JavaScript

◆ 클래스

정수아

Contents

01

클래스란?

객체와 클래스

❖ **객체(Object)**
- 실생활에 존재하는 실제적인 물건 또는 개념

❖ **속성(Attribute)**
- 객체가 가지고 있는 특성

❖ **메서드(Method)**
- 객체가 동작(행동)할 수 있도록 하는 함수

객체와 클래스

❖ **클래스**

- 객체가 가져야 할 기본적인 정보를 담은 코드
- 객체를 효율적으로 생성하기 위해 만들어진 구문
- 일종의 설계도

붕어빵틀
(클래스)

붕어빵
(인스턴스)

생성자 함수 vs 클래스

❖ **생성자 함수**
- new 키워드가 생략되면 일반 함수로 호출됨
- function 키워드로 정의

❖ **클래스**
- new 키워드 생략 시, 타입 에러 발생
- class 키워드로 정의

클래스 선언 및 객체 생성

❖ 클래스 선언 방식

```
class 클래스이름 {
    // 클래스내용
}
```

- 예시

```
class Rectangle {
    // 클래스내용
}
```

클래스 선언 및 객체 생성

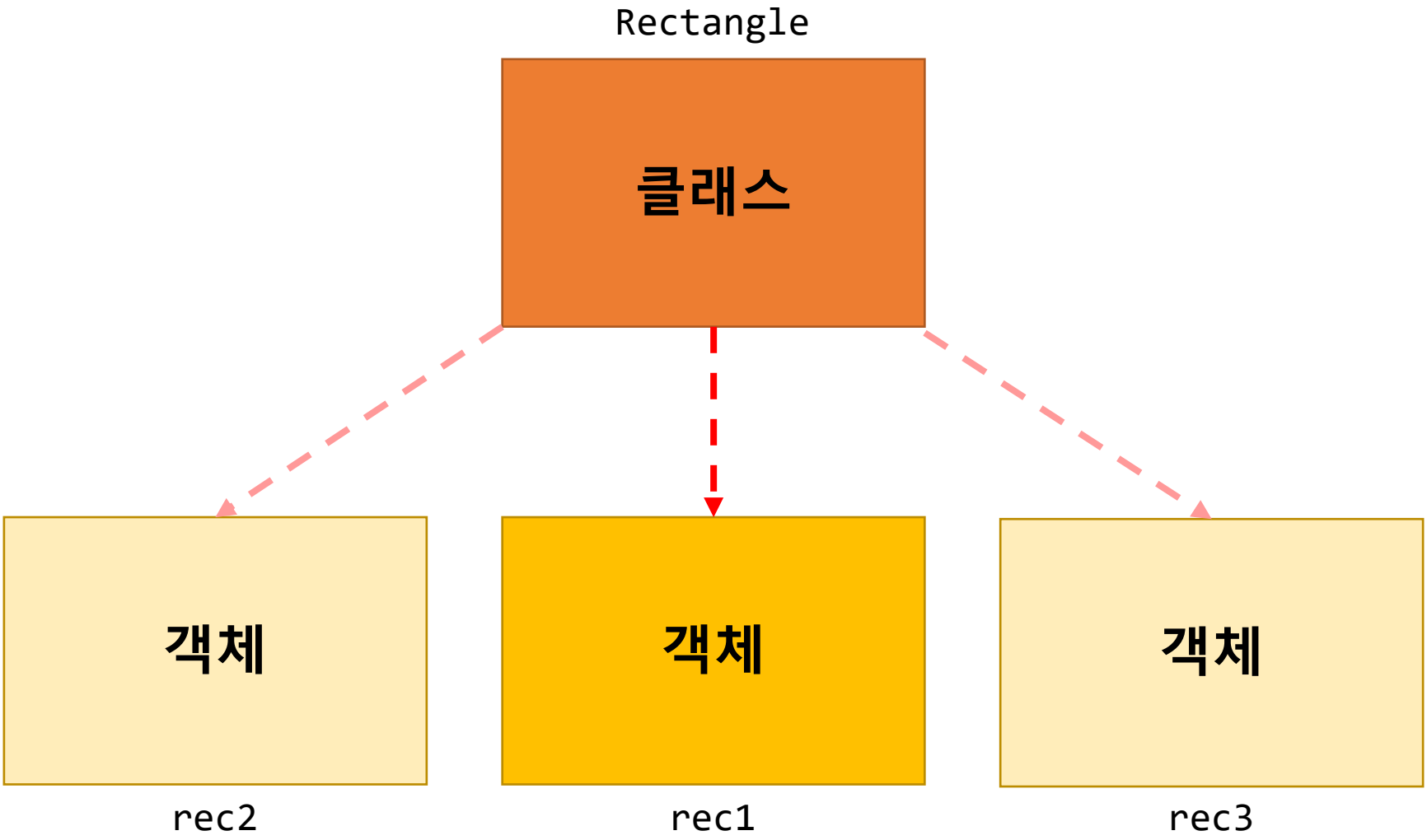
❖ 객체 생성

```
let 객체이름 = new 클래스이름();
```

