19장. 프로토타입

▼ 자바스크립트은 어떤 프로그래밍 언어인가?

클래스 기반 객체지향 프로그래밍 언어보다 더 효율적인 더 강력한 객체 지향 프로그래밍 능력을 가지고 있는 **프로토타입 기반의 객체지향 프로그래밍 언어**

▼ 클래스란?

함수이며, 기존 프로토타입 기반 패턴의 문법적 설탕

⇒ 생성자 함수와 프로토타입 기반의 인스턴스를 생성한다는 점에서 비슷하지만,

클래스는 생성자 함수보다 **더 엄격**하며 생성자 함수에서 **제공하지 않는 기능도 제공**

19.1 객체 지향 프로그래밍

▼ 객체 지향 프로그래밍이란?

여러 개의 독립적 단위, 즉 객체의 집합으로 프로그램을 표현하려는 프로그래밍 패러다임

▼ 추상화란?

다양한 속성 중에서 프로그램에 필요한 속성만 간추려 내어 표현하는 것

▼ 객체란?

속성을 통해 여러 개의 값을 하나의 단위로 구성한 복합적인 자료구조

= 상태 데이터와 동작을 하나의 논리적인 단위로 묶은 복합적인 자료구조

프로퍼티: 객체의 상태 데이터

메서드: 객체의 동작

▼ 객체의 특징은?

1. 자신의 고유한 기능을 수행하면서 다른 객체와 관계성을 가질 수 있다.

2. 다른 객체의 상태 테이터나 동작을 상속받아 사용하기도 한다.

19.2 상속과 프로토타입

▼ 상속이란?

객체 지향 프로그래밍의 핵심 개념으로, 어떤 객체의 프로퍼티 또는 메서드를 다른 객체가 상속받아 그대로 사용할 수 있는 것

▼ 상속의 특징은?

프로토타입을 기반으로 상속을 구현하여 불필요한 중복을 제거 = **코드 재사용** ⇒ 개발 비용 절감

▼ 상속을 하지 않는다면?

동일한 생성자 함수에 의해 생성된 모든 인스턴스가 동일한 메서드를 중복 소유하게 됨

→ 메모리를 불필요하게 **낭비**

→ 퍼포먼스에도 **악영향**

따라서 자바스크립트는 프로토타입을 기반으로 상속을 구현

19.3 프로토타입 객체

▼ 프로토타입 객체란?

객체 지향 프로그래밍의 객체 간 상속을 구현하기 위해 사용

▼ 프로토타입 특징은?

어떤 객체의 상위 객체의 역할을 하는 객체로서 다른 객체에 공유 프로퍼티를 제공

프로토타입을 상속받은 하위객체는 상위 객체의 프로프티를 자신의 프로퍼티처럼 자유롭게 사용 가능

→ 모든 객체는 하나의 프로토타입을 갖는다.

→ 객체와 프로토타입과 생성자 함수는 서로 연결되어있다.

▼ __proto__접근자 프로퍼티란?

모든 객체는 접근자 프로퍼티를 통해 자신의 프로토타입, 즉 내부 슬롯에 간접적으로 접근할 수 있다.

▼ object.prototype란?

프로토타입 체인의 최상위 객체, 이 객체의 프로퍼티와 메서드는 모든 객체에 상속된다.

▼ __proto__ 접근자 프로퍼티를 통해 프로토타입에 접근하는 이유

상호 참조에 의해 프로토타입 체인이 생성되는 것을 방지 하기 위해서

→ 서로가 자신의 프로토타입이 되는 비정상적인 프로토타입 체인이 만들어지기 때문

프로토타입 체인은 단방향 링크드 리스트로 구현 되어야 함

▼ 양방향이 된다면?

순환 참조하는 프로토타입 체인이 만들어지면 프로토타입 체인 종점이 존재하지 않기 때문에 프로토타입 체인에서 프로펕티를 검색할 때 **무한 루프**에 따짐

→ 따라서 무조건적으로 프로토타입을 교체할 수 없도록 __proto__접근자 프로퍼티를 통해 프로토타입에 접근하고 교체하도록 구현되어 있다.

