클래스

- 모던 자바스크립트에 도입된 **클래스(class) 문법을 사용**하면 객체 지향 프로그래밍에서 사용되는 다양한 기능을 자바스크립트에서도 사용 가능

클래스와 기본 문법

클래스 정의 문법

```
// class 키워드와 함께 클래스 이름 짓기
class MyClass {
    // 생성자 메서드 정의
    constructor() { ... }
    // 클래스에 필요한 여러 메서드를 정의
    method1() { ... }
    method2() { ... }
    method3() { ... }
    // ...
}
```

- 클래스를 정의하고, 생성자 함수를 호출하듯이 **new 키워드와 함께 클래스이름을 함수처럼 호출하면 객체가 생성**됨
- 객체의 기본 상태를 설정해주는 **생성자 메서드(constructor)는 new에 의해 자동으로 호출**되므로 특별한 절차 없이 객체를 초기화 가능

```
class User {
    // 생성자 메서드는 new User(...) 명령어에 의해서 자동으로 호출됨
    constructor(name) {
        // 전달받은 인수를 이용해서 필요한 속성값을 초기화
        this.name = name;
    }
    sayHi() {
        alert(this.name);
    }
}

// 객체 생성
let user = new User("John");
// 클래스 내부에서 정의한 메서드 사용 가능
user.sayHi();
```

위의 코드에서 `new User("John")` 명령어를 호출한 이후 다음과 같은 일이 일어남

```
1. 새로운 객체가 생성됨
2. 넘겨받은 인수와 함께 constructor 메서드가 자동으로 실행되며 인수 "John"이 this.name에 할당됨
```

- 자바스크립트에서 **클래스는 내부적으로는 생성자 함수로 취급**됨 (클래스의 constructor 메서드 => 생성자 함수)

```
class User {
    constructor(name) { this.name = name; }
    sayHi() { alert(this.name); }
}

// User는 함수
console.log(typeof User); // function
// User.prototype에 sayHi 함수가 정의되어 있는 것을 확인 가능
console.log(User.prototype); // { constructor: User, sayHi: f }
```

- class 문법을 통해 내부적으로 수행되는 일들은 다음과 같음

```
1. User라는 이름을 가진 함수를 만들고 함수 본문은 생성자 메서드(constructor)에서 가져옴,
생성자 메서드가 없으면 "본문이 비워진 상태"로 함수가 생성됨
2. sayHi와 같은 클래스 내에서 정의한 메서드를 함수의 prototype 객체에 추가함
```

class 대신 생성자 함수와 프로토타입 객체를 이용하는 코드

```
// 1. User라는 이름을 가진 생성자 함수를 만들고 함수 본문은 생성자 메서드 constructor에서
가져오기
function User(name) {
    this.name = name;
}
// 만약 constructor 메서드의 내용이 비어있다면 다음과 같은 "본문이 비워진" 생성자 함수가
정의됨
// function User(name) {}

// 2. 클래스 내에서 사용할 메서드를 User.prototype에 정의하기
User.prototype.sayHi = function() { alert(this.name); }
```

내용 정리

```
class User {
    constructor(name) { this.name = name; }
    sayHi() { alert(this.name); }
}

// 클래스는 함수
alert(typeof User); // function

// 클래스는 "생성자 메서드"와 동일
alert(User === User.prototype.constructor); // true

// 클래스 내부에서 정의한 메서드는 User.prototype에 저장
alert(User.prototype.sayHi); // alert(this.name);

// 현재 프로토타입에는 메서드가 두 개(생성자, sayHi) 존재함
alert(Object.getOwnPropertyNames(User.prototype)); // constructor, sayHi
```

클래스 표현식 (@)

- 함수처럼 클래스도 값처럼 취급하므로 다른 표현식 내부에서 **정의, 전달, 반환, 할당이 가능**

```
// 익명 클래스를 정의하고 User 변수에 대입
let User = class {
    sayHi() {
        alert("Hello");
    }
};

// 이후 변수의 이름을 가지고 참조 가능
new User().sayHi();
```

