

ES5시절의 OOP

Objects and Prototype



Objects and Prototype



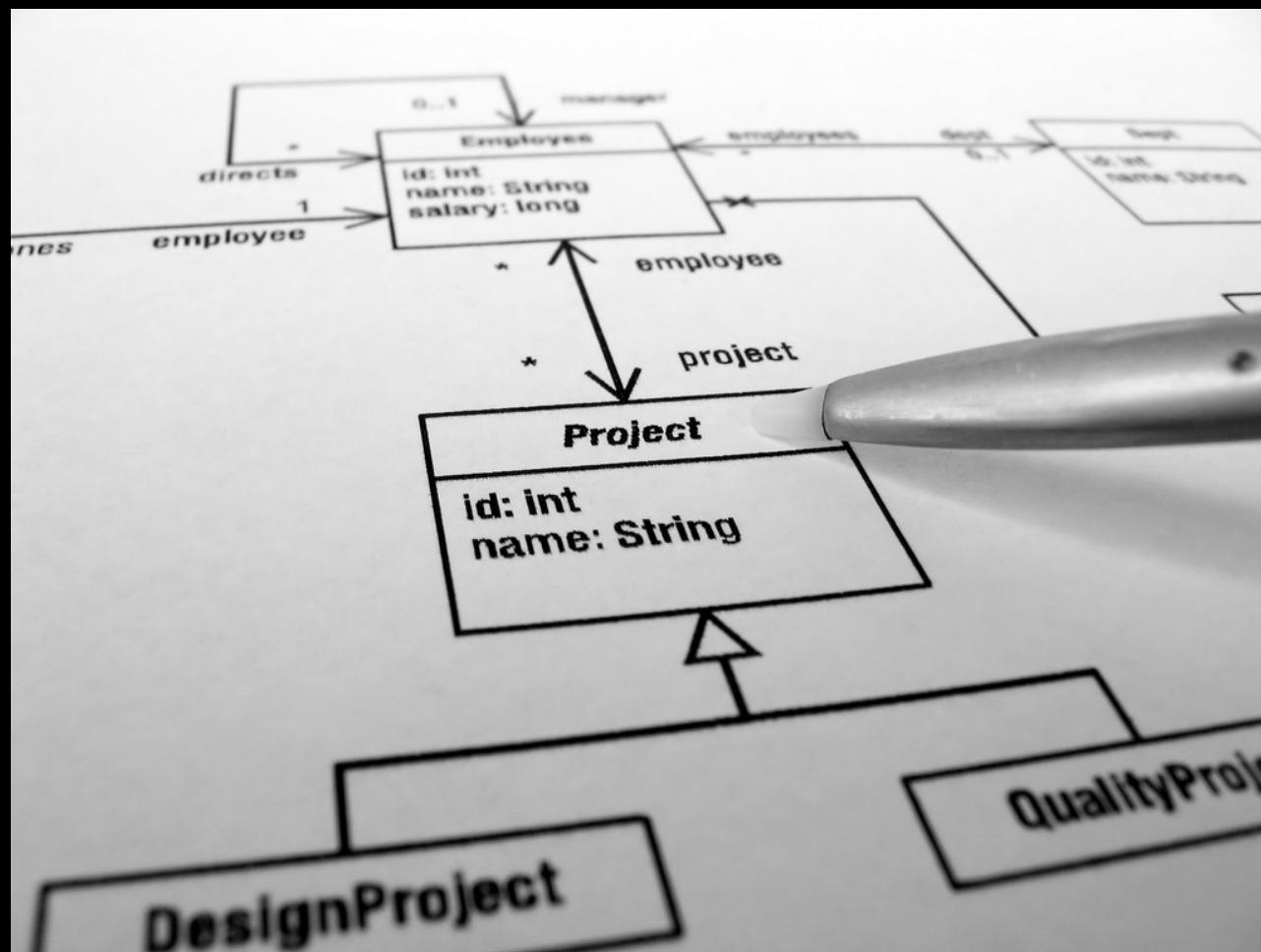
ES6시대의 OOP

class 키워드와 관계

진짜! 모두를 위한
자바스크립트

ES6시대의 OOP

class 키워드와 관계

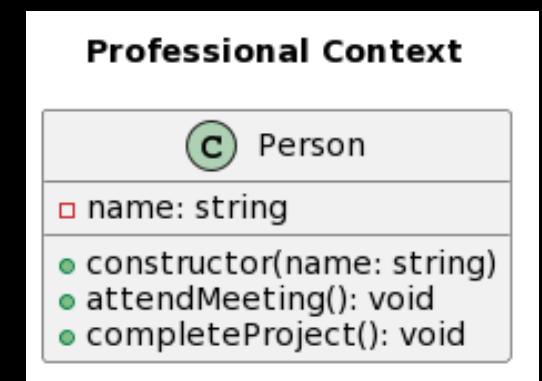
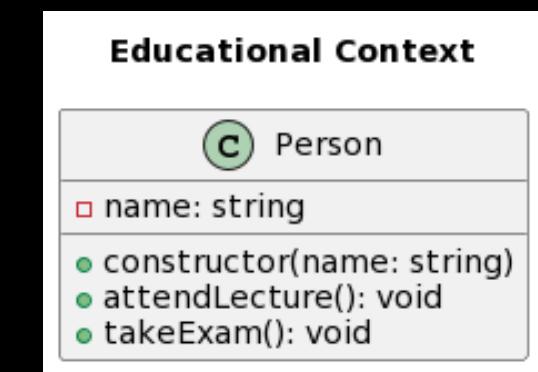
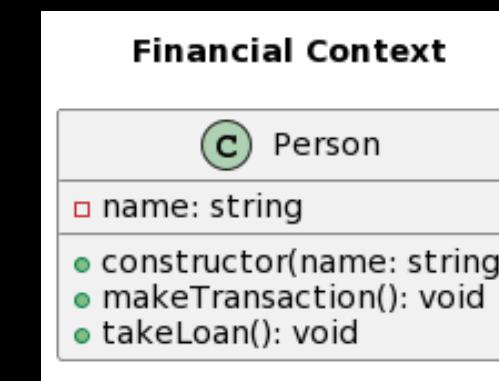
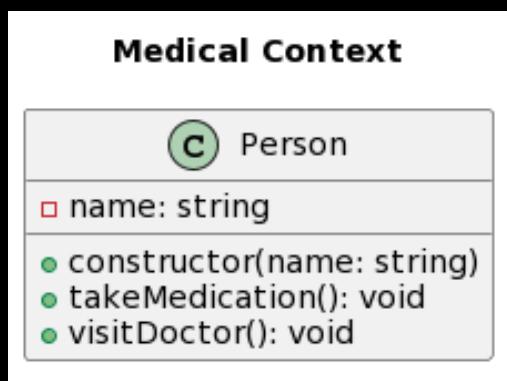