- 예시

```
class Rectangle {
    // 클래스내용
}

let rec1 = new Rectangle();
```

클래스 선언 및 객체 생성



02

클래스 함수

클래스에 함수 추가하기

❖ **클래스 내부 함수 종류**

- 생성자 메서드
- 프로토타입 메서드
- 정적 메서드

생성자 메서드

❖ 생성자 메서드

- 객체를 생성하고 초기화하는 메서드
- 클래스 내에 최대 1개만 존재
- 생략 가능
  - 생략 시, 빈 생성자가 만들어짐

- 예시

```
class Rectangle {
    constructor() {
        // 객체 초기화
    }
}
```

생성자 메서드

❖ **클래스 외부에서 객체의 초기 프로퍼티 값 전달**

```
class Rectangle {
      // 생성자 메서드에 매개변수 선언
      constructor(w, h) {
         // 인스턴스 프로퍼티
         this.width = w;
         this.height = h;
      }
}

// 객체 생성 시, 초기값 전달
let rec1 = new Rectangle(100, 200);
```

생성자 메서드

❖ 생성자 메서드의 return문

- return문은 생략 해야 함
  - new 연산자가 클래스와 함께 호출되면 암묵적으로 객체를 반환해주기 때문

```
class Rectangle {
    constructor(w, h) {
        this.width = w;
        this.height = h;

        return object;
    }
}

// 객체 생성 시, 초기값 전달
let rec1 = new Rectangle(100, 200);
```

생성자 메서드

❖ 생성자 메서드의 return문

- 명시적으로 객체를 리턴하는 경우

```
class Rectangle {
    constructor(w, h) {
        this.width = w;
        this.height = h;

        return {};
    }
}

let rec1 = new Rectangle(100, 200);

console.log(rec1);          // 결과 : {} 빈 객체 리턴
```

생성자 메서드

❖ 생성자 메서드의 return문

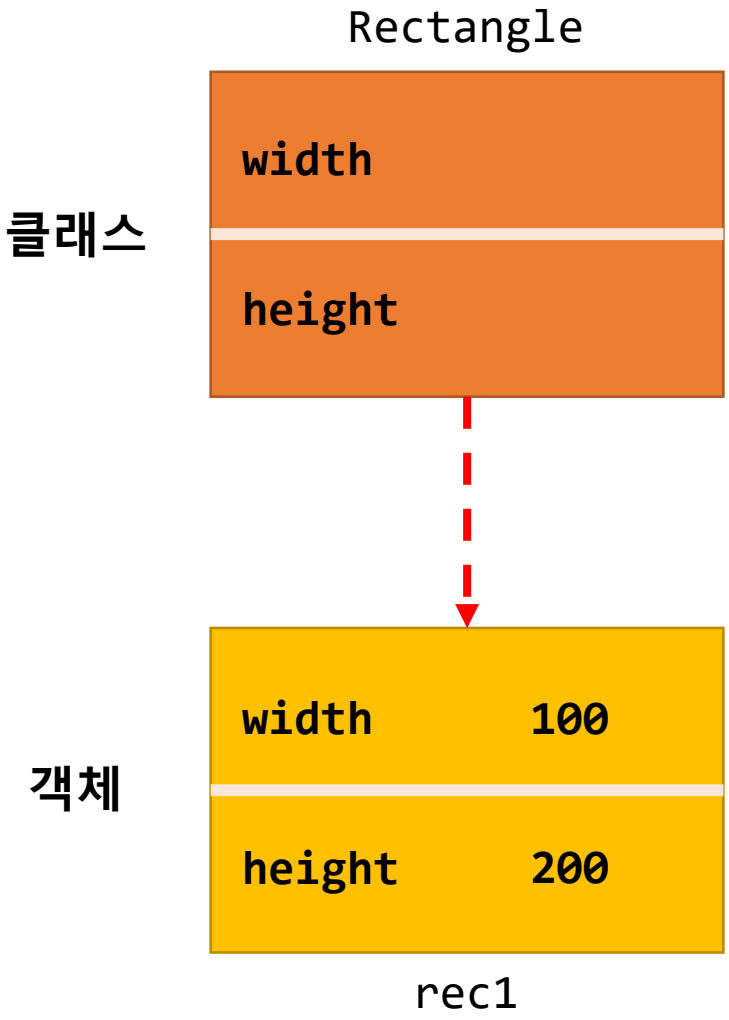
- 명시적으로 다른 값을 반환하는 경우

```
class Rectangle {
    constructor(w, h) {
        this.width = w;
        this.height = h;

        return "soo";
    }
}

let rec1 = new Rectangle(100, 200);

console.log(rec1);          // 결과 : Rectangle { width: 100, height: 200 }
```

