OOP(Object Oriented Programing)

Division of Computer Engineering
Byunghwan Jeon, PhD

객체지향(OOP)의 이해

* Object-Oriented Programming(OOP)
* 객체지향(OOP)의 맥락에서 객체(object)는 속성과 함수로 구성

예를 들어 Car 라는 객체가 있을 때,
속성: fuel, speed, steering wheel, coordinate..
함수: accelerate(), takeLeft(), takeRight()..

객체지향 프로그래밍: 객체

객체(object)의 특성

- 객체는 다른 객체와 상호작용하며 목적을 달성
- 예를 들어 객체 Person과 Car는 프로그램 내 각각의 객체지만 Person은 Car를 이용할 수 있다.
- 프로그램으로 무언가를 구현할 때 그 세계관 내부의 의미 있는 정보의 단위들

객체지향(OOP)의 이해

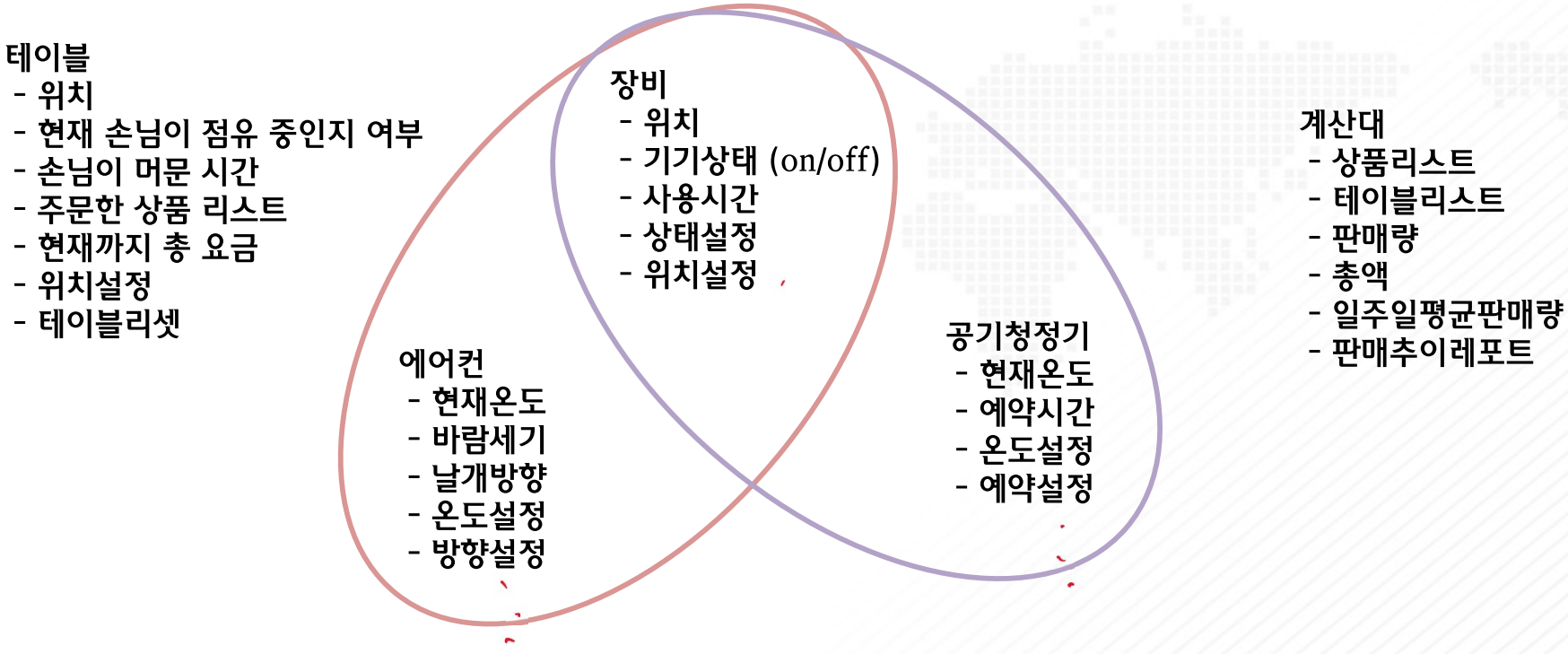
예를 들어 OOP로 커피숍 운영에 대한 scope을 구현한다고 생각해보자.