ES6시대의 OOP

class 키워드와 관계

진짜!
모두를 위한
자바스크립트

Abstraction



ES6시대의 OOP

class 키워드와 관계

진짜!
모두를 위한
자바스크립트

Abstraction

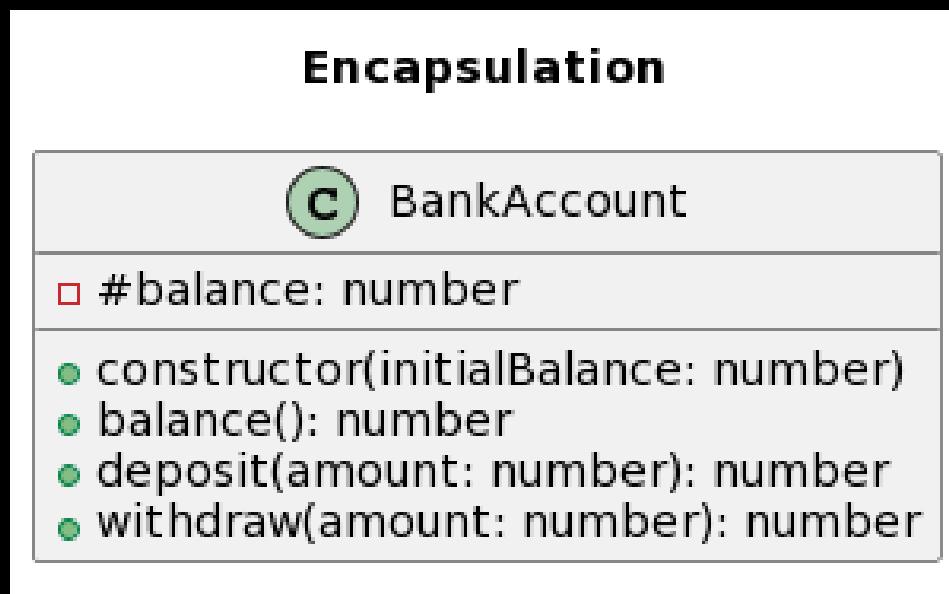


ES6시대의 OOP

class 키워드와 관계

진짜! 모두를 위한
자바스크립트

Encapsulation

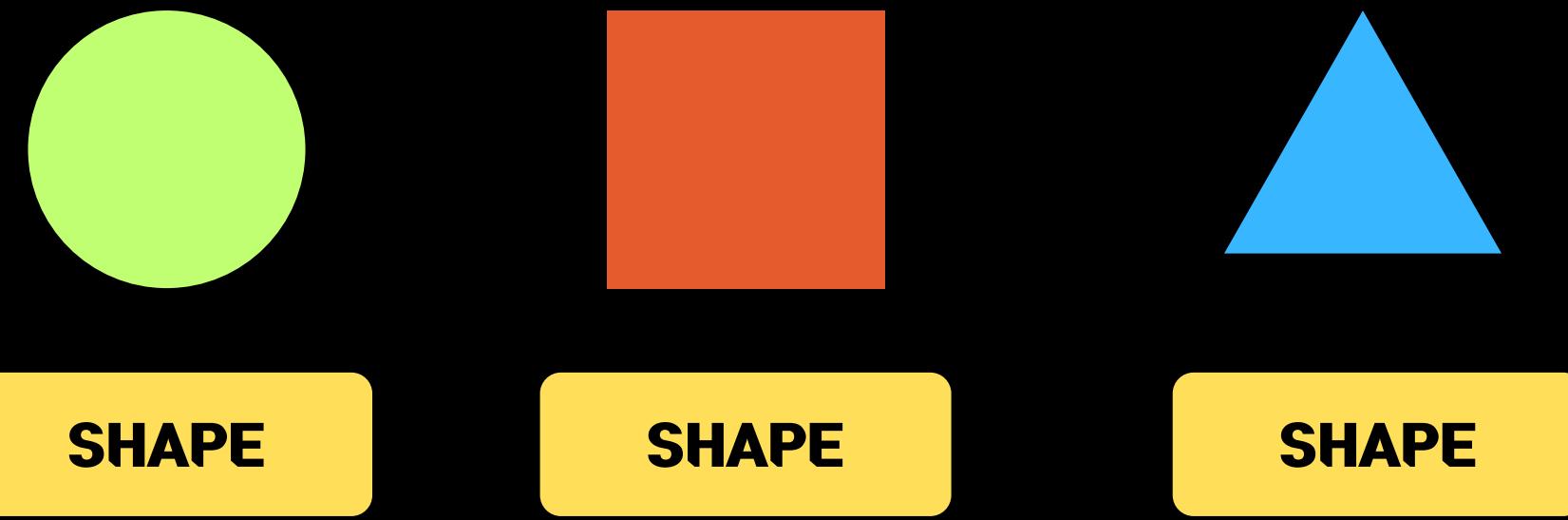


ES6시대의 OOP

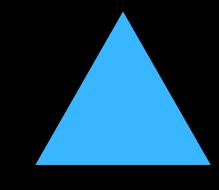
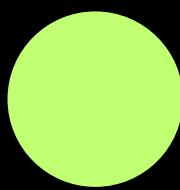
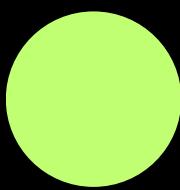
class 키워드와 관계

진짜! 모두를 위한
자바스크립트

Polymorphism



Polymorphism과 느슨한 결합



SHAPE

...

Polymorphism

```
function drawShape(circle: Circle) {  
    circle.draw();  
}
```

...

Polymorphism

```
function drawShape(shape: Shape) {  
    shape.draw();  
}
```

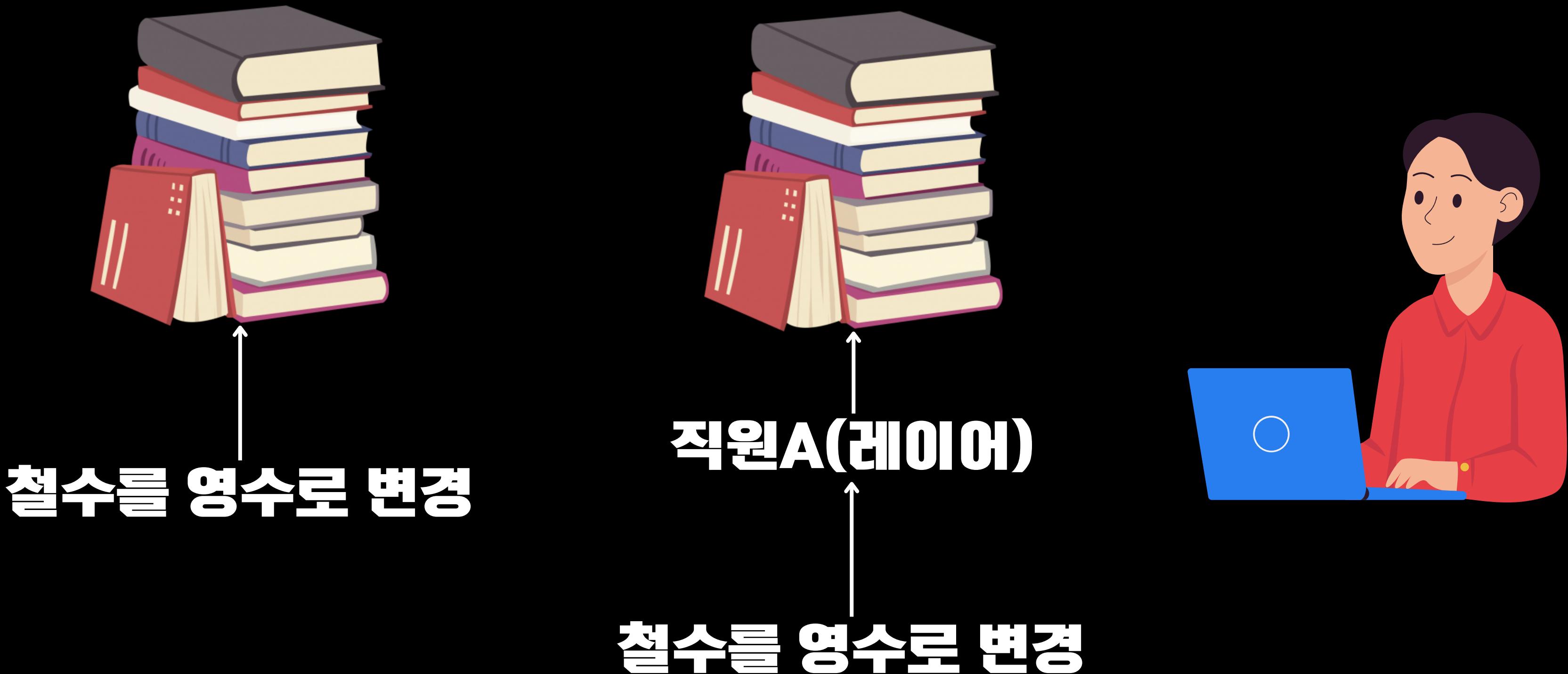


ES6시대의 OOP

class 키워드와 관계

진짜! 모두를 위한
자바스크립트

Polymorphism과 느슨한 결합

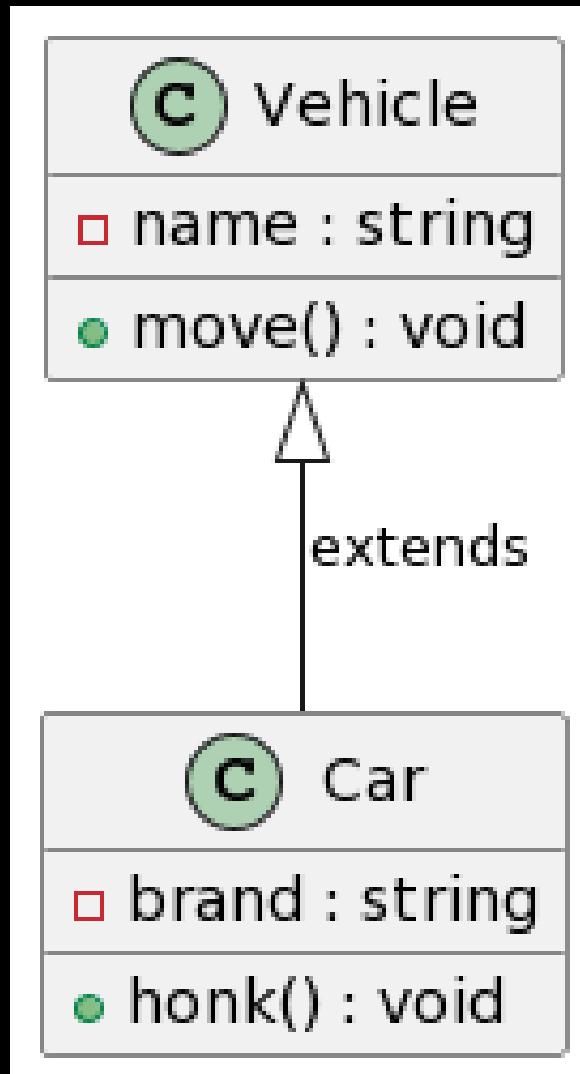


ES6시대의 OOP

class 키워드와 관계

진짜! 모두를 위한
자바스크립트

IS-A(Inheritance) 관계



IS-A RELATIONSHIP

```
class Vehicle {  
    constructor(name) {  
        this.name = name;  
    }  
  
    move() {  
        console.log(`${this.name} is moving.`);  
    }  
}
```

IS-A RELATIONSHIP

```
class Car extends Vehicle {  
    constructor(name, brand) {  
        super(name);  
        this.brand = brand;  
    }  
  
    honk() {  
        console.log(`${this.brand} car is honking.`);  
    }  
}
```

