


혼자 공부하는 자바스크립트

Chapter 09 클래스





Contents

- CHAPTER 09: 클래스

SECTION 9-1 클래스의 기본 기능

SECTION 9-2 클래스의 고급 기능



CHAPTER 09 클래스

객체 지향을 이해하고 클래스의 개념과 문법 학습

SECTION 9-1 클래스의 기본 기능(1)

◦ 객체 지향 패러다임

- 깃허브 통계에 따르면, 자바스크립트, 자바, 파이썬, PHP, C#, C++, 루비, C, 오브젝티브C, 스칼라, 스위프트 등의 프로그래밍 언어가 많이 사용
- C를 제외한 모든 프로그래밍 언어는 객체 지향Object Oriented이라는 패러다임을 기반으로 만들어진 프로그래밍 언어
- 객체 지향 패러다임: 객체 지향 프로그래밍 객체를 만들고 객체들의 상호작용을 중심으로 개발하는 방법론
- 객체 지향 프로그래밍 언어들은 클래스(class)라는 문법으로 객체(object)를 효율적이고 안전하게 만들어 객체 지향 패러다임을 쉽게 프로그래밍에 적용할 수 있도록 도와줌

◦ 추상화(abstraction)

- 복잡한 자료, 모듈, 시스템 등으로부터 핵심적인 개념과 기능을 간추려내는 것
즉, 프로그램에 필요한 요소만 사용해서 객체를 표현하는 것

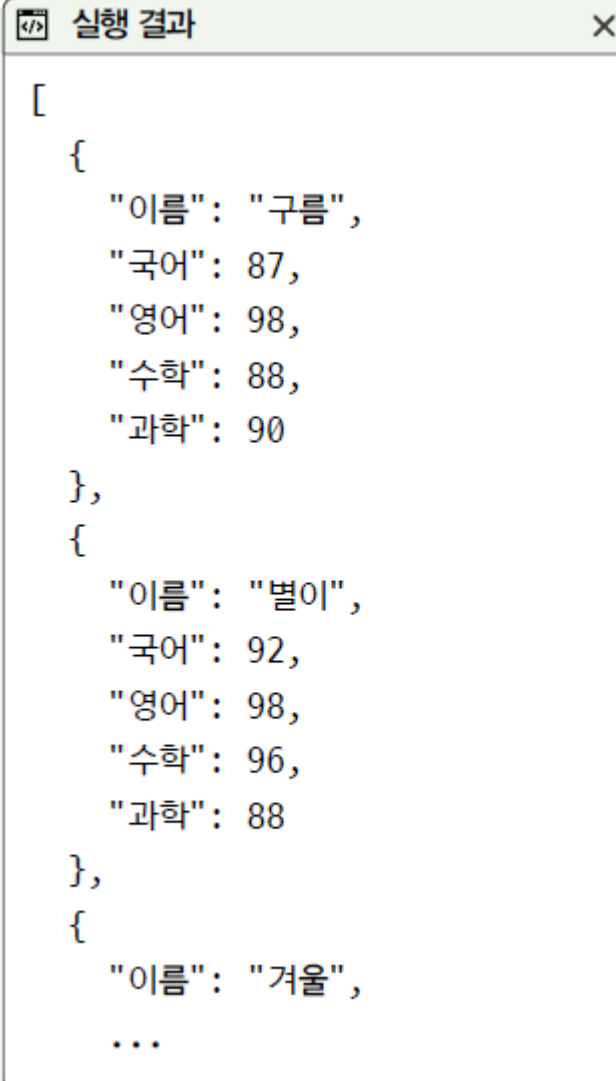
◦ 같은 형태의 개체 만들기

- 학생 성적 관리 프로그램
 - 학생이라는 객체가 필요하고, 학생들로부터 성적 관리에 필요한 공통사항을 추출하는데, 이를 추상화라고 함
 - 학생들이 여러 명이므로 추출한 요소는 배열을 이용해 관리

SECTION 9-1 클래스의 기본 기능(2)

- 같은 형태의 객체 만들기
 - 학생 성적 관리 프로그램
 - 객체와 배열 조합하기 (소스 코드 9-1-1.html)

```
01 <script>
02  // 객체를 선언합니다.
03  const students = []
04  students.push({ 이름: '구름', 국어: 87, 영어: 98, 수학: 88, 과학: 90 })
05  students.push({ 이름: '별이', 국어: 92, 영어: 98, 수학: 96, 과학: 88 })
06  students.push({ 이름: '겨울', 국어: 76, 영어: 96, 수학: 94, 과학: 86 })
07  students.push({ 이름: '바다', 국어: 98, 영어: 52, 수학: 98, 과학: 92 })
08
09  // 출력합니다.
10  alert(JSON.stringify(students, null, 2))
11 </script>
```



```
[
  {
    "이름": "구름",
    "국어": 87,
    "영어": 98,
    "수학": 88,
    "과학": 90
  },
  {
    "이름": "별이",
    "국어": 92,
    "영어": 98,
    "수학": 96,
    "과학": 88
  },
  {
    "이름": "겨울",
    ...
  }
]
```

SECTION 9-1 클래스의 기본 기능(3)