▼ __proto__접근자 프로퍼티를 코드 내에서 직접 사용하는 것은 권장하지 않는 이유

직접 상속을 통해 object.prototype을 상속받지 않는 객체를 생성할 수도 있기 때문

▼ 프로토타입의 참조를 취득하고 싶다면?

object.getPrototyperOf 메서드를 사용

▼ 프로토타입을 교체하고 싶은 경우에는?

object.setprototypeOf 메서드를 사용

▼ prototype 프로퍼티란?

생성자 함수가 생성할 인스턴스(객체)의 프로토타입을 가리킨다.

→ 따라서 생성자 함수로서 호출할 수 없는 함수(=non-constructor인 화살표 함수, ES6 메서드 축약 표현으로 정의한 메서드)는 prototype 프로퍼티를 소유하지 않으며 프로토타입도 생성하지 않는다.

결론: 모든 객체가 가지고 있는 __proto__접근자 프로퍼티와 함수 객체만이 가지고 있는 prototype 프로퍼티는 결국 동일한 프로토타입을 말함.

But, 이들 프로퍼티를 사용하는 주체는 다름(모든 객체이냐 생성자 함수이냐)

▼ construnctor 프로퍼티가 가리키는 것은?

prototype 프로퍼티로 자신을 참조하고 있는 생성자 함수

(연결은 함수 객체가 생성될 때 이뤄진다.)

19.4 리터럴 표기법에 의해 생성된 객체의 생성자 함수와 프로토타입

▼ 추상연산이란?

ECMAScript 사양에서 내부 동작의 구현 알고리즘을 표현한 것으로 설명을 위해 사용되는 함수와 유사한 의사 코드라고 볼 수 있다.

▼ object 생성자 함수에 의한 객체 생성

인수가 전달되지 않았을 때 추상 연산 OrdinaryobjectCreate를 호출하여 빈 객체를 생성

```
let obj = new Object();
console.log(obj); //{}
```

▼ object 생성자 함수 호출 vs 객체 리터럴의 평가

추상 연산 OrdinaryobjectCreate를 호출하여 빈 객체를 생성하는 점에서 동일

new.target의 확인이나 프로퍼티를 추가하는 처리 등 세부 내용이 다름

→ 객체 리터럴에 의해 생성된 객체는 object 생성자 함수가 생성한 객체가 아님

즉, 프로토타입과 생성자 함수는 단독으로 존재할 수 없고 언제나 쌍으로 존재

▼ 리터럴 표기법에 의해 생성된 객체의 생성자 함수와 프로토타입

리터럴 표기법	생성자함수	프로토타입
객체 리터럴	object	object.prototype
함수 리터럴	Function	Function.prototype
배열 리터럴	Array	Array.prototype
정규 표현식 리터럴	RegExp	RegExp.prototype

19.5 프로토타입의 생성 시점

▼ 프로토타입의 생성 시점

프로토타입은 생성자 함수가 생성되는 시점에 더불어 생성

→ 프로토타입과 생성자 함수는 단독으로 존재할 수 없고 언제나 쌍으로 존재하기 때문

▼ 생성자 함수 종류

▼ 사용자가 직접 정의한 **사용자 정의 생성자 함수**

생성자 함수로서 호출 할 수 있는 함수

⇒ constructor는 함수 정의가 평가되어 함수 객체로 생성하는 시점에 프로토타입도 더불어 생성

프로토타입도 객체이고 모든 객체는 프로토타입을 가지므로 프로토타입도 자신의 프로토타입을 갖는다.

생성된 프로토타입의 프로토타입은 object.prototype

하지만 생성자 함수로서 호출할 수 없는 함수(non-constructor)는 프로토타입 생성 X

▼ 자바스크립트가 기본 제공하는 **빌트인 생성자 함수**

일반 함수와 마찬가지로 빌트인 생성자 함수가 생성되는 시점에 프로토타입이 생성

모든 빌트인 생성자 함수는 전역 객체가 생성되는 시점에 생성

객체가 생성되기 이전에 생성자 함수와 프로토타입은 이미 객체화되어 존재

이후 생성자 함수 또는 리터럴 표기법으로 객체 생성 시 프로토타입은 생성된 객체의 내부 슬롯에 할당된다. → 생성된 객체는 프로토타입을 상속받음

▼ 전역 객체란?