- 테이블, 장비(에어컨, 냉장고, TV 등), 상품, 재고관리, 계산대와 같은 것들이 객체로 표현할 수 있는 단위로 생각할 수 있음

객체지향(OOP)의 이해

예를 들어 OOP로 IoT기반의 첨단커피숍 운영에 대한 scope을 구현한다고 생각해보자.

- 테이블, 장비(에어컨, 냉장고, 공기청정기 등), 상품, 재고관리, 계산대, 커피 만드는 로봇과 같은 것들이 객체로 표현할 수 있는 단위로 생각할 수 있음
- 추상화 할 때는 실제 눈에 보이는 물리적인 객체가 아닌 추상적인 객체로 생각해야함



객체지향 프로그래밍: 클래스

클래스를 사용하여 특정 객체를 표현할 수 있음

- 클래스는 속성과 행동을 포함하는 객체를 정의함
- 속성은 데이터의 요소이고 함수는 특정 작업을 수행
- 클래스에는 객체의 초기 상태를 설정하는 생성자가 존재함
- 클래스는 일종의 템플릿으로 쉽게 재사용할 수 있음

객체지향 프로그래밍: 메소드

객체지향 프로그래밍에서 메소드의 역할

- 객체의 행위를 나타냄
- 속성을 이용하고 조작하여 task를 수행

OOP의 주요 기능: 캡슐화

한국외국어대학교
HANKUK UNIVERSITY OF FOREIGN STUDIES

- 객체의 기능과 상태 정보를 외부로부터 은닉
- 사용자는 객체의 내부 구조 및 상태를 직접 수정할 수 없고 대신 수정을 요청
- 요청의 종류에 따라 객체는 getter, setter와 같은 특수 함수를 사용하여 내부 상태를 변경

OOP의 주요 기능: 다형성

한국외국어대학교
HANKUK UNIVERSITY OF FOREIGN STUDIES

* 다형성(polymorphism)에는 두 가지 의미가 있음
  - 객체는 함수 인자에 따라 다른 기능을 수행
  - 동일한 인터페이스를 여러 형식의 객체들이 공유함

* 가령, +연산자는 두 정수를 더하는 것도 가능하지만, 문자열을 피연산자로 사용하면 문자열이 합쳐지는 기능이 수행됨

OOP의 주요 기능: 상속

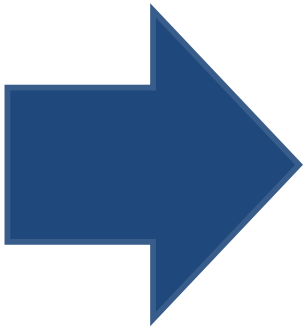
- 어떤 클래스의 기능이 부모 클래스로부터 파생되는 것을 의미
- 부모 클래스에서 정의된 함수를 재사용할 수 있고, 기본기능을 확장할 수 있음
- 클래스간 계층 구조 형성
- 코드의 재사용

OOP의 주요 기능: 추상화

- 공통적인 속성을 묶어서 정의 하는 것을 추상화(abstraction)라고 함
- 흠.. 가만히 보니까 공통적인 속성을 따로 만들고 보니 상위의 개념으로 정의가 가능하군..
- 그리고, 각 기기에 의존되는 속성들에 의해 다른 클래스로 정의 하는 것이 좋겠네..