- 같은 형태의 객체 만들기
 - 각각의 객체에 학생들의 성적 총합과 평균을 구하는 기능을 추가
 - 객체 활용하기 (소스 코드 9-1-2.html)

```
01 <script>
02  // 객체를 선언합니다.
03  const students = [] ;
04  students.push({ 이름: '구름', 국어: 87, 영어: 98, 수학: 88, 과학: 90 })
05  students.push({ 이름: '별이', 국어: 92, 영어: 98, 수학: 96, 과학: 88 })
06  students.push({ 이름: '겨울', 국어: 76, 영어: 96, 수학: 94, 과학: 86 })
07  students.push({ 이름: '바다', 국어: 98, 영어: 52, 수학: 98, 과학: 92 })
08
09  // 출력합니다.
10  let output = '이름\t총점\t평균\n';
11  for (const s of students) {
12    const sum = s.국어 + s.영어 + s.수학 + s.과학;
13    const average = sum / 4;
14    output += `${s.이름}\t${sum}점\t${average}점\n`;
15  }
16  console.log(output) ;
17 </script>
```

실행 결과		
이름	총점	평균
구름	363점	90.75점
별이	374점	93.5점
겨울	352점	88점
바다	340점	85점

SECTION 9-1 클래스의 기본 기능(4)

- 객체를 처리하는 함수

- getSumOf()와 getAverageOf()라는 이름으로 함수를 만들고, 매개변수로 학생 객체를 받아 총합과 평균을 구하는 프로그램 만들기
- 객체를 처리하는 함수(1) (소스 코드 9-1-3.html)

```
01 <script>
02 // 객체를 선언합니다.
03 const students = []
04 students.push({ 이름: '구름', 국어: 87, 영어: 98, 수학: 88, 과학: 90 })
05 students.push({ 이름: '별이', 국어: 92, 영어: 98, 수학: 96, 과학: 88 })
06 students.push({ 이름: '겨울', 국어: 76, 영어: 96, 수학: 94, 과학: 86 })
07 students.push({ 이름: '바다', 국어: 98, 영어: 52, 수학: 98, 과학: 92 })
08
09 // 객체를 처리하는 함수를 선언합니다.
10 function getSumOf (student) {
11   return student.국어 + student.영어 + student.수학 + student.과학;
12 }
13
14 function getAverageOf (student) {
15   return getSumOf(student) / 4;
16 }
```

```
17
18 // 출력합니다.
19 let output = '이름\t총점\t평균\n';
20 for (const s of students) {
21   output += `${s.이름}\t${getSumOf(s)}점\t
22   ${getAverageOf(s)}점\n`;
22 }
23 console.log(output);
24 </script>
```

실행 결과는 9-1-2.html과 같음

▶ 다음 쪽에 코드 이어짐

SECTION 9-1 클래스의 기본 기능(5)

- 객체의 기능을 메소드로 추가하기

- 객체를 처리하는 함수(2) (소스 코드 9-1-4.html): getSum() 메소드와 getAverage() 메소드

```
01 <script>
02 // 객체를 선언합니다.
03 const students = []
04 students.push({ 이름: '구름', 국어: 87, 영어: 98, 수학: 88, 과학: 90 })
05 students.push({ 이름: '별이', 국어: 92, 영어: 98, 수학: 96, 과학: 88 })
06 students.push({ 이름: '겨울', 국어: 76, 영어: 96, 수학: 94, 과학: 86 })
07 students.push({ 이름: '바다', 국어: 98, 영어: 52, 수학: 98, 과학: 92 })
08
09 // students 배열 내부의 객체 모두에 메소드를 추가합니다.
10 for (const student of students) {
11   student.getSum = function () {
12     return this.국어 + this.영어 + this.수학 + this.과학
13   }
14
15   student.getAverage = function () {
16     return this.getSum() / 4
17   }
18 }
```

```
18 // 출력합니다.
19 let output = '이름\t총점\t평균\n';
20 for (const s of students) {
21   output += `${s.이름}\t${getSumOf(s)}점\t
22   ${getAverageOf(s)}점\n`;
22 }
23 console.log(output);
24 </script>
```

실행 결과는 9-1-2.html과 같음

SECTION 9-1 클래스의 기본 기능(6)

- 객체의 기능을 메소드로 추가하기
 - 객체를 생성하는 함수 (소스 코드 9-1-5.html)

```
<script>
function createStudent(이름, 국어, 영어, 수학, 과학) {
  return {
    // 속성을 선언합니다.
    이름: 이름,
    국어: 국어,
    영어: 영어,
    수학: 수학,
    과학: 과학,
    // 메소드를 선언합니다.
    getSum() {
      return this.국어 + this.영어 + this.수학 + this.과학;
    },
    getAverage() {
      return this.getSum() / 4;
    },
    toString() {
      return `${this.이름}\t${this.getSum()}점\t${this.getAverage()}점\n`;
    },
  };
}
```

```
}

// 객체를 선언합니다.
const students = [];
students.push(createStudent('구름', 87, 98, 88, 90));
students.push(createStudent('별이', 92, 98, 96, 88));
students.push(createStudent('겨울', 76, 96, 94, 86));
students.push(createStudent('바다', 98, 52, 98, 92));

// 출력합니다.
let output = '이름\t총점\t평균\n';
for (const s of students) {
  output += s.toString();
}
console.log(output);
</script>
```

