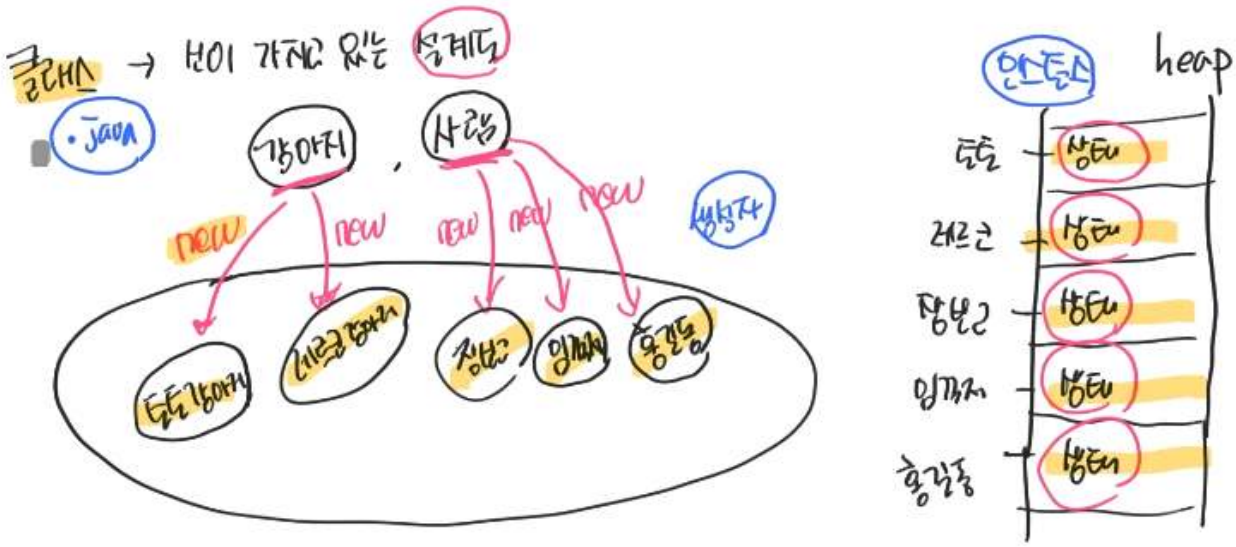
# Chapter 5. 객체지향 프로그래밍 문법

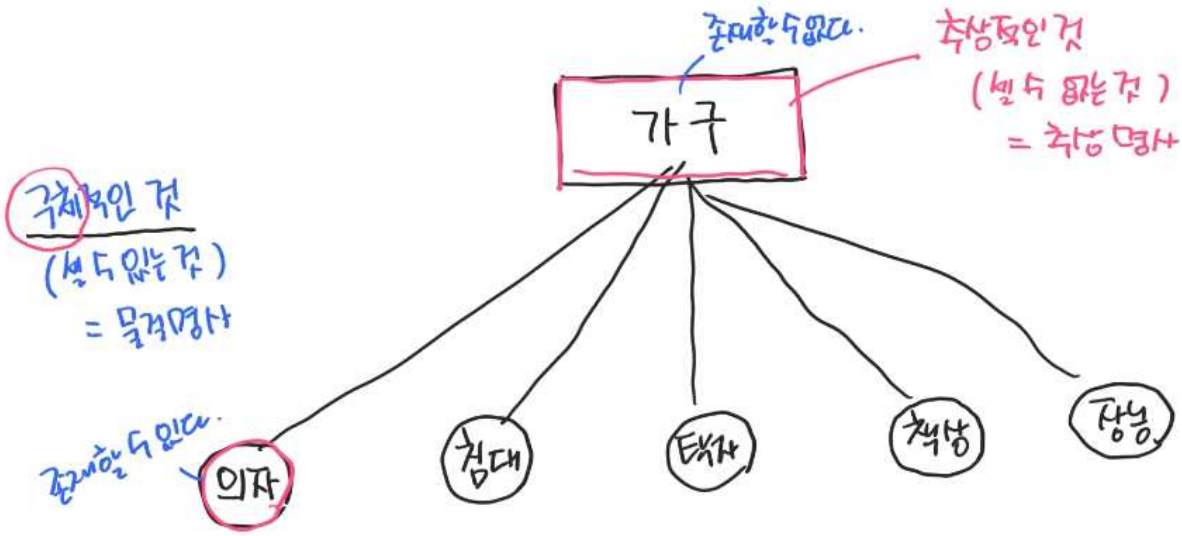
## 35장. 클래스와 오브젝트와 인스턴스

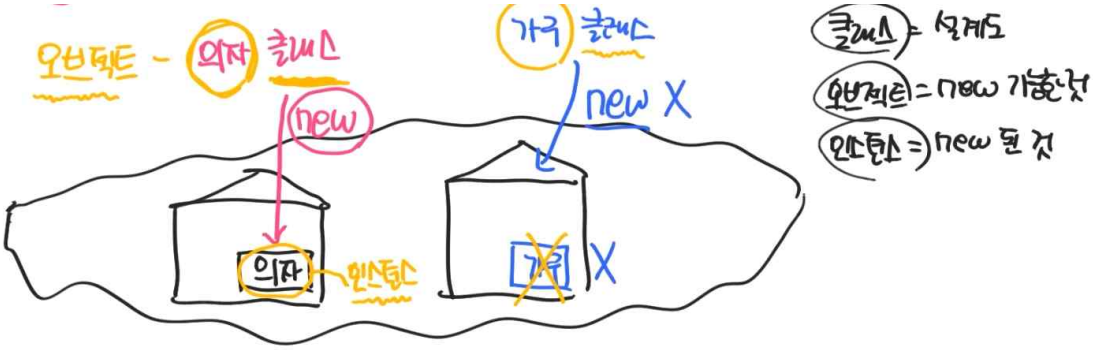
▷ 클래스 : 신이 갖고 있는 설계도(.java 파일)

▷ 인스턴스 : 힙(Heap) 공간에 떠 있는 즉, 메모리에 올라온 오브젝트




▷ 오브젝트 : new 가능한 것.