IS-A RELATIONSHIP

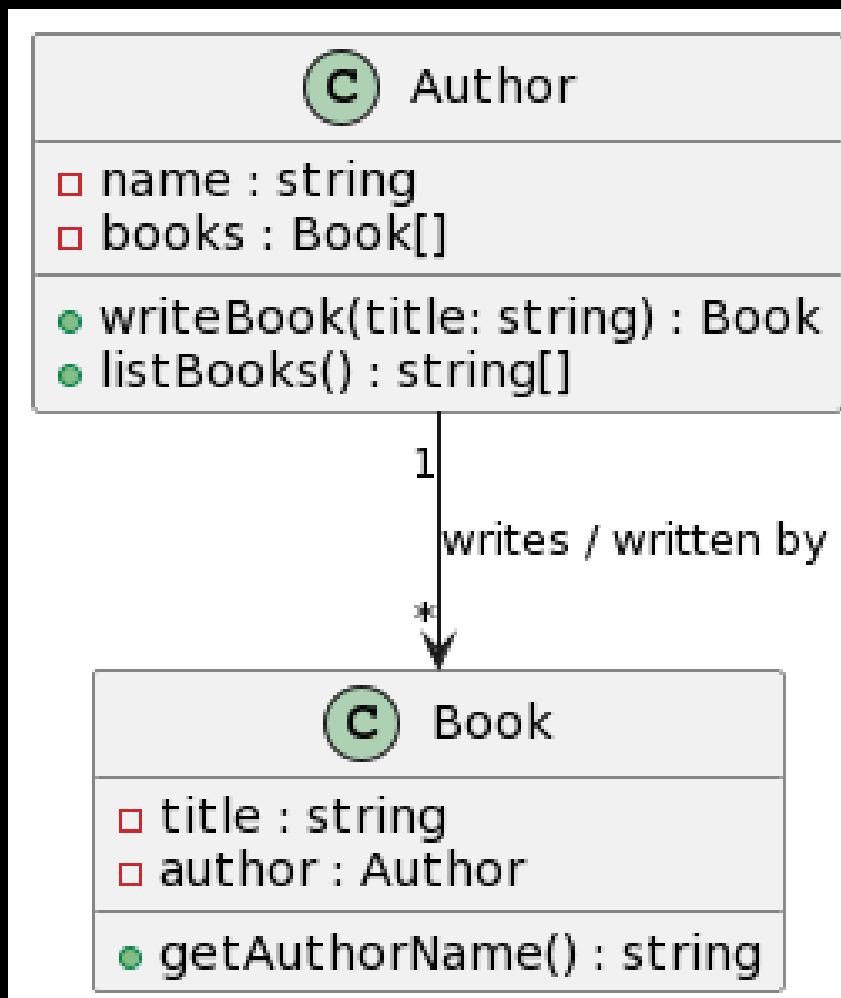
```
const myCar = new Car('K5', 'KIA');  
myCar.move();  
myCar.honk();  
  
console.log(myCar instanceof Car);  
console.log(myCar instanceof Vehicle);
```

ES6시대의 OOP

class 키워드와 관계

진짜! 모두를 위한
자바스크립트

Association 관계



Association RELATIONSHIP

```
class Author {
    constructor(name) {
        this.name = name;
        // Author와 연관된 Book 인스턴스 참조를 담는다.
        this.books = [];
    }

    writeBook(title) {
        const book = new Book(title, this);
        this.books.push(book);
        return book;
    }

    listBooks() {
        return this.books.map(book => book.title);
    }
}
```

Association RELATIONSHIP

```
class Book {
    constructor(title, author) {
        this.title = title;
        // Book 은 작가와 연관 관계에 있다.
        this.author = author;
    }

    getAuthorName() {
        return this.author.name;
    }
}
```

Association RELATIONSHIP

```
const author = new Author('신경림');
const book = author.writeBook('토지');

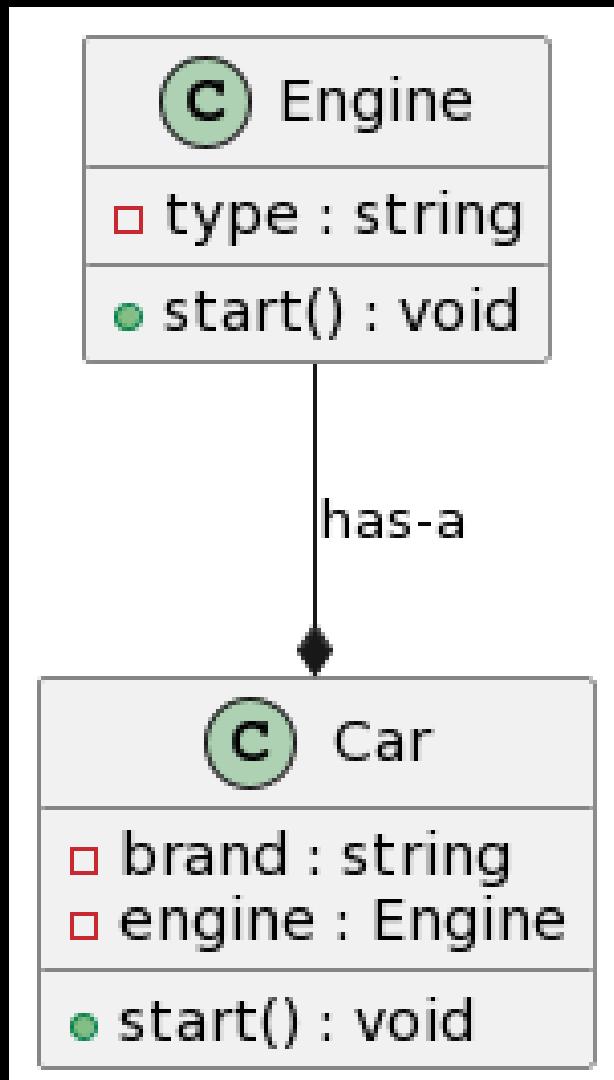
console.log(book.getAuthorName());
console.log(author.listBooks());
```

ES6시대의 OOP

class 키워드와 관계

진짜! 모두를 위한
자바스크립트

HAS-A(Compositon) 관계



HAS-A RELATIONSHIP

```
class Engine {
    constructor(type) {
        this.type = type;
    }

    start() {
        console.log(`The ${this.type} engine is starting.`);
    }
}
```

HAS-A RELATIONSHIP

```
class Car {
    constructor(brand, engineType) {
        this.brand = brand;
        // Car 객체는 Engine 객체를 속성으로 갖는다.
        this.engine = new Engine(engineType);
    }

    start() {
        console.log(`The ${this.brand} car is starting.`);
        this.engine.start();
    }
}
```

HAS-A RELATIONSHIP

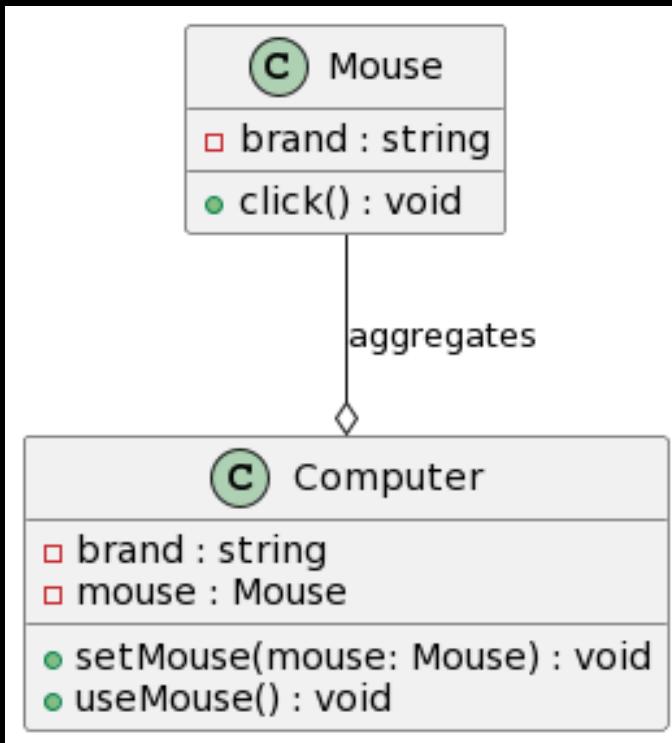
```
const myCar = new Car('KIA', '1.6L a 1591 cc, 4 cylinder Petrol');
myCar.start();
```

ES6시대의 OOP

class 키워드와 관계

진짜! 모두를 위한
자바스크립트

Aggregation 관계



● ● ● Aggregation RELATIONSHIP

```
class Mouse {
    constructor(brand) {
        this.brand = brand;
    }

    click() {
        console.log('마우스 클릭');
    }
}
```

● ● ● Aggregation RELATIONSHIP

```
class Computer {
    constructor(brand, mouse = null) {
        this.brand = brand;
        this.mouse = mouse;
    }

    setMouse(mouse) {
        this.mouse = mouse;
    }

    useMouse() {
        if (this.mouse) {
            this.mouse.click();
        } else {
            console.log('마우스를 연결해 주세요.');
        }
    }
}
```

● ● ● Aggregation RELATIONSHIP

```
const mouse = new Mouse('Logitech');

// 마우스는 컴퓨터 없이도 독립적으로 사용 가능
mouse.click();

// 컴퓨터도 마우스 없이 독립적으로 사용 가능
const computer = new Computer('iMac');
// 단, 마우스를 연결해 주세요. 메시지 출력
computer.useMouse();

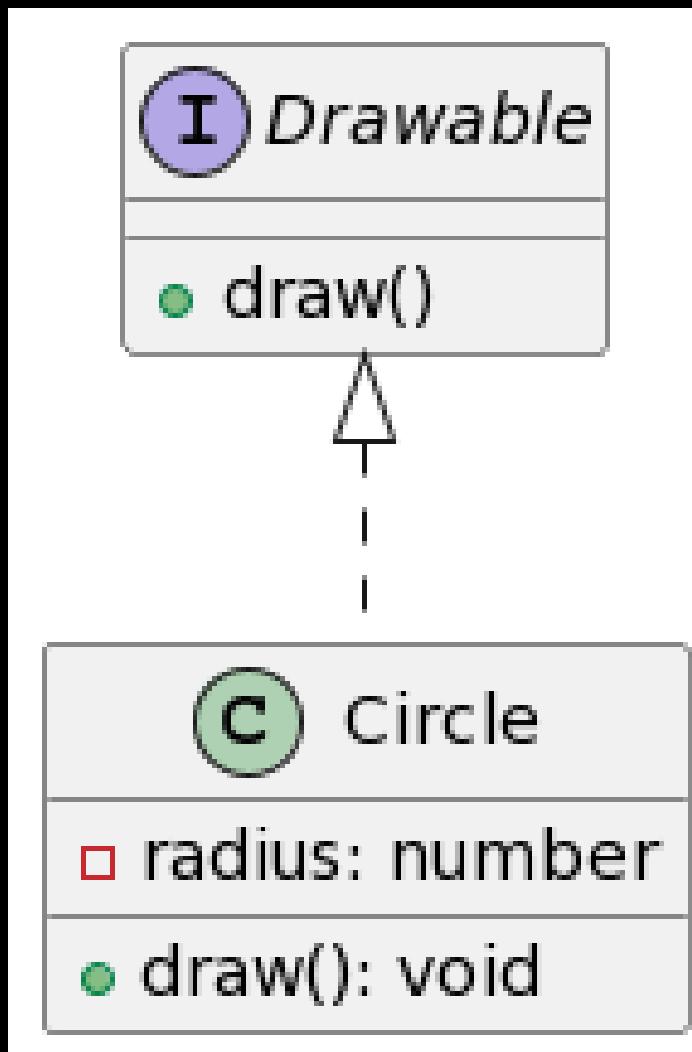
// 마우스를 컴퓨터에 연결
computer.setMouse(mouse);
// 마우스 클릭
computer.useMouse();
```