SECTION 9-1 클래스의 기본 기능(7)

클래스 선언하기

- 클래스와 프로토타입
- 클래스의 형태

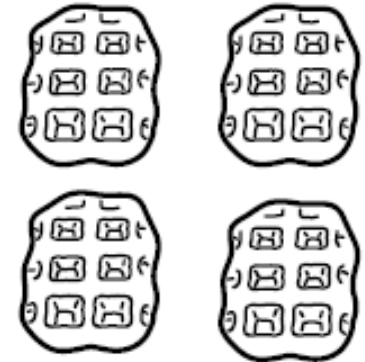
```
class 클래스 이름 {  
}
```

- 인스턴스: 클래스를 기반으로 만든 객체

```
new 클래스 이름()
```



붕어빵 틀
(클래스)



실체화된 붕어빵
(인스턴스)

SECTION 9-1 클래스의 기본 기능(8)

◦ 클래스 특징

- Strict mode : 기본적으로 엄격모드 실행
- constructor (생성자) : 객체를 생성하고 초기화하기 위한 특수 메소드
- 상속을 통한 클래스 확장
- super를 통한 상위 클래스 호출
- 메소드 오버라이딩
- private 속성, 메소드 지원
- static 키워드: 정적 메서드 정의

```
class 클래스 이름 {  
    constructor(x, y) { // 생성자  
        this.x = x;      // 멤버 변수(클래스 필드)  
        this.y = y;  
    }  
  
    // 메소드  
    calcArea() {  
    }  
}
```

SECTION 9-1 클래스의 기본 기능(9)

- 클래스 선언하기
 - 학생을 나타내는 Student 클래스를 만들고, 인스턴스를 생성하는 코드를 작성
 - 클래스 선언하고 인스턴스 생성하기 (소스 코드 9-1-6.html)

```
01 <script>
02 // 클래스를 선언합니다.
03 class Student {           → 클래스 이름은 첫 글자를 대문자로
04
05 }
06
07 // 학생을 선언합니다.
08 const student = new Student();
09
10 // 학생 리스트를 선언합니다.
11 const students = [
12   new Student(),
13   new Student(),
14   new Student(),
15   new Student()
16 ]
17 </script>
```

SECTION 9-1 클래스의 기본 기능(10)

- 생성자(constructor)
 - 생성자는 클래스를 기반으로 인스턴스를 생성할 때 처음 호출되는 메소드.
따라서 생성자에서는 속성을 추가하는 등 객체의 초기화 처리
 - 생성자 함수와 속성 추가하기 (소스 코드 9-1-7.html)

```
01 <script>
02 class Student {
03   constructor (이름, 국어, 영어, 수학, 과학) {
04     this.이름 = 이름;
05     this.국어 = 국어;
06     this.영어 = 영어;
07     this.수학 = 수학;
08     this.과학 = 과학;
09   }
10 }
11
```

```
12 // 객체를 선언합니다.
13 const students = [];
14 students.push(new Student('구름', 87, 98, 88, 90));
15 students.push(new Student('별이', 92, 98, 96, 88));
16 students.push(new Student('겨울', 76, 96, 94, 86));
17 students.push(new Student('바다', 98, 52, 98, 92))
18 </script>
```

SECTION 9-1 클래스의 기본 기능(11)

- 메소드(method)
 - 메소드 추가하기 (소스 코드 9-1-8.html)

```
01 <script>
02 class Student {
03   constructor (이름, 국어, 영어, 수학, 과학) {
04     this.이름 = 이름
05     this.국어 = 국어
06     this.영어 = 영어
07     this.수학 = 수학
08     this.과학 = 과학
09   }
10
11   getSum () {
12     return this.국어 + this.영어 + this.수학 + this.과학
13   }
14   getAverage () {
15     return this.getSum() / 4
16   }
17   toString () {
18     return `${this.이름}\t${this.getSum()}점
19       \t${this.getAverage()}점\n`
20   }
21 }
```

```
22 // 객체를 선언합니다.
23 const students = []
24 students.push(new Student('구름', 87, 98, 88, 90))
25 students.push(new Student('별이', 92, 98, 96, 88))
26 students.push(new Student('겨울', 76, 96, 94, 86))
27 students.push(new Student('바다', 98, 52, 98, 92))
28
29 // 출력합니다.
30 let output = '이름\t총점\t평균\n'
31 for (const s of students) {
32   output += s.toString()
33 }
34 console.log(output)
35 </script>
```

[요점 정리]

- 5가지 키워드로 정리하는 핵심 포인트
 - 객체 지향 패러다임은 객체를 우선적으로 생각해서 프로그램을 만든다는 방법론을 의미
 - **추상화**는 프로그램에서 필요한 요소만을 사용해서 객체를 표현하는 것을 의미
 - **클래스**는 객체를 안전하고 효율적으로 만들 수 있게 해주는 문법
 - **인스턴스**는 클래스를 기반으로 생성한 객체를 의미
 - **생성자**는 클래스를 기반으로 인스턴스를 생성할 때 처음 호출되는 메소드