- 자바스크립트에서는 **클래스를 동적으로 생성**하는 것도 허용함

```
function makeClass(phrase) {
    // 클래스를 선언하고 이를 반환함
    return class {
        sayHi() {
            alert(phrase);
        };
    };
}

// "클래스"를 반환 받음 ("객체"가 반환된 것이 아님을 유의)
let User = makeClass("Hello");

// 반환받은 클래스를 호출하여 객체 생성 가능
new User().sayHi(); // Hello
```

계산된 메서드 이름

- 객체의 속성 이름에 대괄호와 식을 사용할 수 있는 것과 마찬가지로, 대괄호를 이용해 **계산된 메서드 이름(computed method name)**을 만들 수 있음
    - 대괄호 내부에 표현식이 평가되고, 문자열로 변경된 후 해당 이름을 이용하여 메서드 정의

```
class User {
    // 계산된 메서드 이름 방식을 통해 sayHi 메서드를 정의
    // 대괄호 내부의 식이 평가된 결과값이 'sayHi'이므로 해당 이름의 메서드가 정의됨
    ['say' + 'Hi']() {
        alert("Hello");
    }
}

new User().sayHi();
```

클래스 필드

- **클래스 필드(class field)**라는 문법을 사용해서 개별 객체에 포함시킬 속성을 추가할 수 있음

```
class User {
    // 클래스 필드 문법 (클래스 중괄호 내부에 대입문 쓰기)
    name = "John";

    sayHi() {
        alert(`Hello, ${this.name}!`);
    }
}

new User().sayHi(); // Hello, John!
```

- 클래스 필드와 생성자 메서드에서 동시에 속성을 정의 가능

```
class MyClass {
    // prop1, prop2 속성은 클래스 필드 방식으로 정의
    prop1 = 'prop1';
    prop2 = 1234;

    constructor(prop3) {
        // prop3 속성은 생성자 메서드에서 정의
        this.prop3 = prop3;
    }
}

const mc = new MyClass({ hello: "world" });
console.log( mc.prop1 ); // "prop1"
console.log( mc.prop2 ); // 1234
console.log( mc.prop3 ); // { hello: "world" }
```

- 클래스 필드로 선언한 속성값들은 **prototype 객체가 아닌 개별 객체에 적용**됨

```
class User {
    name = "John";
}

let user = new User();
// 개별 객체에 name 속성이 추가됨
alert(user.name); // "John"
// prototype에 추가되어 공통적으로 사용할 속성이 되는 것이 아님을 유의!
alert(User.prototype.name); // undefined
```

클래스 필드로 바인딩 된 메서드 만들기

- 자바스크립트의 함수는 호출 시점에 따라 다른 this를 갖을 수 있음
    - 이는 함수뿐만 아니라 메서드에도 적용되는 규칙으로 **메서드를 전달하여 다른 컨텍스트에서 호출하게 되면 this는 원래 객체를 참조하지 않게 됨**

```
class Button {
    constructor(value) {
        this.value = value;
    }
    click() {
        alert(this.value);
    }
}

let button = new Button("hello");

// 메서드 전달 (단, 메서드가 전달되어 콜백 함수에서 호출되므로 함수가 호출되는 시점에는
this 맥락(점 앞 객체)을 잃게 됨)
setTimeout(button.click, 1000); // undefined
```

- 클래스 필드에 메서드를 정의하며 **화살표 함수를 사용하면 객체마다 독립적인 함수를 만들고 함수의 this를 해당 객체에 바인딩**함
    - 따라서 **메서드가 실행되는 맥락과 관계 없이 this엔 항상 의도한 값(객체)이 들어가게 됨**
    - 클래스 필드의 이런 기능은 브라우저 환경에서 **메서드를 이벤트 리스너로 설정**해야 할 때 특히 유용함

```
class Button {
    constructor(value) {
        this.value = value;
    }
    // click 메서드를 화살표 함수를 이용해서 정의
    click = () => {
        alert(this.value);
    }
    nonBinded() {
        alert(this.value);
    }
}

let button1 = new Button("hello");

setTimeout(button1.click, 1000); // hello

let button2 = new Button("hello");

// click 메서드는 화살표 함수로 정의했으므로,
// 서로 독립적인 함수 객체 참조를 가지게 됨 (내용은 똑같지만 서로 참조가 다른 독립적인 함수)
console.log( button1.click === button2.click ); // false

// nonBinded 메서드는 화살표 함수를 이용하지 않고 정의했으므로,
// 화살표 함수로 정의하지 않았으므로 함수 객체를 공유
console.log( button1.nonBinded === button2.nonBinded ); // true

// click은 prototype 객체에도 포함되지 않음
console.log( Button.prototype.click );
// 그러나 nonBinded는 prototype 객체에 포함됨
console.log( Button.prototype.nonBinded );
```