에어컨
- 위치
- 기기상태 (on/off)
- 사용시간
- 상태설정
- 위치설정
- 현재온도
- 바람세기
- 날개방향
- 온도설정
- 방향설정

공기청정기
- 위치
- 기기상태 (on/off)
- 사용시간
- 상태설정
- 위치설정
- 현재온도
- 예약시간
- 온도설정
- 예약설정



추상화

장비
- 위치
- 기기상태 (on/off)
- 사용시간
- 상태설정
- 위치설정

에어컨
- 현재온도
- 바람세기
- 날개방향
- 온도설정
- 방향설정

공기청정기
- 현재온도
- 예약시간
- 온도설정
- 예약설정

이런 식으로 코드 작성하는 것이 객체지향적 방법

OOP의 주요 기능: 컴포지션

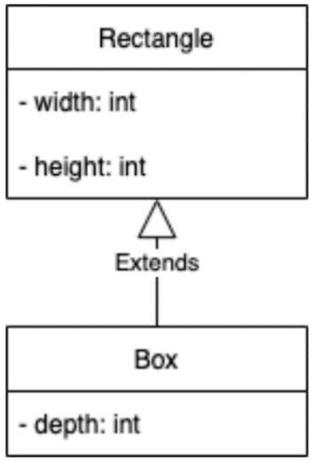
- 객체나 클래스를 더 복잡한 자료 구조나 모듈로 묶는 행위
- 컴포지션을 통해 특정 객체는 다른 모듈의 함수를 호출 할 수 있음
- 즉, 상속 없이 외부 기능을 사용가능
- Loose Coupling (느슨한 결합): 시스템 내 여러 부분이 독립적으로 존재하며, 변경사항이 한 부분에 영향을 미치지 않음

```java
// Engine class
class Engine {
    public void start() {
        System.out.println("Engine started.");
    }
}

// Car class, which contains an Engine
class Car {
    private Engine engine;

    public Car() {
        // Composition: Car "has-a" Engine
        this.engine = new Engine();
    }

    public void drive() {
        engine.start(); // The Car uses the Engine's functionality
        System.out.println("Car is driving.");
    }
}

public class Main {
    public static void main(String[] args) {
        Car car = new Car(); // Create a Car
        car.drive();         // The Car uses its Engine to drive
    }
}
```

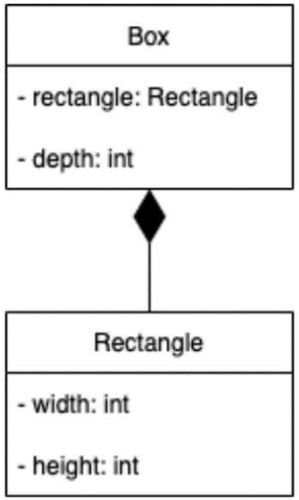
이 엔진은 언제든지 교체 가능!

상속 vs 컴포지션

Inheritance



Composition



Object Oriented Programming

SOLID principles

Division of Computer Engineering
Byunghwan Jeon, PhD

SOLID principles

- Single Responsibility
- Open-Closed
- Liskov Substitution
- Interface Segregation
- Dependency Inversion

Single Responsibility Principle (SRP)

- **정의: 단일 책임 원칙**(single responsibility principle)**이란 모든 클래스는 하나의 책임만 가져야 함**
- **단일 책임 원칙이란 클래스는 하나의 책임만을 가져야 한다.**
- **클래스를 구현할 때 한가지 기능에만 중점을 둬야 한다. 두 가지 이상의 기능이 필요하다면 클래스를 나눠야 한다.**
- **특정 기능을 수정할 때 관련 클래스 외에는 건드릴 필요가 없다.**
- **한 개의 클래스에 여러 가지 기능이 있다면 관련된 모든 클래스를 수정해야하는 상황이 발생할 수 있다.**

Single Responsibility Principle (SRP)