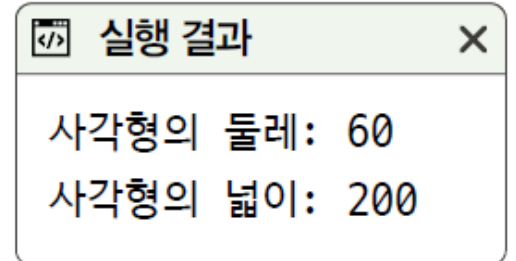
SECTION 9-2 클래스의 고급 기능(1)

◦ 상속

- Rectangle이라는 사각형을 나타내는 클래스를 선언하고 사용하는 예
- Rectangle 클래스 (소스 코드 9-2-1.html): getPerimeter()와 getArea()

```
01 <script>
02 class Rectangle {
03   constructor (width, height) {
04     this.width = width
05     this.height = height
06   }
07
08   // 사각형의 둘레를 구하는 메소드
09   getPerimeter () {
10     return 2 * (this.width + this.height)
11   }
12
13   // 사각형의 넓이를 구하는 메소드
14   getArea () {
15     return this.width * this.height
16   }
17 }
```

```
18
19 const rectangle = new Rectangle(10, 20)
20 console.log(`사각형의 둘레: ${rectangle.getPerimeter()}`)
21 console.log(`사각형의 넓이: ${rectangle.getArea()}`)
22 </script>
```



SECTION 9-2 클래스의 고급 기능(2)

- 상속(inheritance)

- 상속은 클래스의 선언 코드를 중복해서 작성하지 않도록 함으로써 코드의 생산 효율을 올리는 문법

```
class 클래스 이름 extends 부모클래스 이름 {  
}
```

- 사각형 클래스와 정사각형 클래스 (9-2-3.html)

```
01 <script>  
02 // 사각형 클래스  
03 class Rectangle {  
04   constructor (width, height) {  
05     this.width = width;  
06     this.height = height;  
07   }  
08  
09 // 사각형의 둘레를 구하는 메소드  
10 getPerimeter () {  
11   return 2 * (this.width + this.height);  
12 }  
13
```

```
14 // 사각형의 넓이를 구하는 메소드  
15 getArea () {  
16   return this.width * this.height;  
17 }  
18 }  
19  
20 // 정사각형 클래스  
21 class Square extends Rectangle {  
22   constructor (length) {  
23     super(length, length);  
24   }  
26 }  
27  
28 // 클래스 사용하기  
29 const square = new Square(10);  
30 console.log(`정사각형의 둘레: ${square.getPerimeter()}`);  
31 console.log(`정사각형의 넓이: ${square.getArea()}`);  
32 </script>
```

SECTION 9-2 클래스의 고급 기능(3)

- 상속(inheritance)
 - 상속 클래스에서 super를 호출하지 않을 경우

```
20 // 정사각형 클래스
21 class Square extends Rectangle {
22   constructor (length) {
23     // super(length, length);
24   }
26 }
27
28 // 클래스 사용하기
29 const square = new Square(10);
30 console.log(`정사각형의 둘레: ${square.getPerimeter()}`);
31 console.log(`정사각형의 넓이: ${square.getArea()}`);
32 </script>
```

Uncaught ReferenceError ReferenceError:
Must call super constructor in derived
class before accessing 'this' or
returning from derived constructor

- 일반 클래스가 new와 함께 실행되면, 빈 객체가 만들어지고 this에 이 객체를 할당
- 상속 클래스의 생성자 함수는 빈 객체를 만들고, this에 이 객체를 할당하는 일은 부모 클래스의 생성자가 처리

SECTION 9-2 클래스의 고급 기능(3)

- private 속성과 메소드
 - 사각형 클래스와 정사각형 클래스 (소스 코드 9-2-4.html)

```
01 <script>
02  // 정사각형 클래스
03  class Square {
04    constructor (length) {
05      this.length = length
06    }
07
08    getPerimeter () { return 4 * this.length }
09    getArea () { return this.length * this.length }
10  }
11
12  // 클래스 사용하기           └─ 길이에 음수를 넣어서 사용하고 있음
13  const square = new Square(-10)
14  console.log(`정사각형의 둘레: ${square.getPerimeter()}`)
15  console.log(`정사각형의 넓이: ${square.getArea()}`)
16 </script>
```

실행 결과

정사각형의 둘레: -40
정사각형의 넓이: 100

※ 현재 코드를 보면 Square 객체를 생성할 때 생성자의 매개변수로 음수를 전달하고 있음
그런데 '길이'라는 것은 음수가 나올 수 없는 값!

SECTION 9-2 클래스의 고급 기능(7)

- private 속성과 메소드

- 조건문을 활용해서 0 이하의 경우 예외를 발생시켜, 클래스의 사용자에게 불가함을 인지시킴
- 길이에 음수가 들어가지 않게 수정하기 (소스 코드 9-2-5.html)

```
01 <script>
02  // 정사각형 클래스
03  class Square {
04    constructor (length) {
05      if (length <= 0) {
06        throw '길이는 0보다 커야 합니다.' → throw 키워드를 사용해서 강제로 오류를 발생시킴
07      }
08      this.length = length
09    }
10
11    getPerimeter () { return 4 * this.length }
12    getArea () { return this.length * this.length }
13  }
14
15  // 클래스 사용하기
16  const square = new Square(-10)
17  console.log(`정사각형의 둘레: ${square.getPerimeter()}`)
18  console.log(`정사각형의 넓이: ${square.getArea()}`)
19 </script>
```

실행 결과

Uncaught 길이는 0보다 커야 합니다.