크롬 새 탭에서 실행해보기

```
// 브라우저 환경에서 메서드를 이벤트 리스너로 설정하는 상황
class Button {
    constructor(value) {
        this.value = value;
    }
    click = () => {
        alert(this.value);
    }
}
let button = new Button("hello");
document.write('<button>click</button>');

// 클릭되는 시점에 호출할 콜백 함수로 등록 (즉, 호출 시점엔 맥락이 사라지게 됨)
// 하지만 클래스 필드의 형태 및 화살표 함수로 정의했으므로 this값(button 객체)을 맥락에 관계 없이 유지
document.getElementsByTagName('button')[0].onclick = button.click;

// Q) 버튼 눌러보고 결과 확인해보기
```

클래스 상속

- 클래스 확장을 위해서 **extends 키워드**를 사용

```
class Animal {
    constructor(name) {
        this.speed = 0;
        this.name = name;
    }
    run(speed) {
        this.speed = speed;
        alert(`${this.name} 은/는 속도 ${this.speed}로 달립니다.`);
    }
    stop() {
        this.speed = 0;
        alert(`${this.name} 이/가 멈췄습니다.`);
    }
}

// Animal을 상속받는 Rabbit 클래스 정의
class Rabbit extends Animal {
    hide() {
        alert(`${this.name} 이/가 숨었습니다!`);
    }
}

let rabbit = new Rabbit("흰 토끼");

// Animal 클래스에 정의된 메서드 사용 가능
rabbit.run(5); // 흰 토끼 은/는 속도 5로 달립니다.
// 고유 메서드도 당연히 사용 가능
rabbit.hide(); // 흰 토끼 이/가 숨었습니다!

let animal = new Animal("동물")

// class ≒ 생성자 함수, 따라서 prototype의 constructor를 통해 자기 참조 가능
```

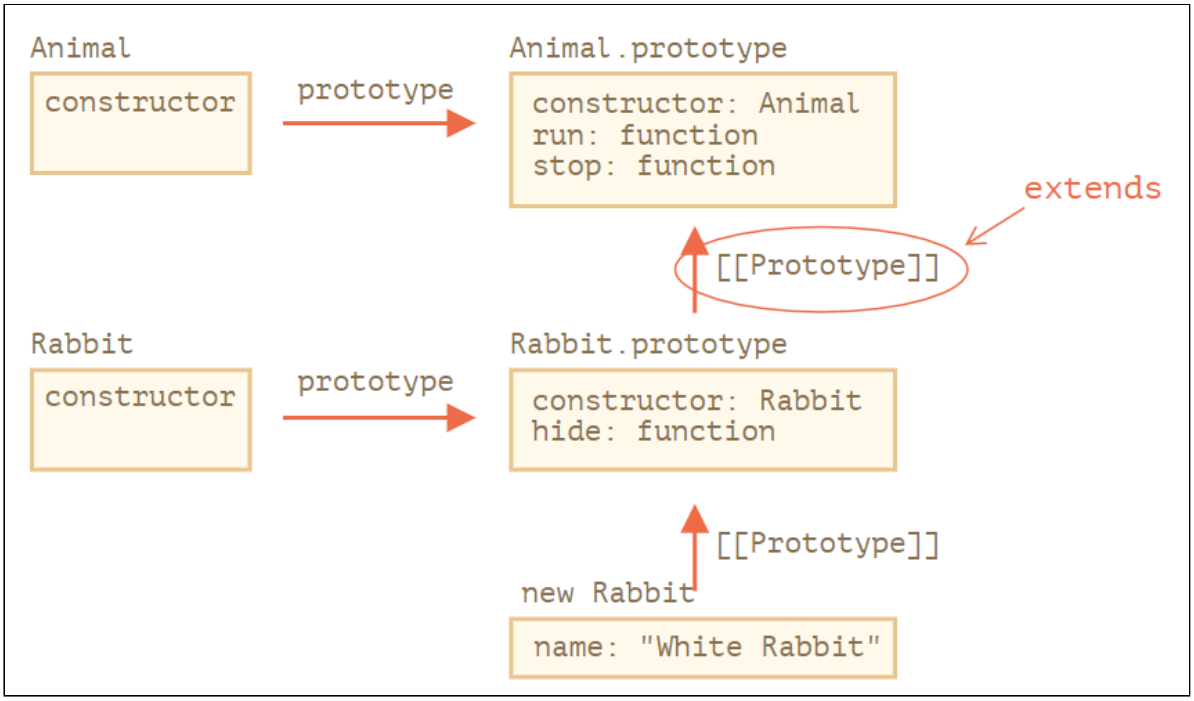
```
console.log( Animal.prototype.constructor ); // Animal 함수
console.log( Rabbit.prototype.constructor ); // Rabbit 함수

// 생성자 함수 확인
console.log( rabbit.constructor ); // Rabbit 함수
console.log( animal.constructor ); // Animal 함수

// 프로토타입 객체 확인
console.log( rabbit.__proto__ === Rabbit.prototype ); // true
console.log( animal.__proto__ === Animal.prototype ); // true

// (extends 키워드에 의해서) Rabbit의 prototype 객체의 __proto__ 값이
Animal.prototype을 가리키게 되므로 프로토타입 체인을 통해 부모 클래스 속성과 메서드 접근
가능
console.log( Rabbit.prototype.__proto__ === Animal.prototype ); // true
```