ES6시대의 OOP

class 키워드와 관계

진짜! 모두를 위한
자바스크립트

Realization/Implementation 관계



Realization Relationship in Typescript

```
interface Drawable {
    draw();
}

class Circle implements Drawable {
    constructor(private radius: number) {}

    draw() {
        console.log(`반지름이 ${this.radius}인 원을 그립니다.`);
    }
}
```

Realization Relationship in Typescript

```
function drawShape(drawable: Drawable) {
    if (!(drawable instanceof Drawable)) {
        throw new Error("Drawable 인터페이스를 구현한 객체가 아닙니다.");
    }
    drawable.draw();
}

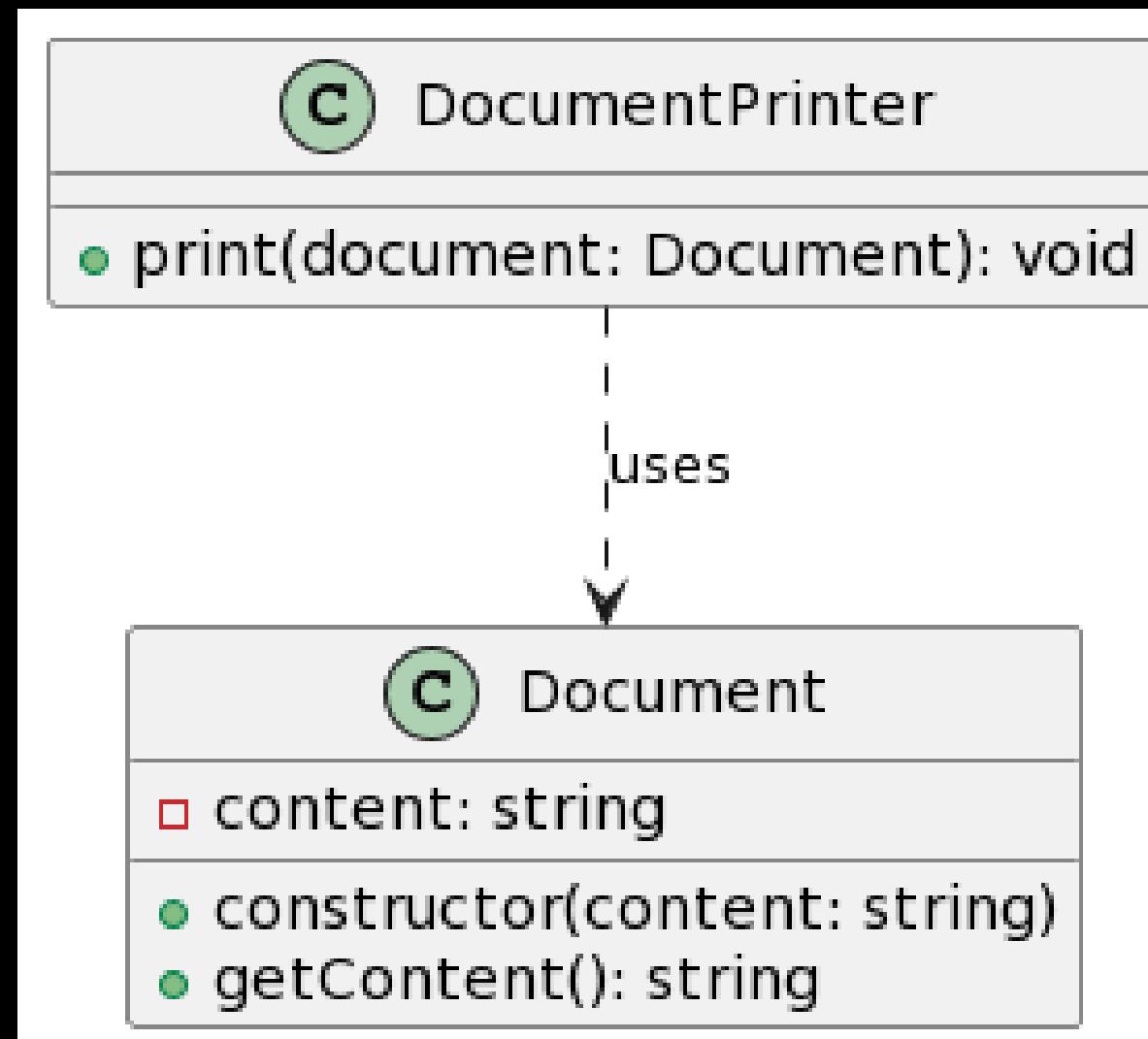
const circle = new Circle(5);
drawShape(circle);
```

ES6시대의 OOP

class 키워드와 관계

진짜! 모두를 위한
자바스크립트

Dependency(using) 관계



Dependency Relationship

```
class DocumentPrinter {
    print(document) {
        if (!(document instanceof Document)) {
            throw new Error('Document 를 제공해 주세요.');
        }
        console.log(`문서를 인쇄합니다.: ${document.getContent()}`);
    }
}
```

Dependency Relationship

```
class Document {
    constructor(content) {
        this.content = content;
    }

    getContent() {
        return this.content;
    }
}
```

Dependency Relationship

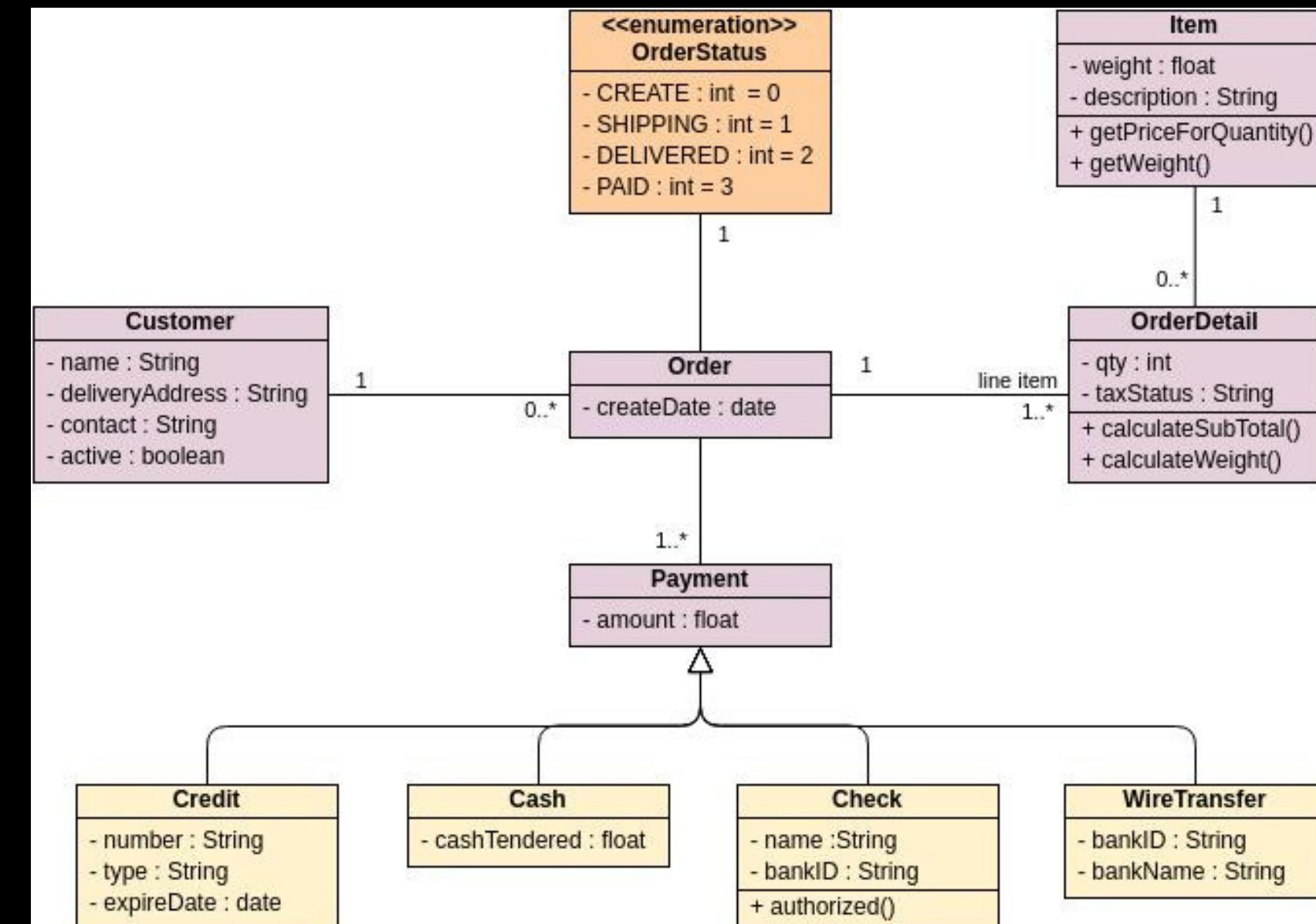
```
const doc = new Document('어느 멋진 날이었다.');
const printer = new DocumentPrinter();
printer.print(doc);
```