[실습] 클래스로 객체 생성하기

```
class Rectangle {
      constructor(w, h) {
            this.width = w;
            this.height = h;
      }
}

// 가로 100, 세로 200인 사각형 객체 생성
let rec1 = new Rectangle(100, 200);

// 객체를 콘솔 화면에 출력
console.log(rec1);

// 객체를 브라우저 화면에 출력
document.write(JSON.stringify(rec1));
```

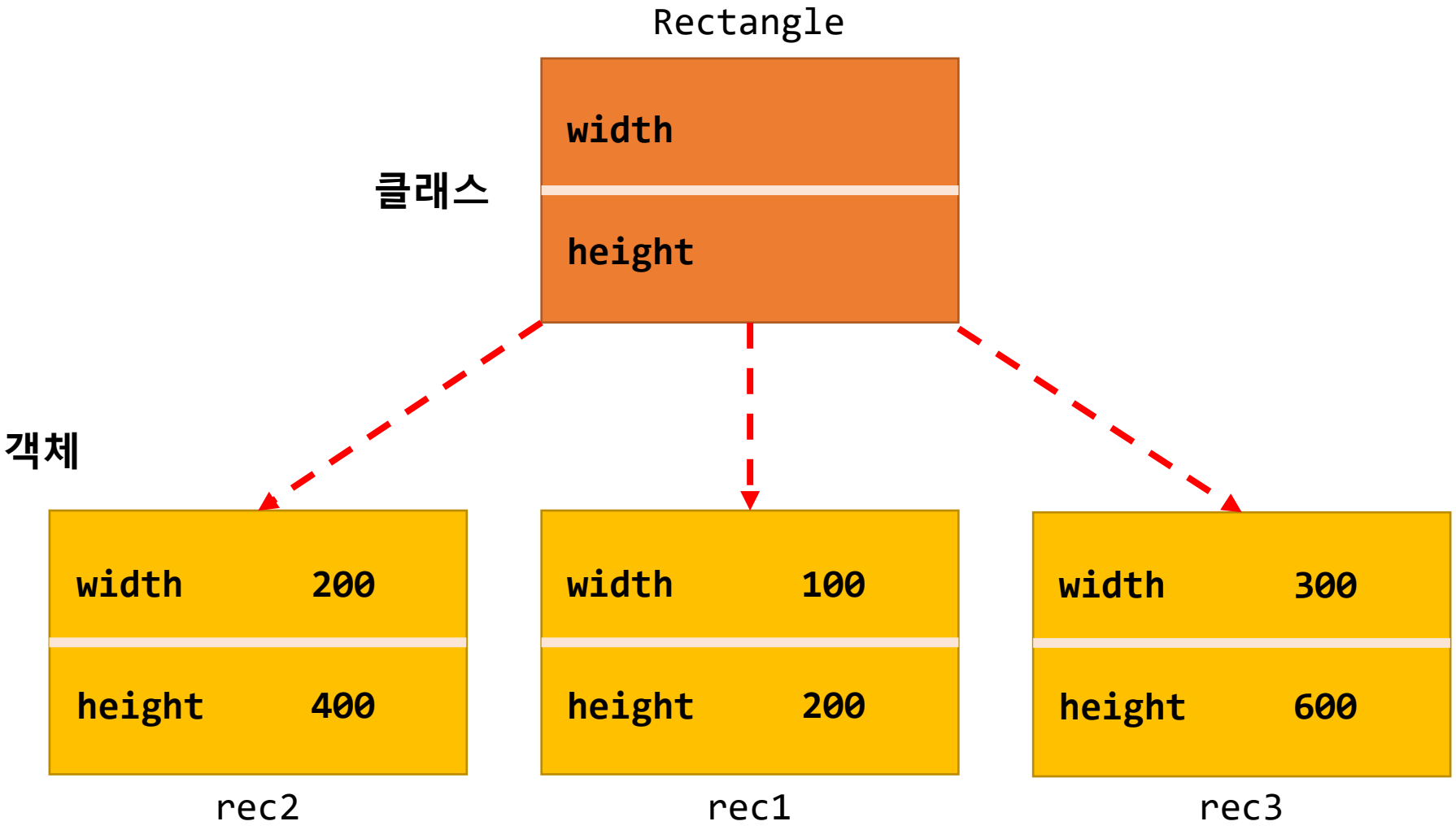
[실습] 클래스로 객체 생성하기



[실습] 클래스로 객체 생성하기

```
class Rectangle {
    constructor(w, h) {
        this.width = w;
        this.height = h;
    }
}

// 가로 100, 세로 200인 사각형 객체 생성
let rec1 = new Rectangle(100, 200);

// 가로 200, 세로 400인 사각형 객체 생성
let rec2 = new Rectangle(200, 400);

// 가로 300, 세로 600인 사각형 객체 생성
let rec2 = new Rectangle(300, 600);