코드가 실행되기 이전 단계에 자바스크립트 엔진에 의해 생성되는 특수한 객체

클라이언트 사이드 환경(브라우저) = window

서버 사이트 환경(Node.js) = global

표준 빌트인 객체(Object, String, Number, Function, Array ..)
환경에 따른 호스트 객체(클라이언트 Web AP or Node.js의 호스트 API)
var 키워드로 선언한 전역 변수와 전역 함수
⇒ 프로퍼티로 갖는다.

Math, Reflect, JSON을 제외한 표준 빌트인 객체는 모두 생성자 함수

19.6 객체 생성 방식과 프로토타입의 결정

▼ 객체 생성 방법

1. 객체 리터럴
2. object 생성자 함수
3. 생성자 함수
4. object.create 메서드
5. 클래스(ES6)

모든 객체는 각 방식마다 세부적인 객체 생성 방식의 차이는 있으나,
추상 연산 OrdinaryIbjectCreate에 의해 생성된다는 공통점이 있다.

프로토타입은 추상 연산 OrdinaryObject에 전달되는 인수에 의해 결정
→ 인수는 객체가 생성되는 시점에 객체 생성 방식에 의해 결정

▼ 객체 리터럴에 의해 생성된 객체의 프로토타입

객체 리터럴을 평가하여 객체를 생성할 때 추상 연산 OrdinaryObject 호출
객체 리터럴에 의해 생성되는 객체의 프로토타입은 Object.prototype

▼ Object 생성자 함수에 의해 생성된 객체의 프로토타입

Object 생성자 함수를 인수 없이 호출하면 빈 객체 생성

Object 생성자 함수를 호출하면 객체 리터럴과 마찬가지로 추상 연산 OrdinaryObject 호출

Object 생성자 함수에 의해 생성되는 객체의 프로토타입은 Object.prototype

▼ 객체 리터럴과 Object 생성자 함수의 차이

프로퍼티를 추가하는 방식이 다름

객체 리터럴 → 객체 리터럴 내부에 프로퍼티를 추가

Object 생성자 함수 → 일단 빈 객체를 생성한 이후 프로퍼티를 추가

▼ 생성자 함수에 의해 생성된 객체의 프로토타입

다른 객체 생성 방식과 마찬가지로 추상 연산 OrdinaryObject 호출

추상 연산 OrdinaryObject에 전달되는 프로토타입은 생성자 함수의 prototype에 바인딩되어 있는 객체

프로토타입은 객체로 일반 객체와 같이 프로토타입에도 프로퍼티를 추가/삭제 가능

→ 이렇게 추가/삭제된 프로퍼티는 프로토타입 체인에 즉각 반영된다.

19.7 프로토타입 체인

▼ 프로토타입 체인이란?

자바스크립트는 객체의 프로퍼티에 접근하려 할 때 해당 객체에 접근하려는 프로퍼티가 없다면 [[prototype]] 내부 슬롯의 참조를 따라 **자신의 부모 역할을 하는 프로토타입의 프로퍼티를 순차적으로 검색**하는 것

프로토타입 체인은 자바스크립트가 객체지향 프로그래밍의 상속을 구현하는 메커니즘

▼ call 메서드란?

this로 사용할 객체를 전달하면서 함수를 호출

this로 사용할 때 me 객체를 전달하면서 Object./prototype.hasOwnProperty 메서드를 호출

▼ 프로토타입 체인의 종점

프로토타입 체인의 최상위에 위치하는 객체 = Object.prototype

→ 따라서 모든 객체는 Object.prototype을 상속 ⇒ 따라서 프로토타입 체인의 종점

object.prototype의 프로토타입([[Prototype]]) 내부 슬롯의 값은 null

만약 프로토타입 체인의 종점이 Object.prototype에서 프로퍼티를 검색할 수 없는 경우

→ undefined를 반환(에러가 발생하지 않음)