SECTION 9-2 클래스의 고급 기능(8)

- private 속성과 메소드

- 앞의 코드만으로는 다음과 같이 생성자로 객체를 생성한 이후에 사용자가 length 속성을 변경하는 것을 막을 수 없음

- 사용자의 잘못된 사용 예

```
// 클래스 사용하기
```

```
const square = new Square(10);
```

```
square.length = -10; → 이렇게 음수를 지정하는 것은 막을 수 없음
```

```
console.log(`정사각형의 둘레: ${square.getPerimeter()}`);
```

```
console.log(`정사각형의 넓이: ${square.getArea()}`);
```

- private 속성과 메소드: 클래스 사용자가 클래스 속성(또는 메소드)을 의도하지 않은 방향으로 사용하는 것을 막아 클래스의 안정성을 확보하기 위해 나온 문법

```
class 클래스 이름 {
```

```
  #속성 이름
```

```
  #메소드 이름 () {
```

```
  }
```

```
}
```

→ 속성과 메소드 이름 앞에 #을 붙임

SECTION 9-2 클래스의 고급 기능(9)

- private 속성과 메소드

- private 속성 사용하기(1) (소스 코드 9-2-6.html)

```
01 <script>
02  // 사각형 클래스
03  class Square {
04    #length;  → 이 위치에 해당 속성을 private 속성으로 사용하겠다고 미리 선언
05
06    constructor (length) {
07      if (length <= 0) {
08        throw '길이는 0보다 커야 합니다.';
09      }
10      this.#length = length;
11    }
12
13    getPerimeter () { return 4 * this.#length };
14    getArea () { return this.#length * this.#length };
15  }
16
17  // 클래스 사용하기
18  const square = new Square(10);
19  console.log(`정사각형의 둘레: ${square.getPerimeter()}`);
20  console.log(`정사각형의 넓이: ${square.getArea()}`);
21 </script>
```

실행 결과

정사각형의 둘레: 40
정사각형의 넓이: 100

SECTION 9-2 클래스의 고급 기능(10)

- private 속성과 메소드
 - private 속성 사용하기(2) (소스 코드 9-2-7.html)

```
01 <script>
02 // 사각형 클래스
03 class Square {
04   #length;
05
06   constructor (length) {
07     if (length <= 0) {
08       throw '길이는 0보다 커야 합니다.';
09     }
10     this.#length = length;
11   }
12
13   getPerimeter () { return 4 * this.#length }
14   getArea () { return this.#length * this.#length }
15 }
```

```
16
17 // 클래스 사용하기
18 const square = new Square(10);
19 square.length = -10; // 클래스 내부의 length 속성을 사용하여 변경
20 console.log(`정사각형의 둘레: ${square.getPerimeter()}`);
21 console.log(`정사각형의 넓이: ${square.getArea()}`);
22 console.log(square);
22 </script>
```

PROBLEMS	OUTPUT	DEBUG CONSOLE	TERMINAL
	정사각형의 둘레: 40		
	정사각형의 넓이: 100		
	> Square {#length: 10, length: -10}		

SECTION 9-2 클래스의 고급 기능(11)

- private 속성과 메소드
 - #length 속성을 사용하면 다음과 같은 오류를 발생
 - private 속성 사용하기(3) (소스 코드 9-2-8.html)

```
01 <script>
02  // 사각형 클래스
03  class Square {
04    #length;
05
06    constructor (length) {
07      if (length <= 0) {
08        throw '길이는 0보다 커야 합니다.';
09      }
10      this.#length = length;
11    }
12
13    getPerimeter () { return 4 * this.#length }
14    getArea () { return this.#length * this.#length }
15  }
```

```
16
17  // 클래스 사용하기
18  const square = new Square(10);
19  square.#length = -10;
20  console.log(`정사각형의 둘레: ${square.getPerimeter()}`);
21  console.log(`정사각형의 넓이: ${square.getArea()}`);
22  console.log(square);
22 </script>
```

실행 결과

⊗ Uncaught SyntaxError: Private field '#length' must be declared in an enclosing class

SECTION 9-2 클래스의 고급 기능(12)

- getter(획득자)와 setter(설정자)
 - 캡슐화를 통한 정보 은닉
 - 데이터 무결성

```
class Person {  
    constructor(name, age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    get name() {  
        return this._name.toUpperCase();  
    }  
  
    set name(newName) {  
        if (newName) {  
            this._name = newName;  
        }  
    }  
}
```

SECTION 9-2 클래스의 고급 기능(12)

- 게터와 세터

- 게터(getter)와 세터(setter) (메소드 소스 코드 9-2-9.html)

```
01 <script>
02 // 정사각형 클래스
03 class Square {
04   #length
05
06   constructor (length) {
07     this.setLength(length)
08   }
09
10   setLength (value) {
11     if (value <= 0) {
12       throw '길이는 0보다 커야 합니다.'
13     }
14     this.#length = value
15   }
```

함수를 사용하므로, 내부에서 예외 처리 등을 할 수 있음

```
17   getLength () {
18     return this.#length
19   }
20
21   getPerimeter () { return 4 * this.#length }
22   getArea () { return this.#length * this.#length }
23 }
24
25 // 클래스 사용하기
26 const square = new Square(10)
27 console.log(`한 변의 길이는 ${square.getLength()}입니다.`)
28
29 // 예외 발생시키기
30 square.setLength(-10)
31 </script>
```

실행 결과

한 변의 길이는 10입니다.
⊗ Uncaught 길이는 0보다 커야 합니다.