한국외국어대학교 HANKUK UNIVERSITY OF FOREIGN STUDIES

```java
public class NotFollowingSRP {

	public int addPrint(int num1, int num2) {
		int result = num1 + num2;
		System.out.println(result);
		return result;
	}

	public static void main(String[] args) {
		NotFollowingSRP nsrp = new NotFollowingSRP();
		nsrp.addPrint(3, 4);
	}
}
```

원칙을 따름

```java
public class SingleResponsibility {

	public int add(int num1, int num2) {
	return num1 + num2;
	}

	public void numPrint(int num) {
	System.out.println(num);
	}

	public static void main(String[] args) {
		SingleResponsibility srp = new SingleResponsibility();
		int result = srp.add(3, 4);
		srp.numPrint(result);
	}
}
```

원칙을 따르지 않음

- Add 함수는 덧셈을 하는 것에만 책임을 다 해야 함
- numPrint 함수는 숫자를 출력하는 것에만 책임을 다 해야함
- addPrint함수는 Single Responsibility 원칙을 따르지 않음

Single Responsibility Principle (SRP)

한국외국어대학교 HANKUK UNIVERSITY OF FOREIGN STUDIES

- 그럼 클래스로 작성된 코드 예시를 보자.

```java
class Cat {
    private int age;
    private String name;

    public Cat(int age, String name) {
       this.age = age;
       this.name = name;
    }

    public void eat(String food) {
        // Code for eating
    }

    public void walk() {
        // Code for walking
    }

    public void speak() {
        // Code for speaking
    }

    public String repr() {
        return "age: " + age + " name: " + name;
    }

    public static void main(String[] args) {
        Cat kitty = new Cat(2, "Whiskers");
        System.out.println(kitty.repr());
    }
}
```

원칙을 따름

```java
class Cat {
    private int age;
    private String name;

    public Cat(int age, String name) {
        this.age = age;
        this.name = name;
    }

    public void eat(String food) {
        // Code for eating
    }

    public void walk() {
        // Code for walking
    }

    public void speak() {
        // Code for speaking
    }

    public void print() {
        System.out.println("age: " + age + " name: " + name);
    }

    public void log(Logger logger) {
        logger.log("age: " + age + " name: " + name);
        logger.log(java.time.LocalDateTime.now().toString());
    }

    public static void main(String[] args) {
        Cat kitty = new Cat(2, "Whiskers");
        kitty.print();
    }
}
```

원칙을 따르지 않음

Open-close Principle

- **개방-폐쇄 원칙**(OCP, Open-Closed Principle)은 '소프트웨어 개체(클래스, 모듈, 함수 등등)는 확장에 대해 열려 있어야 하고, 수정에 대해서는 닫혀 있어야 한다'는 프로그래밍 원칙

- **확장에 대해서는 개방**
- **수정에 대해서는 폐쇄**

```
class Animal {
    String type;

    public Animal(String type) {
        this.type = type;
    }

    public void speak() {
        if (type.equals("Cat")) {
            System.out.println("meow");
        } else if (type.equals("Dog")) {
            System.out.println("bark");
        } else {
            throw new IllegalArgumentException("Wrong animal type");
        }
    }

    public static void main(String[] args) {
        Animal kitty = new Animal("Cat");
        Animal bingo = new Animal("Dog");
        Animal cow = new Animal("Cow"); // This would throw an error
        kitty.speak();
        bingo.speak();
    }
}
```

- **기능이 추가될 때마다 수정이 필요한 코드**

Open-close Principle

한국외국어대학교 Hankuk University of Foreign Studies

Interface

```java
abstract class Animal {
    public abstract void speak();
}

// Cat class extending Animal
class Cat extends Animal {
    @Override
    public void speak() {
        System.out.println("meow");
    }
}

// Dog class extending Animal
class Dog extends Animal {
    @Override
    public void speak() {
        System.out.println("bark");
    }
}

// Sheep class extending Animal
class Sheep extends Animal {
    @Override
    public void speak() {
        System.out.println("meh");
    }
}

// Cow class extending Animal
class Cow extends Animal {
    @Override
    public void speak() {
        System.out.println("moo");
    }
}
```