ES6시대의 OOP

class 키워드와 관계

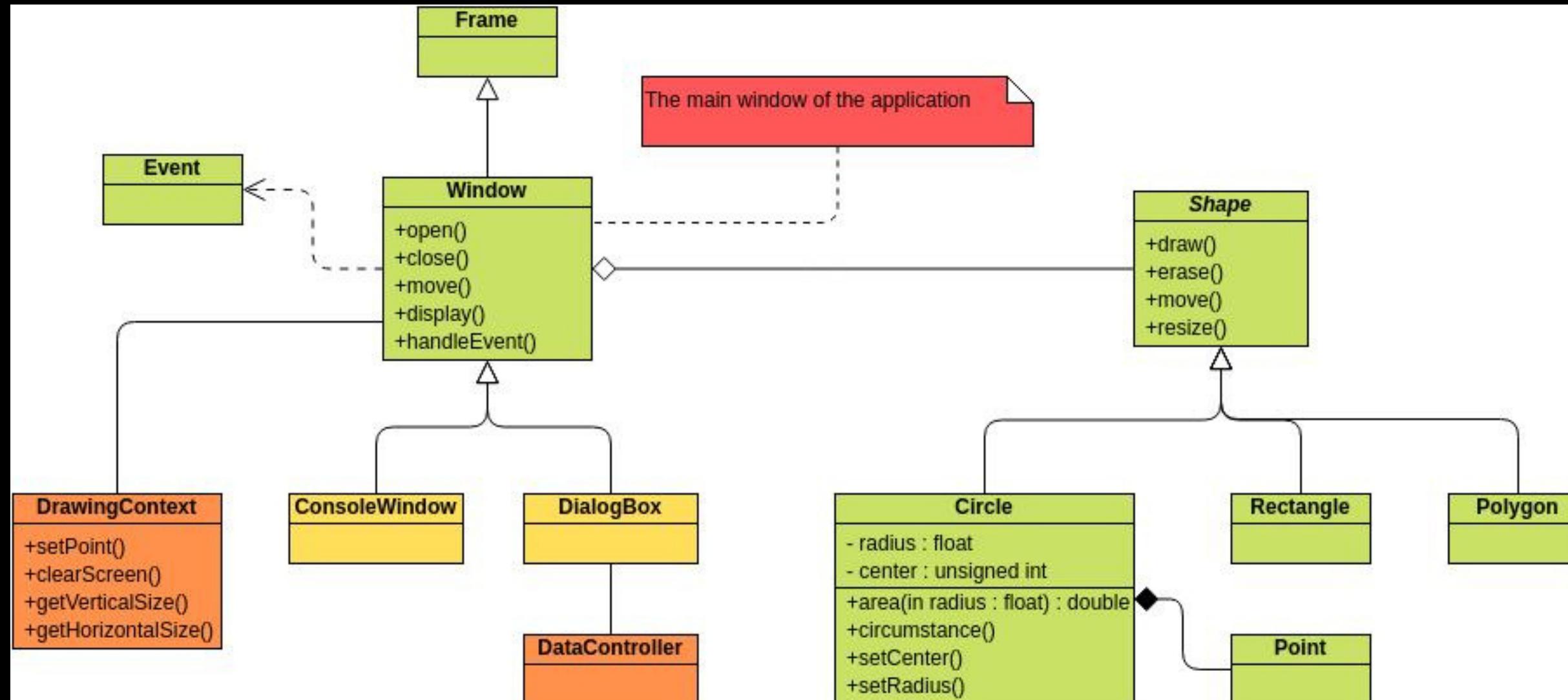
진짜! 모두를 위한
자바스크립트

주문 시스템



<https://blog.visual-paradigm.com/what-are-the-six-types-of-relationships-in-uml-class-diagrams/>

GUI



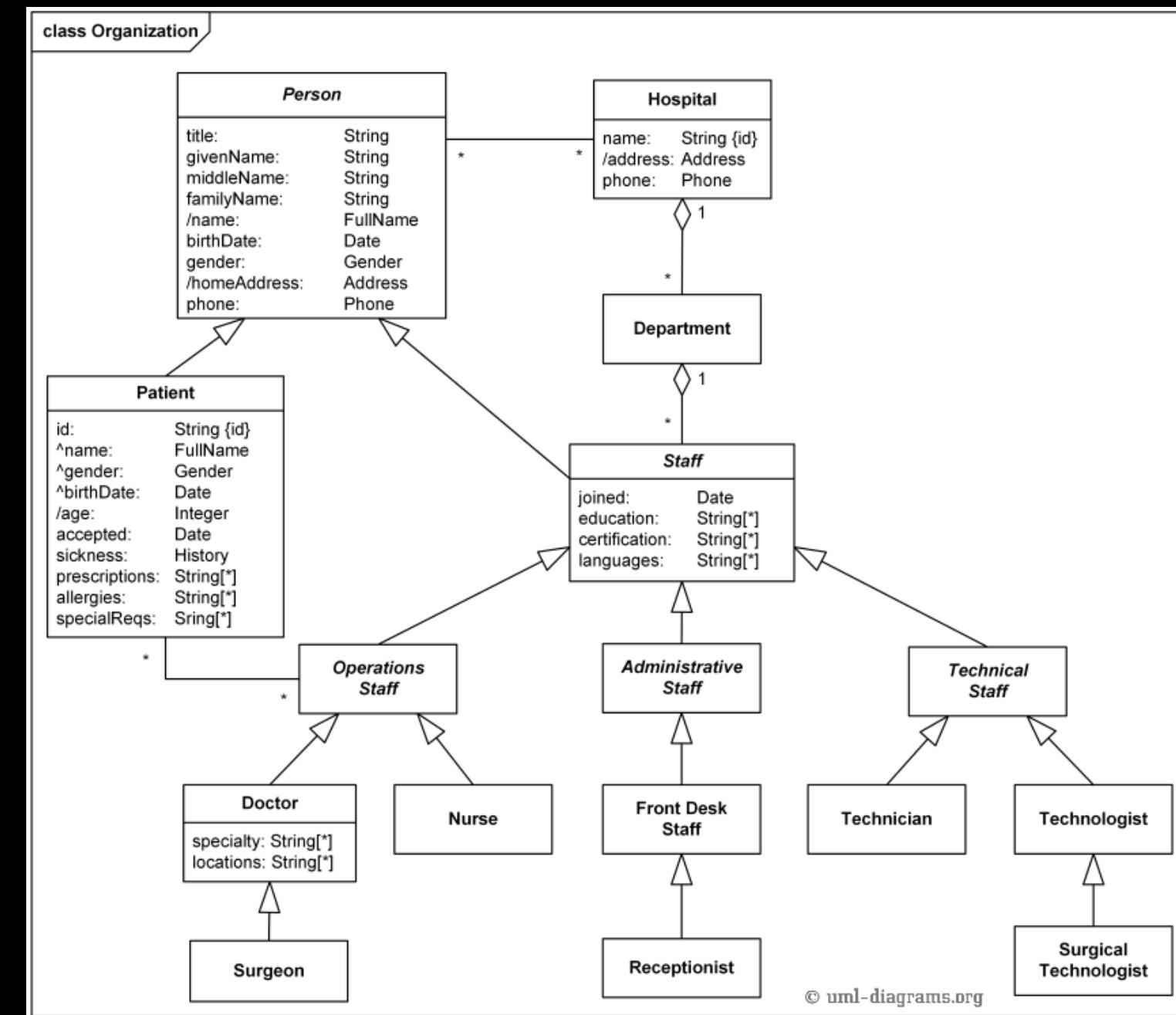
<https://blog.visual-paradigm.com/what-are-the-six-types-of-relationships-in-uml-class-diagrams/>

ES6시대의 OOP

class 키워드와 관계

진짜! 모두를 위한
자바스크립트

병원관리



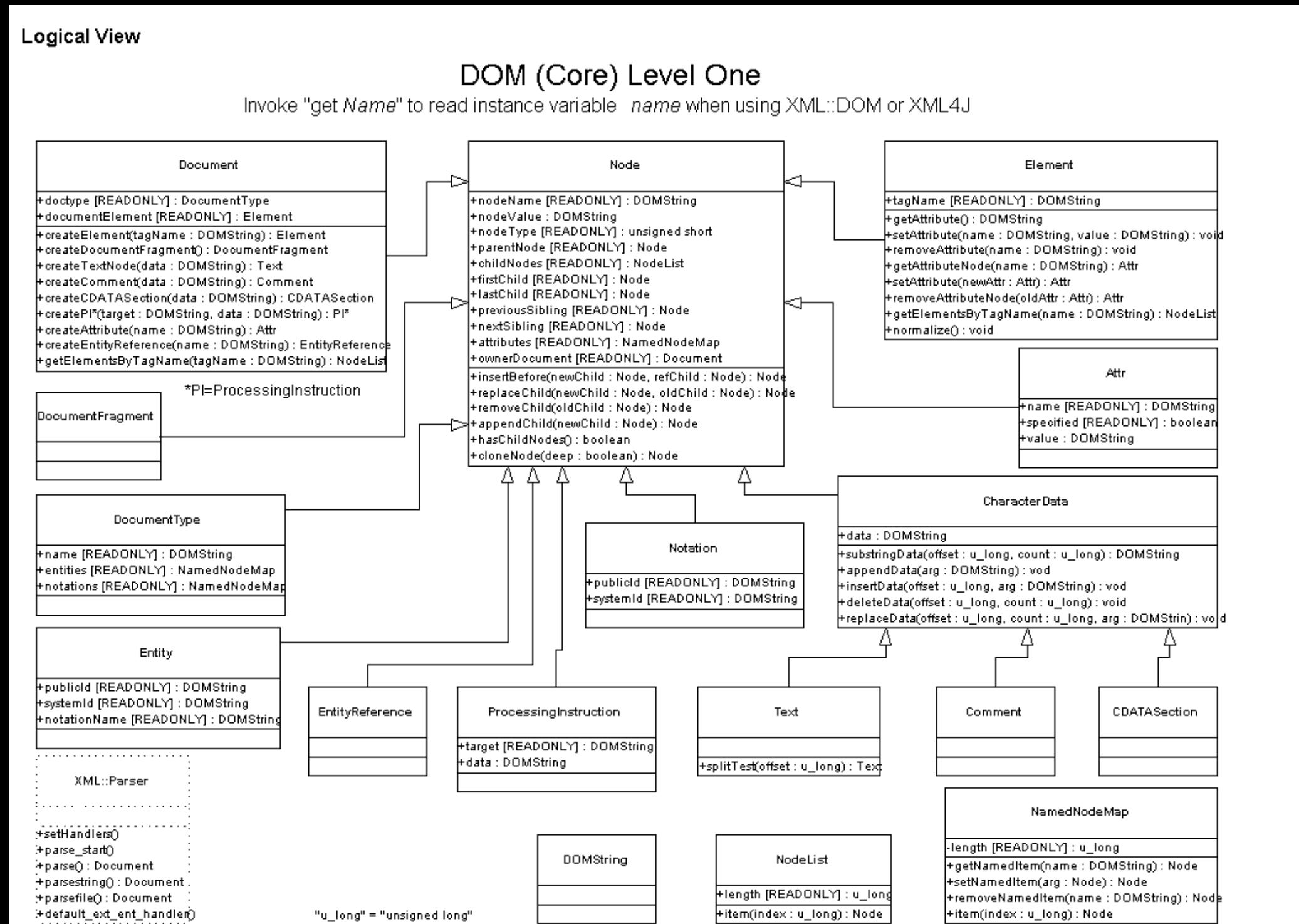
<https://www.uml-diagrams.org/examples/hospital-domain-diagram.html>