SECTION 9-2 클래스의 고급 기능(14)

- 게터와 세터

- get 키워드와 set 키워드 조합하기 (소스 코드 9-2-10.html)

```
01 <script>
02 // 정사각형 클래스
03 class Square {
04     #length;
05
06     constructor (length) {
07         this.length = length;
08     }
09         this.length에 값을 지정하면,
10         set length (length) 메소드 부분이 호출됨
11     get length () {
12         return this.#length;
13     }
14     get perimeter () {
15         return this.#length * 4;
16     }
17 }
```

```
18     get area () {
19         return this.#length * this.#length;
20     }
21
22     set length (length) {
23         if (length <= 0) {
24             throw '길이는 0보다 커야 합니다.';
25         }
26         this.#length = length;
27     }
28 }
29
30 // 클래스 사용하기
31 const squareA = new Square(10);
32 console.log(`한 변의 길이: ${squareA.length}`);
33 console.log(`둘레: ${squareA.perimeter}`);
34 console.log(`넓이: ${squareA.area}`);
35
36 // 예외 발생시키기
37 const squareB = new Square(-10);
38 </script>
```

SECTION 9-2 클래스의 고급 기능(16)

- static 속성과 메소드

- static 키워드 사용하기 (소스 코드 9-2-11.html)

<pre>01 <script> 02 class Square { 03 #length; 04 static #counter = 0; 05 static get counter () { 06 return Square.#counter; 07 } 08 09 constructor (length) { 10 this.length = length 11 Square.#counter += 1; 12 } 13 14 static perimeterOf (length) { 15 return length * 4; 16 } 17 static areaOf (length) { 18 return length * length; 19 }</pre>	<div style="display: flex; align-items: center; margin-bottom: 10px;"><div style="flex: 1; border-bottom: 1px solid black; margin-right: 10px;"></div><div style="color: red; font-size: 0.8em;">private 특성과 static 특성은 한꺼번에 적용할 수도 있음</div></div> <pre>21 get length () { return this.#length } 22 get perimeter () { return this.#length * 4 } 23 get area () { return this.#length * this.#length } 24 25 set length (length) { 26 this.#length = length; 27 } 28 } </pre> <div style="margin-top: 20px;"><pre>33 // static 속성 사용하기 34 const squareA = new Square(10); 35 const squareB = new Square(20); 36 console.log(`생성된 Square 인스턴스는 \${Square.counter}개`); 37 38 // static 메소드 사용하기 39 console.log(`한 변이 20인 정사각형의 둘레는 \${Square.perimeterOf(20)}`); 40 console.log(`한 변이 30인 정사각형의 둘레는 \${Square.areaOf(30)}`); 41 </script></pre></div>
---	---

[좀 더 알아보기①] 오버라이딩

- 오버라이딩(overriding) – 상속받은 부모의 메소드를 재정의하는 것
- LifeCycle이라는 간단한 클래스를 선언하고 사용하기
 - LifeCycle 클래스에는 a(), b(), c()라는 이름의 메소드가 있고, call()이라는 이름의 메소드에서 이를 호출
- 메소드에서 순서대로 메소드 호출하기 (소스 코드 9-2-12.html)

```
01 <script>
02  // 클래스를 선언합니다.
03  class LifeCycle {
04    call () {
05      this.a();
06      this.b();
07      this.c();
08    }
09
10    a () { console.log('a() 메소드를 호출합니다.');
```

```
11    b () { console.log('b() 메소드를 호출합니다.');
```

```
12    c () { console.log('c() 메소드를 호출합니다.');
```

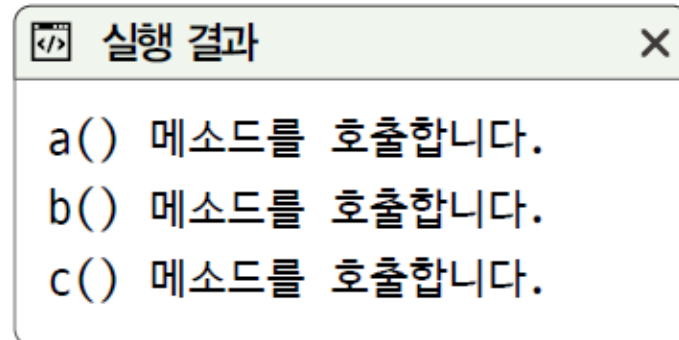
```
13  }
```

```
14
```

```
15  // 인스턴스를 생성합니다.
```

```
16  new LifeCycle().call();
```

```
17 </script>
```