- **키워드 extends는 프로토타입을 기반으로 동작**함
    - extends는 Rabbit.prototype.[[Prototype]] (즉, Rabbit.prototype.__proto__)을 Animal.prototype으로 설정함



(ES6 등장 이전) class 키워드의 도움 없이 상속 구현하기

```
/* Animal 클래스 정의 */
function Animal(name) {
    this.speed = 0;
    this.name = name;
}

// 메서드 정의
Animal.prototype.run = function(speed) {
    this.speed = speed;
    alert(`${this.name} 은/는 속도 ${this.speed}로 달립니다.`);
}
Animal.prototype.stop = function() {
    this.speed = 0;
    alert(`${this.name} 이/가 멈췄습니다.`);
}
```

```
/* Rabbit 클래스 정의 */
function Rabbit(name, color) {
    // 부모 생성자 호출 (super() 메소드 호출과 비슷한 역할 수행)
    Animal.call(this, name)

    // 이후 Rabbit 객체의 고유 속성값 정의
    this.color = color
}

/* extends 키워드에서 수행하는 상속 관련 작업 진행 */
// 1. Animal 객체 생성하여 prototype 객체로 대입 (Object.create 메서드 사용해도 무방함)
// 이 때 prototype의 모습은 { speed: 0, name: undefined }
// 생성된 객체의 __proto__에는 Animal.prototype 객체가 저장되어 있으므로, 나중에 프로토타입 체인을 따라갈 때 해당 객체를 만날 수 있도록 설정됨
Rabbit.prototype = new Animal();
// 2. 단, 이 경우 Rabbit.prototype 객체의 constructor가 Animal 함수가 되어버리므로, 원래 참조해야 할 constructor 함수(Rabbit)를 참조하도록 복구 진행
Rabbit.prototype.constructor = Rabbit;

// 고유 메서드 정의
Rabbit.prototype.hide = function() {
    // this.stop(); // => super.stop();
    alert(`${this.name} 이/가 숨었습니다!`);
}

/* 객체 활용 코드 */
let rabbit = new Rabbit("흰 토끼", "흰색");

// class 키워드 사용하여 정의한 것과 똑같이 작동함
rabbit.run(5);
rabbit.hide();
alert( rabbit.color );

let animal = new Animal("동물")

console.log( Animal.prototype.constructor ); // Animal 함수
console.log( Rabbit.prototype.constructor ); // Rabbit 함수
console.log( rabbit.constructor ); // Rabbit 함수
console.log( animal.constructor ); // Animal 함수
console.log( rabbit.__proto__ === Rabbit.prototype ); // true
console.log( animal.__proto__ === Animal.prototype ); // true
console.log( Rabbit.prototype.__proto__ === Animal.prototype ); // true
```