No need to modify

```java
// The Hey function that calls speak for any Animal
class Zoo {
    public void hey(Animal animal) {
        animal.speak();
    }

    public static void main(String[] args) {
        Zoo zoo = new Zoo();

        Animal cow = new Cow();
        Animal sheep = new Sheep();
        Animal cat = new Cat();
        Animal dog = new Dog();

        zoo.hey(cow);
        zoo.hey(sheep);
        zoo.hey(cat);
        zoo.hey(dog);
    }
}
```

- 기능이 추가되어도 수정이 필요하지 않은 코드

Dependency Inversion Principle

- 의존관계 역전 원칙은 소프트웨어 모듈들을 분리하는 특정 형식을 지칭
- 이 원칙을 따르면, 상위 계층이 하위 계층에 의존하는 전통적인 의존관계를 반전(역전)시킴으로써 상위 계층이 하위 계층의 구현으로부터 독립되게 할 수 있다.

```
class Cat {
    public void speak() {
        System.out.println("meow");
    }
}

class Dog {
    public void speak() {
        System.out.println("bark");
    }
}

class Zoo {
    private Cat cat;
    private Dog dog;

    public Zoo() {
        this.cat = new Cat();
        this.dog = new Dog();
    }

    public void makeSound() {
        cat.speak();
        dog.speak();
    }

    public static void main(String[] args) {
        Zoo zoo = new Zoo();
        zoo.makeSound();
    }
}
```

수정 필요

Zoo 클래스는 Cat, Dog 클래스들에 의존관계가 있음
동물들이 늘어나면 Zoo클래스를 계속 수정해야함 (ex. Sheep이 추가된다면?)

Dependency Inversion Principle

한국외국어대학교 HANKUK UNIVERSITY OF FOREIGN STUDIES

Interface

```
interface Animal {
    void speak();
}

class Cat implements Animal {
    @Override
    public void speak() {
        System.out.println("meow");
    }
}

class Dog implements Animal {
    @Override
    public void speak() {
        System.out.println("bark");
    }
}

class Sheep implements Animal {
    @Override
    public void speak() {
        System.out.println("meh");
    }
}

class Cow implements Animal {
    @Override
    public void speak() {
        System.out.println("moo");
    }
}
```

Polymorphism

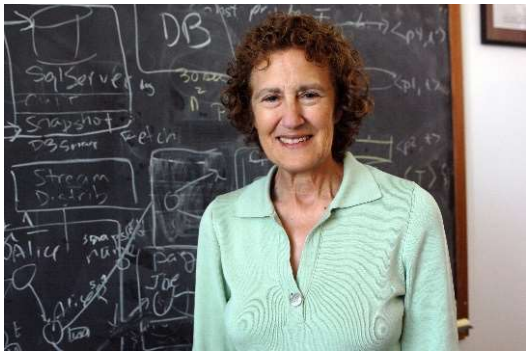
```
class Zoo {
    private Animal[] animals;

    public Zoo(Animal[] animals) {
        this.animals = animals;
    }

    public void makeSound() {
        for (Animal animal : animals) {
            animal.speak();
        }
    }

    public static void main(String[] args) {
        Animal[] animals = { new Cat(), new Dog(), new Sheep(), new Cow() };
        Zoo zoo = new Zoo(animals);
        zoo.makeSound();
    }
}
```