ES6시대의 OOP

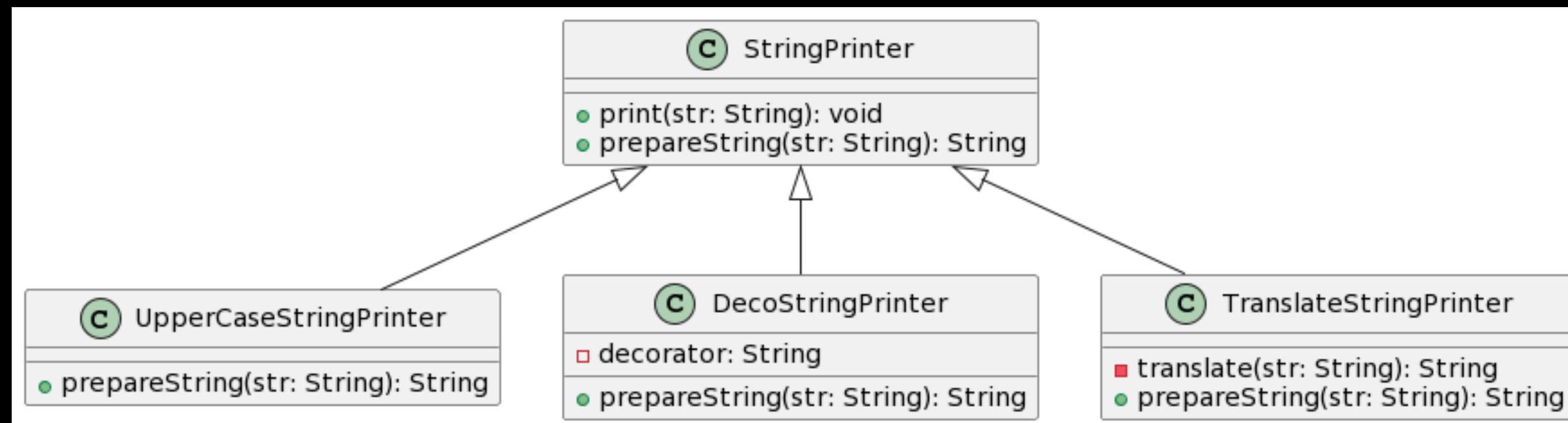
class 키워드와 관계

진짜! 모두를 위한
자바스크립트

DOM



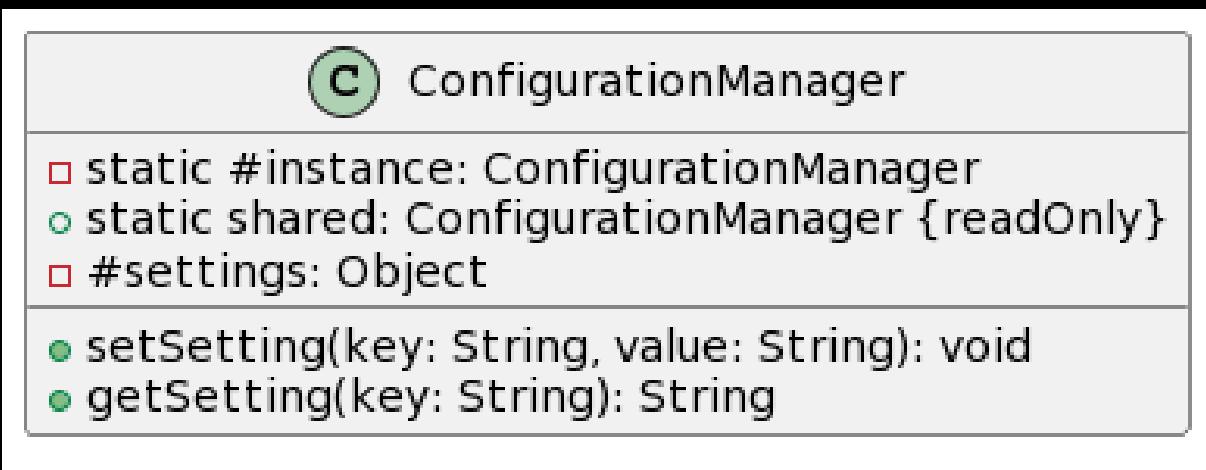
Template Method 디자인 패턴



템플릿 메서드 패턴은 정해진 알고리즘에 커스텀 코드를 적용하는 방법

알고리즘의 특정 단계에서 수행하는 부분을 서브 클래스에서 구현하여 중복 코드를 최대한 줄이고 변경되는 부분만 새로 추가하여 기능을 확장 할 수 있다.

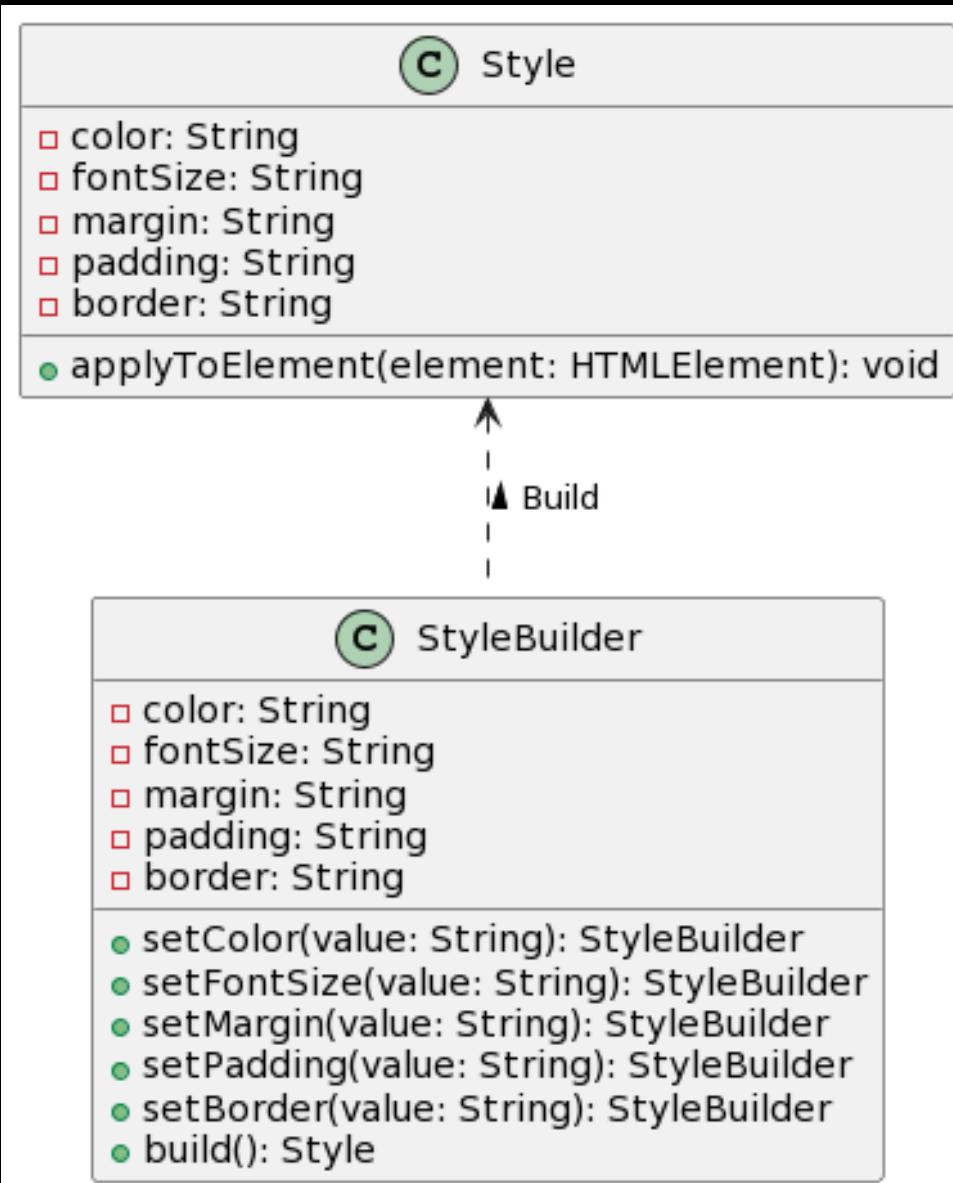
Singleton 디자인 패턴



항상 인스턴스가 특정 n개만 존재할 수 있도록 보장한다.