메서드 오버라이딩

- 메서드를 자체적으로 정의하면, **상속받은 메서드가 아닌 자체 메서드가 사용**됨
- 부모 메서드를 호출하고 싶으면 **super 키워드** 이용

```
1. super 객체에 접근하여 부모 클래스에 정의된 메서드를 호출 가능
2. super를 메서드처럼 호출하면 "부모 생성자"를 호출하며 부모 생성자 호출은 자식 생성자 내부에서만 허용됨
```

```
class Animal {
```

```
    constructor(name) {
        this.speed = 0;
        this.name = name;
    }

    run(speed) {
        this.speed = speed;
        alert(`${this.name}가 속도 ${this.speed}로 달립니다.`);
    }

    stop() {
        this.speed = 0;
        alert(`${this.name}가 멈췄습니다.`);
    }
}

class Rabbit extends Animal {
    hide() {
        alert(`${this.name}가 숨었습니다!`);
    }
    stop() {
        // super 객체에 접근하여 부모 클래스의 stop 메서드를 호출
        super.stop();
        // 이후 고유 메서드 호출
        this.hide();
    }
}
```

생성자 오버라이딩

- 클래스가 다른 클래스를 상속받고 자체 constructor 메서드가 없는 경우엔 다음과 같이 **내용이 없는 constructor 메서드**가 만들어지게 됨
- 자동 생성된 생성자 메서드의 내용
    - super 키워드를 이용하여 부모 생정자 메서드를 호출
    - 부모 클래스의 생성자 메서드에 전달받은 인수를 모두 전달

```
class Rabbit extends Animal {
    // 특정 클래스를 상속받은 클래스 내부에 생성자 메서드가 정의되어 있지 않을 경우, 자동으로 만들어지는 생성자 메서드
    constructor(...args) {
        // 전달받은 모든 인수를 그대로 부모 생성자 메서드로 전달
        super(...args);
    }
}
```

- 상속 클래스의 생성자에선 **반드시 super를 이용하여 부모 생성자를 호출**해야 함

자체 생성자를 가졌지만 에러가 나는 코드

```
class Animal {
    constructor(name) {
        this.speed = 0;
        this.name = name;
```

```
    }
}

class Rabbit extends Animal {
    constructor(name, earLength) {
        // super를 이용한 부모 생성자 호출을 진행하지 않음 (에러 발생!)
        this.speed = 0;
        this.name = name;
        this.earLength = earLength;
    }
    // ...
}

// ReferenceError: Must call super constructor in derived class before accessing
'this' or returning from derived constructor
// 해석 => 참조에러: 반드시 상속받는 클래스에서 "this에 접근"하거나 "상속받는 클래스의 생성
자 메서드가 종료되기 전"까지 super 키워드를 이용한 부모 생성자 호출을 진행해야 합니다.
let rabbit = new Rabbit("흰 토끼", 10);
```

```
class Animal {
    constructor(name) {
        this.speed = 0;
        this.name = name;
    }
}

class Rabbit extends Animal {
    constructor(name, earLength) {
        // 상속 받은 클래스에서는 반드시 super를 통해 부모 생성자 호출해야 함!
        super(name);
        this.earLength = earLength;
    }
}

// 문제 없이 객체 생성 가능
let rabbit = new Rabbit("흰 토끼", 10);
```

정적 메서드와 정적 속성 (@)

- prototype이 아닌 **클래스 자체에도 메서드를 설정 가능하며 이를 통해 정적(static) 메서드를 정의**할 수 있음 (정적 메서드 => 클래스 이름을 통해 접근 가능한 메서드)
- 정적 메서드를 만들려면 앞에 **static 키워드**를 붙임

```
class User {
    // static 키워드를 붙여 메서드를 정의 => 정적 메서드 정의
    static staticMethod() {
        alert(this === User);
    }
}

// 객체가 아닌 클래스 이름으로 접근하여 호출 가능
User.staticMethod();
```