의존관계가 Animal 추상클래스에 의해 역전되었고, Zoo 클래스는 각 Cat, Dog 등의 클래스로부터 독립되었음

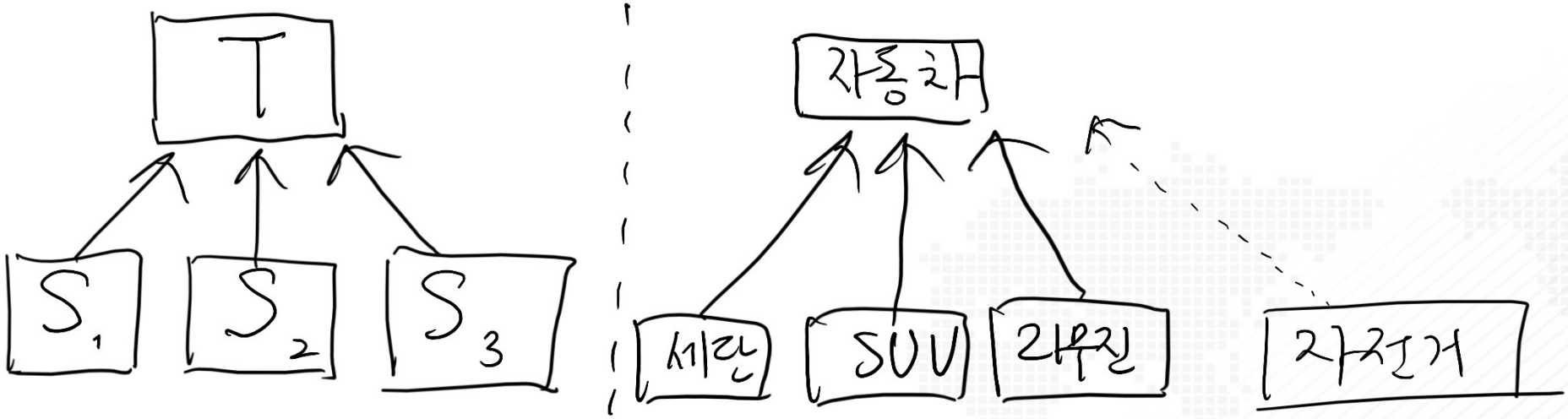
Liskov Substitution principle

한국외국어대학교
HANKUK UNIVERSITY OF FOREIGN STUDIES

정의: 컴퓨터 프로그램에서 자료형 S가 자료형 T의 하위형이라면 필요한 프로그램의 속성(정확성, 수행하는 업무 등)의 변경 없이 자료형 T의 객체를 자료형 S의 객체로 교체(치환)할 수 있어야 한다는 원칙



Barbara Liskov



추상클래스와 구체클래스의 관계를 잘 생각해보자

가령 “자전거는 자동차인가?” 라는 질문을 던져보자

Liskov Substitution principle


```java
// Base class Cat
class Cat {
    public void speak() {
        System.out.println("meow");
    }
}

// Subclass BlackCat adhering to LSP
class BlackCat extends Cat {
    @Override
    public void speak() {
        System.out.println("black meow");
    }
}

// Fish class mistakenly extending Cat (violating LSP)
class Fish extends Cat {
    @Override
    public void speak() {
        throw new UnsupportedOperationException("Fish cannot speak.");
    }
}

public class Main {
    public static void main(String[] args) {
        // LSP valid substitution
        Cat cat = new Cat();
        cat.speak();  // Output: meow

        Cat blackCat = new BlackCat();
        blackCat.speak();  // Output: black meow

        // LSP violation: Fish class extends Cat but does not follow the speak() behavior
        try {
            Cat fish = new Fish();
            fish.speak();  // Throws exception
        } catch (UnsupportedOperationException e) {
            System.out.println(e.getMessage());  // Output: Fish cannot speak.
        }
    }
}
```