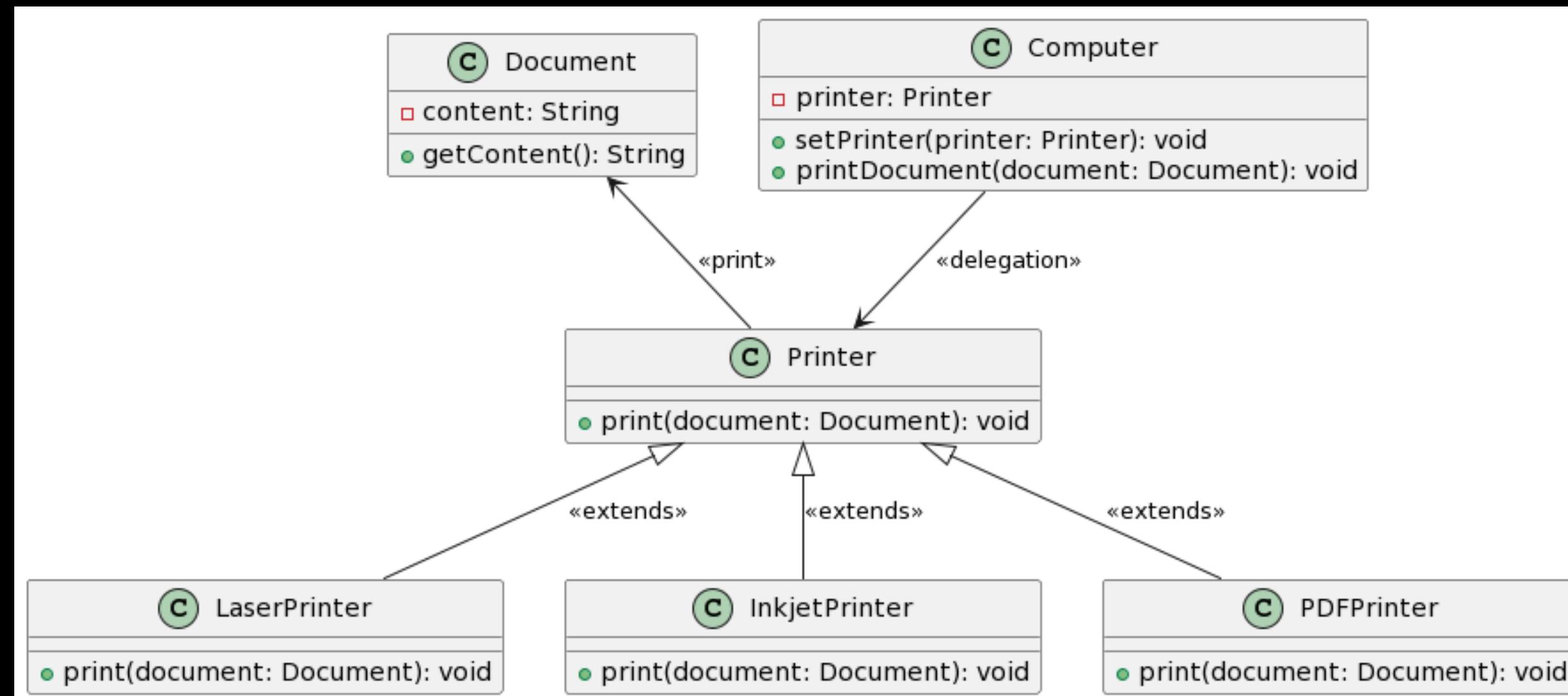
주로 1개의 인스턴스만 필요할 때, Singleton 패턴을 많이 사용한다.

Builder 디자인 패턴



**빌더 패턴은 복잡한 객체를 만들 때,
나누어서 부분별로 생성할 수 있게 도와주는 패턴**

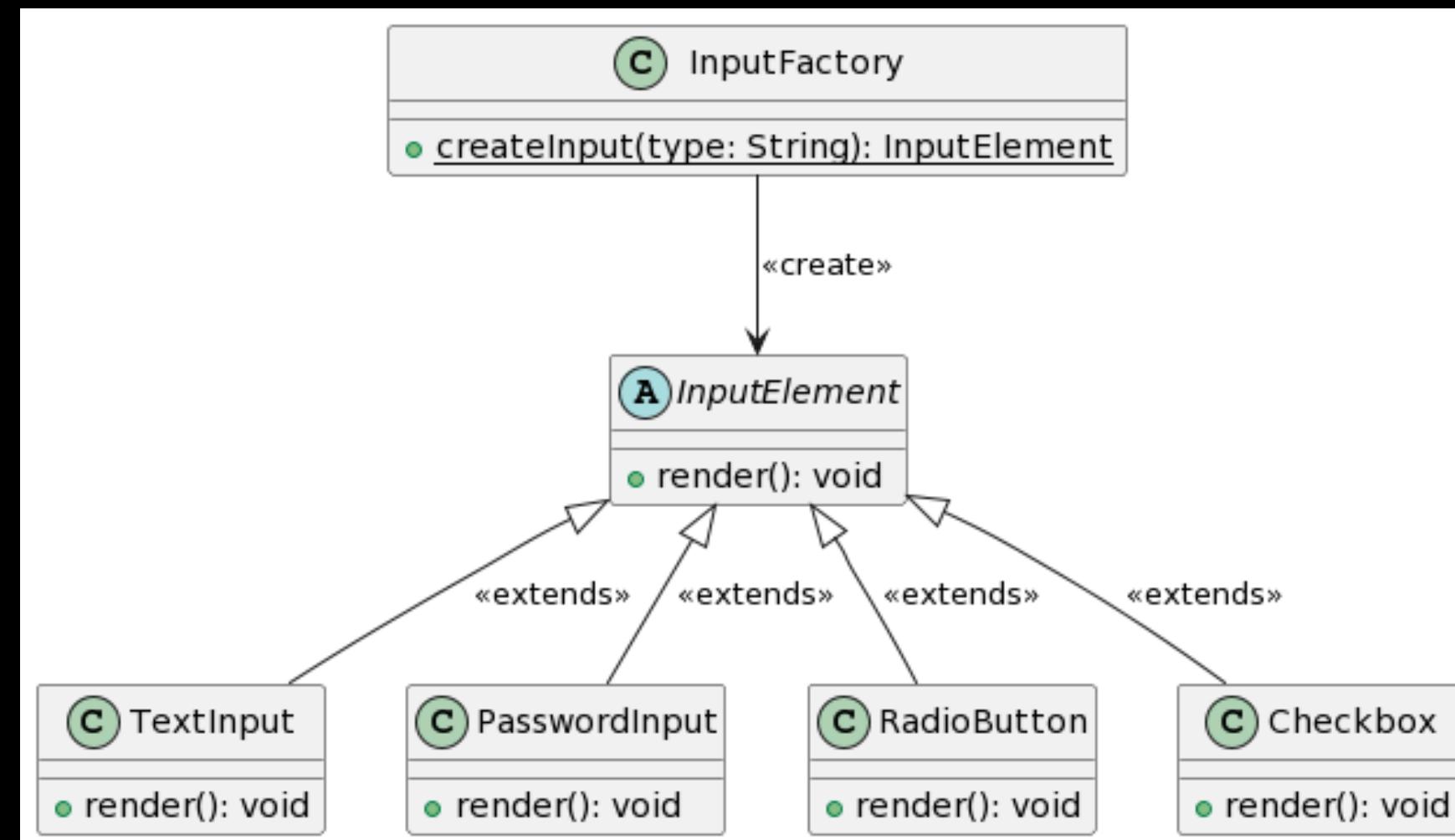
Delegate 디자인 패턴



우리도 일상 생활에서 외주를 주어 일을 해결하는 경우가 많다.

이처럼 델리게이트 패턴은 해당 객체에 주어진 일을 다른 객체에게 위임하여 해결하는 패턴이다.