▼ 프로토타입 체인과 스코프 체인

자바스크립트 엔진은 프로토타입 체인을 따라 프로퍼티/메서드를 검색

객체 간의 상속 관계로 이루어진 프로토타입의 계층적인 구조에서 객체의 프로퍼티를 검색

프로퍼티가 아닌 식별자는 스코프 체인에서 검색

자바스크립트 엔진은 함수의 중첩 관계로 이루어진 스코프의 계층적 구조에서 식별자를 검색

프로토타입 체인은 **상속과 프로퍼티 검색을 위한 메커니즘**

스코프 체인은 **식별자 검색을 위한 메커니즘**

스코프 체인과 프로토타입 체인은 서로 연관없이 별도로 동작하는 것이 아닌 서로 협력하여 식별자와 프로퍼티를 검색하는데 사용

19.8 오버라이딩과 프로퍼티 섀도잉

▼ 프로토타입과 인스턴스의 프로퍼티

프로토타입이 소유한 프로퍼티 = 프로토타입 프로퍼티

인스턴스가 소유한 프로퍼티 = 인스턴스 프로퍼티

-인스턴스(객체) 생성한 후 인스턴스에 메서드 추가할 때

프로토타입 프로퍼티와 같은 이름의 프로퍼티를 인스턴스에 추가하면 프로토타입 체인을 따라 프로토타입 프로퍼티를 검색하여 **인스턴스 프로퍼티로 추가**

-인스턴스(객체) 생성한 후 인스턴스에 메서드 삭제할 때

하위 객체를 통해 프로토타입의 프로퍼티를 변경 또는 삭제하느 것은 불가능

하위 객체를 통해 프로토타입에 get 엑세스는 허용되나 set 엑세스는 허용되지 않는다

변경 또는 삭제하려면 하위 객체를 통해 프로토타입 체인으로 접근하는 것이 아닌 **프로토타입에 직접 접근**

▼ 프로퍼티 섀도잉이란?

상속 관계에 의해 프로퍼티가 가려지는 현상

▼ 오버라이딩

상위 클래스가 가지고 있는 메서드를 하위 클래스가 재정의하여 사용하는 방식

▼ 오버로딩

함수의 이름은 동일하지만 매개변수의 타입 또는 개수가 다른 메서드를 구현하고 매개변수에 의해 메서드를 구별하여 호출하는 방식

자바스크립트는 오버로딩을 지원하지 않지만 arguments 객체를 사용하여 구현 가능

19.9 프로토타입의 교체

▼ 프로토타입 교체

프로토타입은 임의의 다른 객체로 변경가능

부모 객체인 프로토타입을 동적으로 변경할 수 있음

프로토타입은 생성자 함수 또는 인스턴스에 의해 교체할 수 있음

▼ 생성자 함수에 의한 프로토타입의 교체

생성자 함수가 생성할 객체의 프로토타입을 객체 리터럴로 교체

프로토타입으로 교체한 객체 리터럴에는 constructor 프로퍼티가 없음

→ 자바스크립트 엔진이 프로토타입을 생성할 때 암묵적으로 추가한 프로퍼티

프로토타입을 교체하면 constructor 프로퍼티와 생성자 함수 간의 연결 파괴

→ 프로토타입으로 교체한 객체 리터럴에 constructor 프로퍼티를 추가하여 프로토타입의 constructor 프로퍼티를 되살린다.

▼ 인스턴스에 의한 프로토타입의 교체

인스턴스의 __proto__접근자 프로퍼티를 통해 프로토타입 교체 가능

생성자 함수의 prototype 프로퍼티에 다른 임의의 객체를 바인딩한다

= 이미 생성된 객체의 프로토타입을 교체

마찬가지로 프로토타입으로 교체한 객체에는 constructor 프로퍼티가 없기에 연결 파괴

▼ 생성자 함수와 인스턴스에 의한 프로토타입 교체 차이

Person 생성자 함수의 prototype 프로퍼티가 교체된 프로토타입을 가리킨다.

Person 생성자 함수의 prototype 프로퍼티가 교체된 프로토타입을 가리키지 않는다.

프로토타입 교체를 통해 객체 간의 상속 관계를 동적으로 변경하는 것은 번거롭기에 프로토타입은 직접 교체하지 않는 것이 좋다. ⇒ 직접 상속이 더 편리하고 안전

19.10 instanceof 연산자

▼ instanceof 연산자란?

이항 연산자로서 좌변에 객체를 가리키는 식별자로 우변에 생성자 함수를 가리키는 식별자를 피연산자로 받는다.