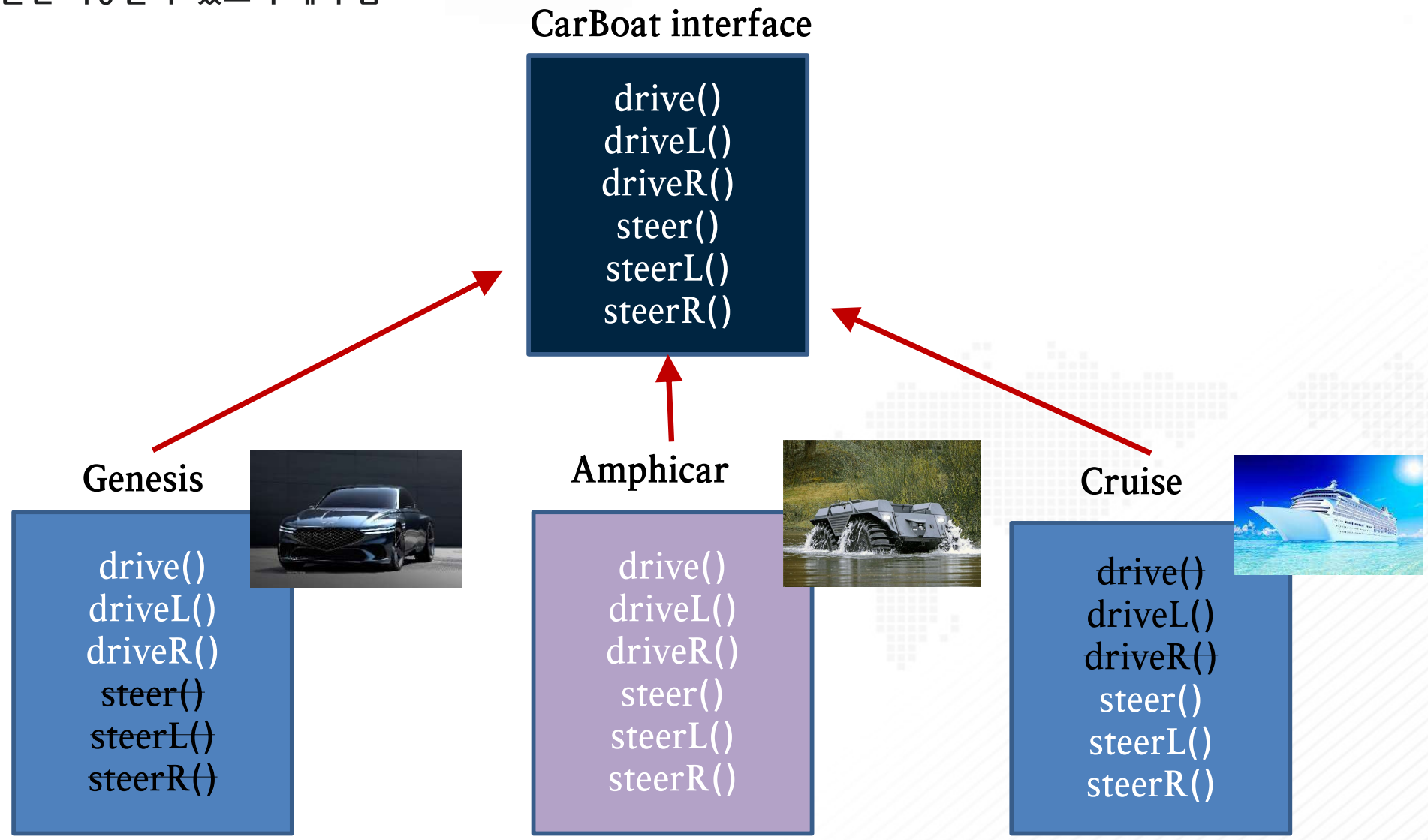
Cat 인터페이스를 사용하는 다른 함수로 Fish를 넘기게될 경우 오류가 날 수 있음

즉, 치환이 안됨

- Fish와 관련된 클래스는 따로 설계해야 함
- 처음 설계할 때부터 고려 했어야 함

Interface Segregation

- 인터페이스 분리 원칙은 클라이언트가 자신이 이용하지 않는 메서드에 의존하지 않아야 한다는 원칙: 큰 덩어리의 인터페이스들을 구체적이고 작은 단위들로 분리시킴으로써 클라이언트들이 꼭 필요한 메서드들만 이용할 수 있도록 해야 함



Interface Segregation