```

[실습] 클래스로 객체 생성하기



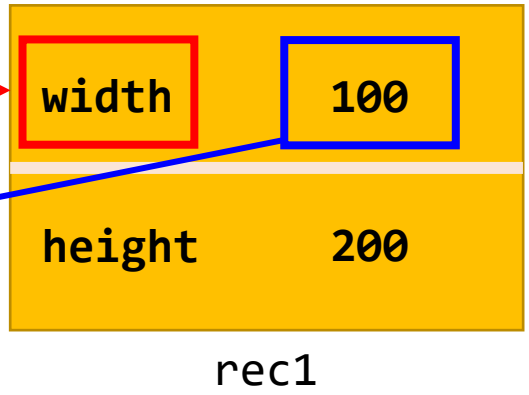
객체 값 사용하기

❖ 객체 값 접근

객체이름.프로퍼티이름

- 예시 - rec1 객체의 width 속성의 값에 접근

rec1.width



- 결과

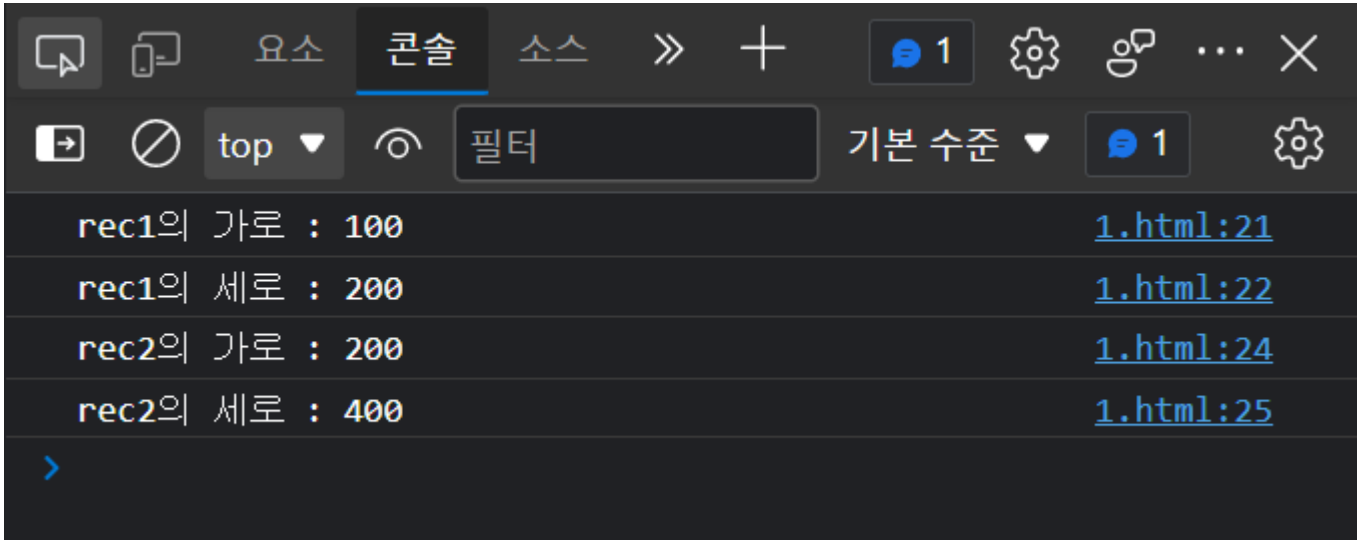
100

[실습] 객체 값 콘솔에 출력하기

```
class Rectangle {
    constructor(w, h) {
        this.width = w;
        this.height = h;
    }
}

// 가로 100, 세로 200인 사각형 객체 생성
let rec1 = new Rectangle(100, 200);

// 가로 200, 세로 400인 사각형 객체 생성
let rec2 = new Rectangle(200, 400);

// 콘솔 화면에 객체 값 출력
console.log("rec1의 가로 : " + rec1.width);
console.log("rec1의 세로 : " + rec1.height);

console.log("rec2의 가로 : " + rec2.width);
console.log("rec2의 세로 : " + rec2.height);
```

