console.log(person); // {name: "Lee", address: {city: "Bus
```

`Object.freeze` 메서드는 중첩 객체까지 동결할 수없다. 따라서 불변 객체를 구현하려면 객체를 값으로 갖는 모든 프로퍼티에 대해 재귀적으로 Object.freeze 메서드를 호출행한다.

```
function deepFreeze(target) {
  // 객체가 아니거나 동결된 객체는 무시하고 객체이고 동결되지 않은 객체
  if (target && typeof target === 'object' && !Object.isFr
    Object.freeze(target);
    /*
      모든 프로퍼티를 순회하며 재귀적으로 동결한다.
      Object.keys 메서드는 객체 자신의 열거 가능한 프로퍼티 키를 배
      ("19.15.2. Object.keys/values/entries 메서드" 참고)
      forEach 메서드는 배열을 순회하며 배열의 각 요소에 대하여 콜백
      ("27.9.2. Array.prototype.forEach" 참고)
    */
    Object.keys(target).forEach(key => deepFreeze(target[k
  }
  return target;
}

const person = {
  name: 'Lee',
  address: { city: 'Seoul' }
};

// 깊은 객체 동결
deepFreeze(person);

console.log(Object.isFrozen(person)); // true
// 중첩 객체까지 동결한다.
console.log(Object.isFrozen(person.address)); // true

person.address.city = 'Busan';
console.log(person); // {name: "Lee", address: {city: "Seo
```