- 메서드를 **클래스 이름을 통해 직접 할당**해도 위의 코드와 완전히 동일하게 작동함

```
class User {}

// 속성값 직접 할당 (prototype 객체에 할당하는 것이 아님을 유의)
User.staticMethod = function() {
    console.log(this === User);
};

User.staticMethod();
```

- 정적 메서드는 **어떤 특정한 객체가 아닌 클래스에 속한 함수를 구현하고자 할 때 주로 사용**됨 (주로 유틸성 함수)

정적 메서드를 이용하여 팩토리 메서드를 구현한 코드 (팩토리 메서드 => 객체를 생성하는 용도로 사용하는 메서드)

```
class Article {
    constructor(title, date) {
        this.title = title;
        this.date = date;
    }

    // Article 객체를 생성하는 팩토리 메서드 정의
    static createTodays() {
        // Article.createTodays와 같은 형태로 호출했으므로, 여기서 this(점 앞 객체)는
Article 생성자 함수를 가리킴
        // (즉, 밑의 코드는 new Article(...)과 같은 결과를 수행하게 됨)
        return new this("Today's digest", new Date());
    }
}

let article = Article.createTodays();

console.log( article.title ); // Today's digest
```

정적 속성

- 클래스 필드 정의하듯이 하되, 앞에 static 키워드를 붙여서 정의

```
class Article {
    static publisher = "Ilya Kantor";
}

// 위의 코드는 클래스에 직접 속성을 설정하는 아래의 코드와 완전히 동일한 역할 수행
// Article.publisher = "Ilya Kantor";

// 정적 메서드와 마찬가지로 함수 이름으로 접근
console.log( Article.publisher );
```

정적 속성과 메서드 상속

- 정적 속성과 메서드는 자식 클래스에도 상속됨

```
class Animal {
    static planet = "지구";

    constructor(name, speed) {
        this.speed = speed;
        this.name = name;
    }

    run(speed = 0) {
        this.speed += speed;
        alert(`${this.name}가 속도 ${this.speed}로 달립니다.`);
    }

    static compare(animalA, animalB) {
        return animalA.speed - animalB.speed;
    }
}

// Animal을 상속받음
class Rabbit extends Animal {
    hide() {
        alert(`${this.name}가 숨었습니다!`);
    }
}

let rabbits = [ new Rabbit("흰 토끼", 10), new Rabbit("검은 토끼", 5) ];

// 정적 메서드도 상속됨
rabbits.sort(Rabbit.compare); // speed 순으로 정렬

// 정적 속성도 상속됨
alert(Rabbit.planet); // 지구

// 일반 메서드도 당연히 상속됨
rabbits[0].run(); // 검은 토끼가 속도 5로 달립니다.
```

- Rabbit extends Animal은 내부적으로 **두 개의 [[Prototype]]을 참조**하도록 설정 작업을 진행함

```
1. 함수 Rabbit은 프로토타입을 통해 함수 Animal을 상속 (정적 속성, 메서드 접근 가능)
2. Rabbit.prototype은 프로토타입을 통해 Animal.prototype을 상속 (일반 속성, 메서드 접근 가능)
```

```
// 정적 속성, 메서드 접근이 가능하도록 "함수 객체"의 __proto__ 값 조정
(Rabbit.prototype의 __proto__가 아님을 유의!)
// 상속 과정에서 Rabbit의 __proto__ 값으로 Animal "함수 객체"를 대입하여 Rabbit에 속성 값이 없으면 Animal에서 찾도록 유도함
alert( Rabbit.__proto__ === Animal ); // true

// Rabbit "함수 객체"에 compare 프로퍼티가 없으므로 __proto__를 참조하여 Animal 객체에 접근, 이후 compare 발견
alert( Rabbit.compare );

// 일반 메서드는 이전에 살펴본대로 프로토타입 체인을 따라가면서 속성, 메서드를 찾음
alert( Rabbit.prototype.__proto__ === Animal.prototype ); // true
```

리퍼런스

- 클래스 상속
  - https://levelup.gitconnected.com/prototypal-inheritance-the-big-secret-behind-classes-in-javascript-e7368e76e92a