[실습] 객체 값 콘솔에 출력하기

❖ 실행 결과



프로토타입 메서드

❖ 프로토타입 메서드

- 클래스 내부에서 명시적으로 정의한 메서드

```
class Rectangle {
    constructor(w, h) {
        this.width = w;
        this.height = h;
    }

    // 프로토타입 메서드
    area() {
        console.log("사각형의 넓이를 구합니다.");
    }
}
```

프로토타입 메서드

❖ 프로토타입 메서드 호출

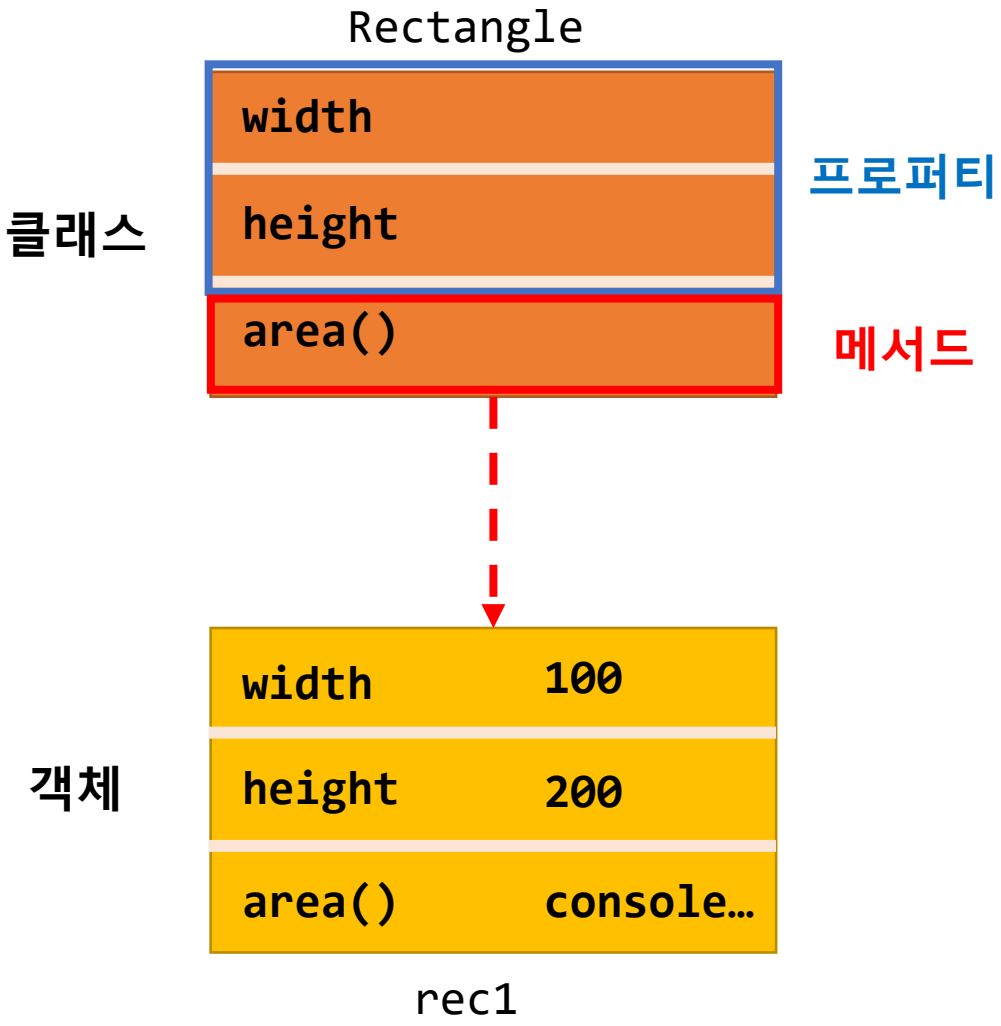
```
let rec1 = new rectangle(100, 200);

// 클래스 내부의 area() 메서드 호출
rec1.area();
```

프로토타입 메서드

❖ 프로토타입 메서드 - 화살표 함수

```
class Rectangle {
    constructor(w, h) {
        this.width = w;
        this.height = h;
    }

    area = () => {
        console.log("사각형의 넓이를 구합니다.");
    }
}
```