[좀 더 알아보기②] 오버라이딩

- LifeCycle 클래스를 상속받는 Child라는 이름의 클래스를 선언하고 내부에서 부모에 있던 a()라는 이름의 메소드 만들기
- 메소드 오버라이딩 (소스 코드 9-2-13.html)

```
01 <script>
02  // 클래스를 선언합니다.
03  class LifeCycle {
04    call () {
05      this.a();
06      this.b();
07      this.c();
08    }
09
10    a () { console.log('a() 메소드를 호출합니다.');" }
11    b () { console.log('b() 메소드를 호출합니다.');" }
12    c () { console.log('c() 메소드를 호출합니다.');" }
13  }
14
```

```
15 class Child extends LifeCycle {
16   a () {
17     console.log('자식의 a() 메소드입니다.');" }
18   }
19 }
20
21  // 인스턴스를 생성합니다.
22  new Child().call();
23 </script>
```

오버라이딩

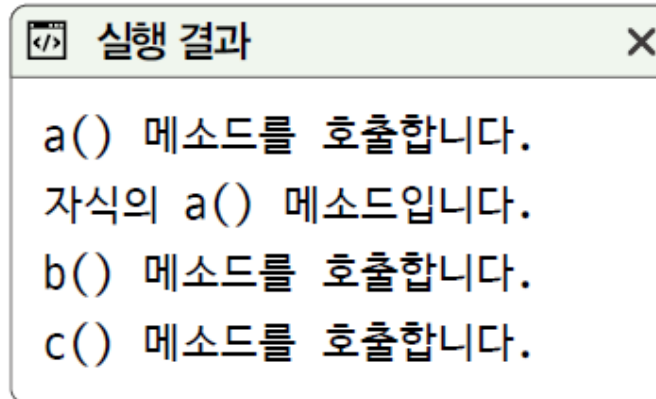
실행 결과

자식의 a() 메소드입니다.
b() 메소드를 호출합니다.
c() 메소드를 호출합니다.

[좀 더 알아보기③] 오버라이딩

- 부모에 있던 메소드의 내용도 사용하고 싶다면 다음과 같이 super.메소드() 형태의 코드를 사용
 - super.a()는 부모의 a() 메소드를 실행하는 코드
- 부모에 있던 내용 가져오기 (소스 코드 9-2-14.html)

```
01 <script>
02  // 클래스를 선언합니다.
03  class Lifecycle { /* 생략 */ }
04  class Child extends Lifecycle {
05    a () {
06      super.a();
07      console.log('자식의 a() 메소드입니다.');
```





[좀 더 알아보기④] 오버라이딩

- 자바스크립트는 내부적으로 어떤 객체를 문자열로 만들 때 toString() 메소드를 호출
 - 따라서 toString() 메소드를 오버라이딩하면 내부적으로 문자열로 변환되는 형태를 바꿀 수 있음
- toString() 메소드 오버라이드하기 (소스 코드 9-2-15.html)

```
01 <script>
02 class Pet {
03   constructor (name, age) {
04     this.name = name;
05     this.age = age;
06   }
07
08   toString () {
09     return `이름: ${this.name}\n나이: ${this.age}살`;
10   }
11 }
12
13 const pet = new Pet('구름', 6);
14 alert(pet);
15 console.log(pet + "");
16 </script>
```

→ 오버라이딩

 실행 결과 

[경고창 출력]

이름: 구름
나이: 6

[콘솔 출력]

이름: 구름
나이: 6

[요점 정리]

- 5가지 키워드로 정리하는 핵심 포인트
 - 상속은 어떤 클래스가 갖고 있는 유산(속성과 메소드)을 기반으로 새로운 클래스를 만드는 것
 - **private** 속성/메소드는 클래스 내부에서만 접근할 수 있는 속성/메소드
 - **게터**는 get○○() 형태로 값을 확인하는 기능을 가진 메소드를 의미
 - **세터**는 set○○() 형태로 값을 지정하는 기능을 가진 메소드를 의미
 - **오버라이드**는 부모가 갖고 있는 메소드와 같은 이름으로 메소드를 선언해서 덮어 쓰는 것을 의미

SECTION 9-3 정규표현식

- 정규 표현식(regular expression)이란?

- 문자열에서 특정한 규칙을 가지는 문자열의 집합을 찾아내기 위한 검색 패턴

- 정규 표현식 생성 방식

- 정규 표현식 리터럴을 이용한 생성
- RegExp 객체를 이용한 생성

```
let regStr = /a+bc/;      // 정규 표현식 리터럴을 이용한 생성
let regObj = new RegExp('a+bc'); // RegExp 객체를 이용한 생성
```

```
regStr;           // /a+bc/
regObj;           // /a+bc/
```

- 정규 표현식 문법

/검색패턴/플래그

```
let targetStr = 'bcabcAB';
```

```
let strReg = /AB/;           // 검색 패턴 비교 시 대소문자를 구분함.
let strUsingFlag = /AB/i;    // new RegExp('AB', 'i')와 동일함.
```

```
targetStr.search(strReg);    // 5
targetStr.search(strUsingFlag); // 2 -> 대소문자를 구분하지 않고 검색함.
```