만약, 우변의 피연산자가 함수가 아닌 경우 TypeError 발생

```
객체 instanceof 생성자 함수
```

우변의 생성자 함수의 prototype에 바인딩된 객체가 좌변의 객체의 프로토타입 체인 상에 존재하면 true, 그렇지 않으면 false

▼ instanceof 연산자의 역할

생성자 함수의 prototype에 바인딩된 객체가 프로토타입 체인 상에 존재하는지 확인

따라서 생성자 함수에 의해 프로토타입이 교체되어 constructor 프로퍼티와 생성자 함수 간의 연결이 파괴되어도 생성자 함수의 prototype 프로퍼티와 프로토타입 간의 연

결은 파괴되지 않음 → instanceof 연산자는 아무런 영향을 받지 않음

19.11 직접 상속

▼ Object.create 메서드

첫번째 매개변수: 생성할 객체의 프로토타입으로 지정할 객체 전달

→ 객체를 생성하면서 직접적으로 상속을 구현

▼ 메서드의 장점

1. new 연산자가 없이도 객체 생성
2. 프로토타입을 지정하면서 객체를 생성
3. 객체 리터럴에 의해 생성된 객체도 상속받을 수 있다.

두번째 매개변수: 생성할 객체의 프로퍼티 키와 프로퍼티 디스크립터 객체로 이뤄진 객체 전달(옵션으로 생략 가능)

▼ Object.prototype의 빌트인 메서드를 객체가 직접 호출하는 것을 권장하지 않는 이유

Object.create 메서드를 통해 프로토타입 체인의 종점에 위치하는 객체 생성이 가능하기 때문

19.12 정적 프로퍼티/메서드

▼ 정적 프로퍼티/메서드란?

생성자 함수로 인스턴스를 생성하지 않아도 참조/호출할 수 있는 프로퍼티/메서드

생성자 함수가 생성한 인스턴스로 참조/호출할 수 없다.

▼ 생성자 함수가 생성한 인스턴스 vs 정적 프로퍼티/메서드가 생성한 인스턴스

자신의 프로토타입 체인에 속한 객체의 프로퍼티/메서드에 접근가능

프로토타입 체인에 속한 객체의 프로퍼티/메거드가 아니므로 인스턴스로 접근 불가능

▼ 정적 메서드 특징

인스턴스/프로토타입 메서드 내에서 this를 사용하지 않는다면 메서드는 정적 메서드로 변경할 수 있음

프로토타입 메서드를 호출하려면 → 인스턴스를 생성해야 함

정적 메서드를 호출하려면 → 인스턴스를 생성하지 않아도 호출 가능

19.13 프로퍼티 존재 확인

▼ in 연산자란?

in 연산자는 객체 내에 특정 프로퍼티가 존재하는지 여부를 확인

in 연산자는 확인 대상 객체의 프로퍼티 뿐 아니라 확인 대상 객체가 상속받은 모든 프로토타입의 프로토타입의 프로퍼티를 확인하므로 주의 필요

= in 연산자 대신 Reflect.has 메서드 사용 가능

19.14 프로퍼티 열거

▼ for..in 문이란?

객체의 모든 프로퍼티를 순회하며 열거하기 위해서는 for...in 문을 사용한다.

객체의 프로퍼티 개수만큼 순회하고 순화한 변수에 프로퍼티 키를 할당

→ 프로퍼티 어트리뷰트 [[Enumerable]]의 값이 true인 프로퍼티를 순회하며 열거

in 연산자처럼 순회 대상 객체의 프로퍼티뿐 아니라 상속받은 프로토타입의 프로퍼티도 열거

→ 상속받은 프로퍼티는 제외하고 객체 자신의 프로퍼티만 열거하려면 Object.prototype.hasOwnProperty 메서드를 사용하여 객체 자신의 프로퍼티인지 확인

for..in 문은 프로퍼티를 열거할 때 순서를 보장하지 않음

▼ Object.keys/values/entrites 메서드

for...in문은 객체 자신의 고유 프로퍼티뿐 아니라 상속받은 프로퍼티도 열거

하지만 객체 자신의 고유 프로퍼티만 열거하기 위해서는 for..in문 보다
Object.keys/values/entrites 메서드 사용하기

▼ Object.keys 메서드

객체 자신의 열거 가능한 프로퍼티 **키를 배열**로 반환

▼ Object.values 메서드

객체 자신의 열거 가능한 프로퍼티 **값을 배열**로 반환

▼ Object.entrites 메서드

객체 자신의 열거 가능한 프로퍼티 **키와 값의 쌍의 배열을 배열에 담아** 반환