프로토타입 메서드


Rectangle
width
height
area()
프로퍼티
메서드
클래스
width
100
height
200
area()
console…
객체
rec1

정적 메서드

❖ 정적 프로퍼티와 정적 메서드

- 클래스 자체에 선언된 프로퍼티 또는 메서드
- 객체를 생성하지 않아도 호출 가능

- 선언 방법
  - static 키워드를 붙여서 생성

```
class Rectangle {
    static color = “red”;

    constructor(w, h) {
        this.width = w;
        this.height = h;
    }

    static area = () => {
        console.log("사각형의 넓이를 구합니다.");
    }
}
```

정적 메서드

❖ **정적 메서드 호출**

- 객체가 호출하지 않고, 클래스로 호출해야 함
- 클래스.정적메소드()

```
Rectangle.area();
```

정적 메서드

❖ 프로토타입 메서드와의 차이

구분	정적 메서드	프로토타입 메서드
프로토타입 체인	클래스	인스턴스
호출 방식	클래스로 호출	인스턴스로 호출
인스턴스 프로퍼티 참조 가능 여부	불가능	가능

[실습] 정적 메서드

❖ 인스턴스 프로퍼티를 참조하지 않고 면적 계산

```
class Rectangle {
    constructor(w, h) {
        this.width = w;
        this.height = h;
    }

    static area = (width, height) => {
        return width * height;
    }
}

console.log(Rectangle.area(100, 200));
```

- 실행 결과

```
20000
```

[실습] 프로토타입 메서드

❖ 인스턴스 프로퍼티를 참조하여 면적 계산

```
class Rectangle {
    constructor(w, h) {
        this.width = w;
        this.height = h;
    }

    area = () => {
        return this.width * this.height;
    }
}

let rec1 = new Rectangle(100, 200);
console.log(rec1.area());
```

- 실행 결과

```
20000
```

03

접근 제어

클래스 접근 제어(캡슐화)

❖ **캡슐화(Encapsulation)**

- 클래스 내부의 특정 프로퍼티나 메소드를 외부에서 참조하지 못하도록 숨기는 것

- 클래스 외부
  - 제한된 접근 권한을 제공

- 클래스 내부
  - 원하지 않는 외부의 접근에 대해 내부를 보호

- 방법
  - 캡슐화를 원하는 프로퍼티나 메서드 앞에 # 기호를 붙임

클래스 접근 제어(캡슐화)

```
class Rectangle {
  #color = "red";

  constructor(w, h) {
     this.width = w;
     this.height = h;
  }

  area = () => {
     console.log("사각형의 넓이를 구합니다.");
  }
}

var rec1 = new Rectangle(100, 200);

console.log(rec1.color);     // 결과 : undefined
```

- #color 변수는 rec1.color 방식으로 접근 불가

클래스 접근 제어(캡슐화)

```
class Rectangle {
  #color = "red";

  constructor(w, h) {
     this.width = w;
     this.height = h;
  }

  area = () => {
     console.log("사각형의 넓이를 구합니다.");
  }
}

var rec1 = new Rectangle(100, 200);

console.log(rec1.#color);
```

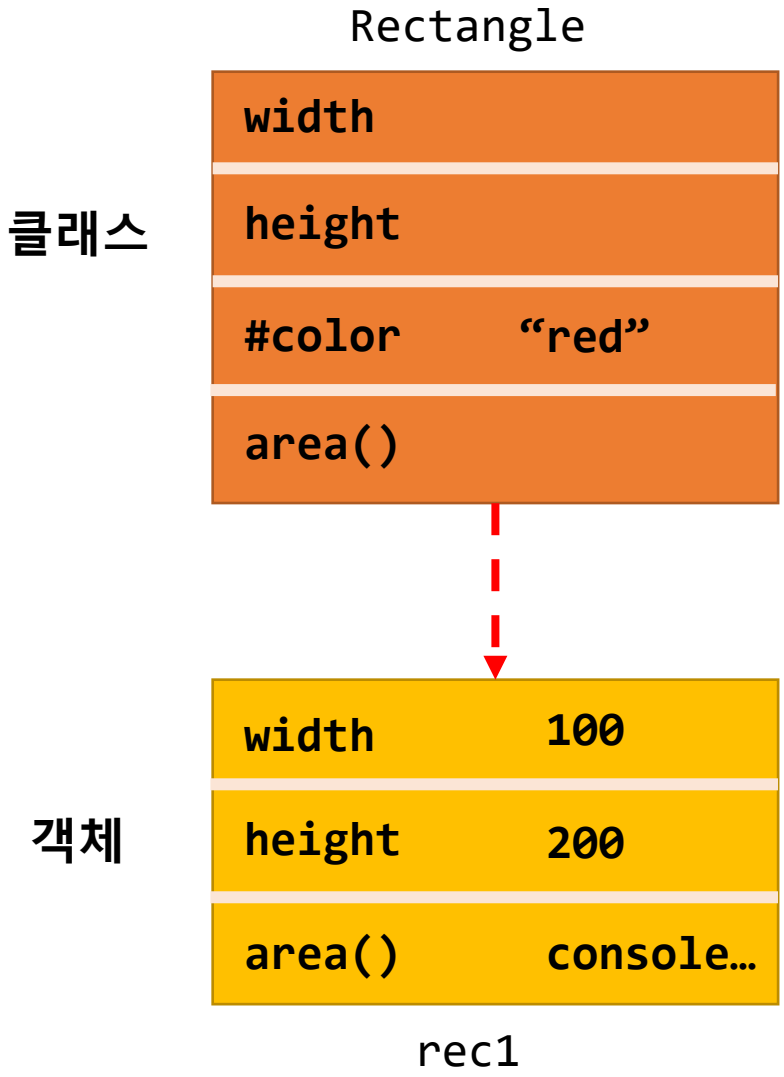
- 에러 : SyntaxError: Private field '#color' must be declared in an enclosing class