SECTION 9-3 정규표현식

- 정규표현식 매칭 패턴(문자, 숫자, 기호 등)
 - 정규 표현식을 생성할 때 플래그를 사용하여 기본 검색 설정을 변경

패턴	의미
a-zA-Z	영어알파벳(-으로 범위 지정)
ㄱ-ㅎ가-힣	한글 문자(-으로 범위 지정)
0-9	숫자(-으로 범위 지정)
.	모든 문자열(숫자, 한글, 영어, 특수기호, 공백 모두! 단, 줄바꿈x)
\d	숫자
\D	숫자가 아닌 것
\w	영어 알파벳, 숫자, 언더스코어(_)
\W	/w 가 아닌 것
\s	space 공백
\S	space 공백이 아닌 것
\특수기호	특수기호

SECTION 9-3 정규표현식

- 정규표현식 검색 패턴
 - 검색 패턴을 이용하면 AND, OR, StartWith, EndWith 등의 다양한 조합을 만들 수 있음

기호	의미
	OR
[]	괄호안의 문자들 중 하나
[^문자]	괄호안의 문자를 제외한 것
^문자열	특정 문자열로 시작(괄호 없음 주의!)
문자열\$	특정 문자열로 끝남
()	그룹 검색 및 분류(match메서드에서 그룹별로 묶어줌)
(?: 패턴)	그룹 검색(분류x)
\b	단어의 처음/끝
\B	단어의 처음/끝이 아님

SECTION 9-3 정규표현식

- 정규표현식 개수(수량) 패턴
 - 특정 패턴이 몇 번 반복되는지 필터링

기호	의미
?	최대 한번(없음 한개)
*	없거나 있거나 (없음 있음): 여러개 포함
+	최소 한개(한개 여러개)
{n}	n개
{Min,}	최소 Min개 이상
{Min, Max}	최소 Min개 이상, 최대 Max개 이하

- 정규표현식 예제

```
const str = '유선전화: 02-900-1234, 핸드폰: 010-2000-7777';  
console.log(str.match(/\d{2,3}-\d{3,4}-\d{4}/g)); // ['02-900-1234', '010-2000-7777']
```

```
const text = '이메일: test1@gmail.com';  
console.log(text.match(/[w\-\.\+]\@[w\-\.\+]/g)); // [ 'test1@gmail.com' ]
```

SECTION 9-3 정규표현식

- 플래그(flag)
 - 정규 표현식을 생성할 때 플래그를 사용하여 기본 검색 설정을 변경

플래그	설명
i	대소문자를 구분없이 검색
g	패턴과 일치하는 모든 것을 검색. (기본값은 첫 번째 결과만 반환)
m	여러 줄 일치(multi line)

- 이스케이프 문자(Escape Character)
 - 정규표현식 내에서 특정한 기능으로 활용되는 기호를 문자로 해석하도록 해 줌

```
const str = 'test1@gmail.com';
```

```
console.log(str.match(/\./gi)); // .(dot)은 정규식 내에서 특정 기능으로 활용되므로 앞에 \ (백슬래시)를 추가하여 검색함
```

SECTION 9-3 정규표현식

◦ str.match 검색

- str.match(regex) : str 문자열에서 regex와 일치하는 것을 검색
- 플래그 g가 포함될 경우 패턴과 일치하는 모든 값을 담은 배열을 반환

```
let str = 'We will, we will rock you';
```

```
console.log(str.match(/we/gi)); // We,we (패턴과 일치하는 부분 문자열 두 개를 담은 배열)
```

```
console.log(str.match(/we/i)); // We (패턴과 일치하는 부분 문자열 한 개를 담은 배열)
```

```
console.log(str.match(/my/i)); // null (패턴과 일치하는 부분 문자열이 없을 경우)
```

◦ str.replace 치환

- str.replace(regex, replacement): str 문자열에서 regex와 일치하는 것을 replacement로 치환

```
// 플래그 g 없음
```

```
console.log('We will, we will'.replace(/we/i, 'I')); // I will, we will
```

```
// 플래그 g 있음
```

```
console.log('We will, we will'.replace(/we/gi, 'I')); // I will, I will
```

SECTION 9-3 정규표현식

- regexp.test 일치 여부 확인
 - regexp.test(str) : str 문자열에서 regexp와 일치하는 부분 문자열이 하나라도 있는지 확인.
 - 일치하는 문자열이 있으면 true, 없으면 false 반환

```
str = 'I love JavaScript';
```

```
let regexp = /LOVE/i;
```

```
console.log(regexp.test(str)); // true
```

SECTION 9-3 정규표현식 – RegExp 객체

- RegExp 객체

- 정규 표현식을 구현한 자바스크립트 표준 내장 객체
- 문법

```
new RegExp(검색패턴[, 플래그]);
```

- RegExp - exec() 메소드

- 인수로 전달된 문자열에서 특정 패턴을 검색하여, 패턴과 일치하는 문자열을 반환
- 일치하는 문자열이 없으면 null 반환

```
const str = 'I like football.';
```

```
const regex1 = RegExp('foo*', 'g');
```

```
console.log(regex1.exec(str)); // ['foo', index: 7, input: 'I like football.', groups:  
undefined]
```

SECTION 9-3 정규표현식 – RegExp 객체

- RegExp – test() 메소드

- 인수로 전달된 문자열에서 특정 패턴을 검색하여 일치하는 문자열이 있는지 확인
- 일치하는 문자열이 있으면 true, 없으면 false 반환

```
const str = 'table football';  
const regex = new RegExp('foo*', 'g');  
console.log(regex.test(str)); // true
```