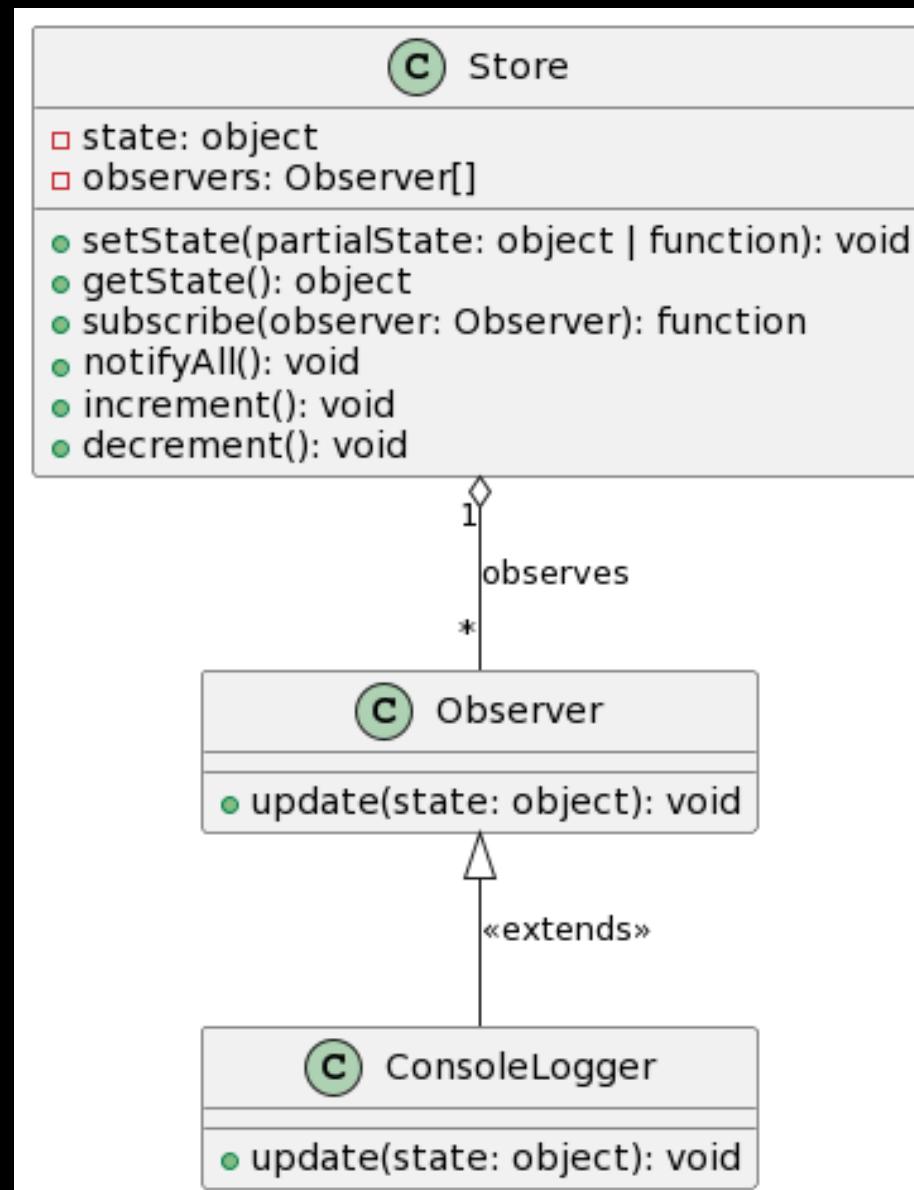
Factory 디자인 패턴



팩토리 패턴은 사용자가 직접 객체를 생성하지 않고
팩토리에 객체 생성을 위임한다.

그렇게 해서, 객체 생성의 복잡한 내용을 사용자에게서
술길 수 있다.

Observer 디자인 패턴

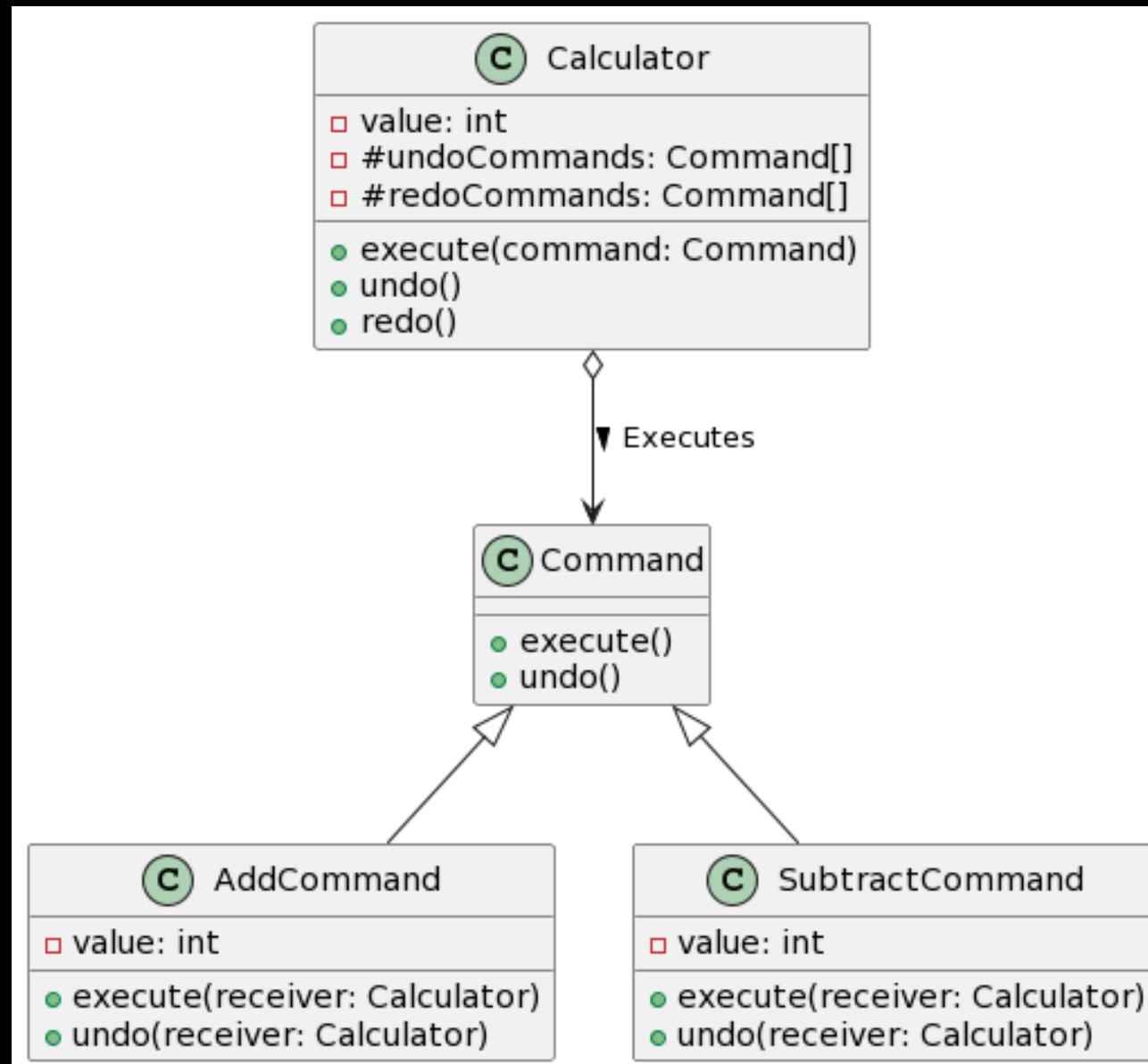


옵저버 패턴은 상태를 가지고 있는 Subject(또는 Observable)와 Observer 객체로 구성된다.

Observer는 Subject의 상태 변화를 관찰하게 되고 Subject의 상태가 변경되면 해당 내용을 통지 받는다.

Reactive 라이브러리들의 기본이 되는 패턴

Command 디자인 패턴



커맨드 패턴은 어떤 객체의 행위(메서드, 비즈니스 로직)를 독립된 객체로 분리하여 구현하는 방법

이렇게 하면 커맨드를 일정 로직에 따라 실행할 수도 있다

부하에 따라 큐에 넣어서 일정 간격으로 실행할 수도 있다.

즉, 요청의 실행을 임의로 지연시킬 수 있다.

ES6시대의 OOP

객체가 갖춰야 할 기본 기능

진짜!
모두를 위한
자바스크립트

Comparable

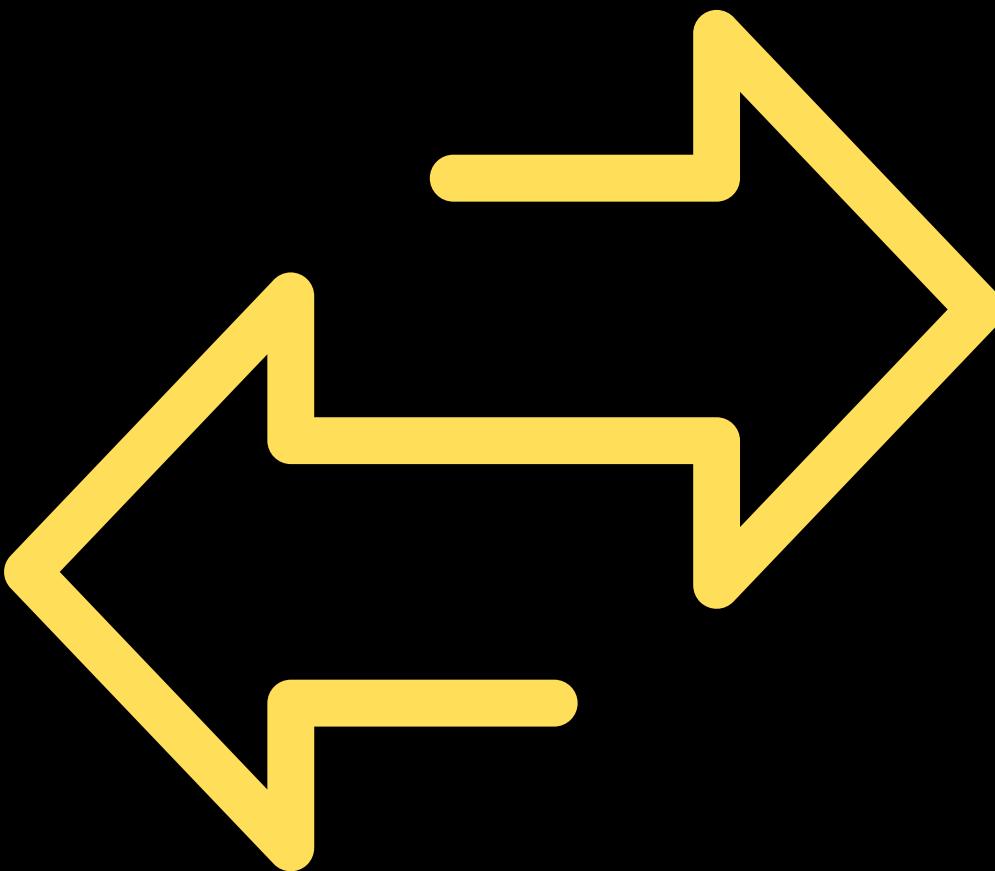


ES6시대의 OOP

객체가 갖춰야 할 기본 기능

진짜! 모두를 위한
자바스크립트

Codable



```
010010010
110011001
001000110
011011000
100100100
010010010
```

ES6시대의 OOP

객체가 갖춰야 할 기본 기능

진짜!
모두를 위한
자바스크립트

Copyable