클래스 접근 제어(캡슐화)

```
class Rectangle {
  #color = "red";

  constructor(w, h) {
     this.width = w;
     this.height = h;
  }

  area = () => {
     console.log("사각형의 넓이를 구합니다.");
  }
}

var rec1 = new Rectangle(100, 200);

console.log(rec1);
```

- 결과
  - Rectangle { width: 100, height: 200, area: [Function: area] }

클래스


Rectangle
클래스
width
height
#color “red”
area()
객체
width 100
height 200
area() console…
rec1

04

접근자 프로퍼티

접근자 프로퍼티

❖ **데이터 프로퍼티**
- 키와 값으로 구성된 일반적인 프로퍼티

❖ **접근자 프로퍼티**
- 자체적으로는 값을 갖지 않고 다른 데이터 프로퍼티의 값을 읽거나 저장할 때 호출되는 접근자 함수로 구성된 프로퍼티

접근자 프로퍼티

❖ 접근자 프로퍼티

종류	설명
get	• 접근자 프로퍼티를 통해 데이터 프로퍼티의 값을 읽을 때 호출되는 접근자 함수 • getter 함수가 호출되고 그 결과가 프로퍼티 값으로 반환
set	• 접근자 프로퍼티를 통해 데이터 프로퍼티의 값을 저장할 때 호출되는 접근자 함수 • setter 함수가 호출되고 그 결과가 프로퍼티 값으로 저장

접근자 프로퍼티

❖ getter 함수

- get 키워드로 선언

```
get display() {
    return `가로는 ${this.width}, 세로는 ${this.height}입니다.`;
}
```

- 호출 방법
  - 함수 호출 방식이 아닌 프로퍼티 호출 방식으로 호출

```
객체이름.display
```

접근자 프로퍼티

❖ setter 함수

- set 키워드로 선언

```
set changeWidth(value) {
    console.log(value);
}
```

- 값 저장 방법
  - 값 할당 방식으로 저장

```
객체이름.changeWidth = 100;
```

[실습] 접근자 프로퍼티 - 1

```
class Rectangle {
  constructor(w, h) {
    this.width = w;
    this.height = h;
  }

  area = () => console.log("사각형의 넓이를 구합니다.");

  get display() {
    return `가로는 ${this.width}, 세로는 ${this.height}입니다.`;
  }

  set changeWidth(value) {
    this.width = value;
  }
}
```

```
// 객체 생성
var rec1 = new Rectangle(100, 200);

// getter 함수 호출
console.log(rec1.display);

// setter 함수 호출
rec1.changeWidth = 300;

// 객체 호출
console.log(rec1);
```

[실습] 접근자 프로퍼티 - 1

❖ 실행 결과

```
가로는 100, 세로는 200입니다.
Rectangle { area: [Function: area], width: 300, height: 200 }
```

[실습] 접근자 프로퍼티 - 2

```
class Rectangle {
  #color = "red";

  constructor(w, h) {
     this.width = w;
     this.height = h;
  }

  area = () => console.log("사각형의 넓이를 구합니다.");

  get display() {
    return `가로는 ${this.width}, 세로는 ${this.height},
            색상은 ${this.#color} 입니다.`;
  }

  set changeColor(value) {
     this.#color = value;
  }
}
```

```
// 객체 생성
var rec1 = new Rectangle(100, 200);

// 객체 호출
console.log(rec1.display);

// setter 함수 호출
rec1.changeColor = "orange";

// 객체 호출
console.log(rec1.display);
```

[실습] 접근자 프로퍼티 - 2

❖ 실행 결과

```
가로는 100, 세로는 200, 색상은 red 입니다.
가로는 100, 세로는 200, 색상은 orange 입니다.
```

05

상속

상속

❖ 클래스 상속

- 한 클래스가 다른 클래스에서 정의된 속성 및 함수를 물려받아 그대로 사용

- 상속 받아 만들어진 클래스의 특징
  - **물려받은 기능**에 **필요한 기능을 추가**하여 정의할 수 있음

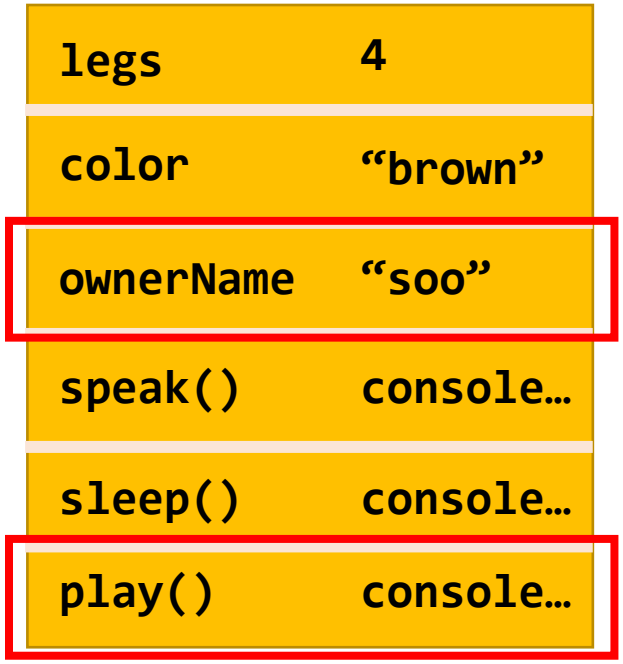
- 클래스 상속 방법

```
class 클래스이름 extends 부모클래스이름 {
    // 클래스 구현
}
```

상속

부모클래스 Animal

자식클래스 Dog



[실습] 클래스 상속

❖ 부모 클래스

```
class Animal {
    constructor(legs, color) {
        this.legs = legs;
        this.color = color;
    }

    speak() {
        console.log("소리낸다.");
    }

    sleep() {
        console.log("잔다.");
    }
}
```

[실습] 클래스 상속

❖ 자식 클래스

```
class Dog extends Animal {
    constructor(legs, color, ownerName) {
        super(legs, color);
        this.ownerName = ownerName;
    }

    play() {
        console.log("신나게 논다.");
    }

    sleep() {
        super.sleep();
        console.log("많이 잔다.");
    }
}
```

[실습] 클래스 상속

❖ 객체 생성 및 사용

```
const dog = new Dog(4, "brown", "soo");
dog.speak();
dog.play();
dog.sleep();
```

❖ 실행 결과

```
소리낸다.
신나게 논다.
잔다.
많이 잔다.
```

오버라이딩(overriding)

❖ 오버라이딩(overriding)

- 부모 클래스로부터 상속받은 메서드를 자식 클래스에서 재정의

```
sleep() {
          super.sleep();
          console.log("많이 잔다.");
}
```

- 실행 결과

```
잔다.          // 부모 클래스로부터 상속
많이 잔다.     // 자식 클래스에서 추가
```

오버라이딩(overriding)

❖ 오버라이딩(overriding)

- 부모 클래스로부터 상속받은 메서드를 자식 클래스에서 재정의

```
sleep() {
          super.sleep();
          console.log("많이 잔다.");
}
```

- 실행 결과

```
잔다.          // 부모 클래스로부터 상속
많이 잔다.     // 자식 클래스에서 추가
```

오버라이딩(overriding)

❖ 오버라이딩(overriding)

- 부모 클래스로부터 상속받은 메서드를 자식 클래스에서 재정의

```
sleep() {
          console.log("많이 잔다.");
}
```

- 실행 결과

```
많이 잔다.    // 자식 클래스에서 추가
```

instanceof 연산자

❖ instanceof 연산자

- 객체가 특정 클래스에 속하는지 아닌지를 확인해주는 연산자
  - 해당하면 true 리턴
  - 해당하지 않으면 false 리턴

```
console.log(dog instanceof Animal);  // true
console.log(cat instanceof Animal);  // false
```


THANK YOU