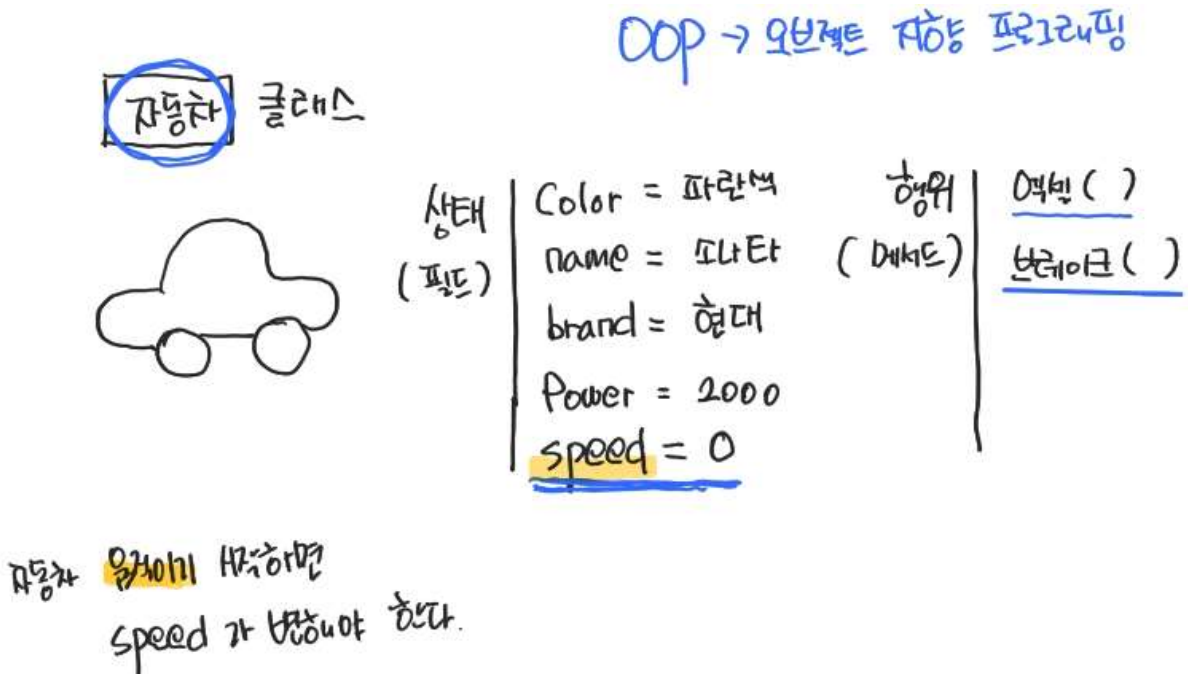



오브젝트 - 의자 클래스
new
의자
인스턴스
가구 클래스
new X
가구
X
클래스 = 설계도
오브젝트 = new 가능한 것
인스턴스 = new 된 것

## 36장. 상태는 행위를 통해 변경한다.

▷ 클래스의 상태(행위)는 메서드에 의해서 변한다.

▶ 상태(필드)를 구성할 때는 private 접근 제어자를 통해 외부 접근을 제어하고 Getter와 Setter를 통해 상태값을 변경하도록 구성한다. → 이렇게 해야 객체지향프로그램이라 할 수 있다.



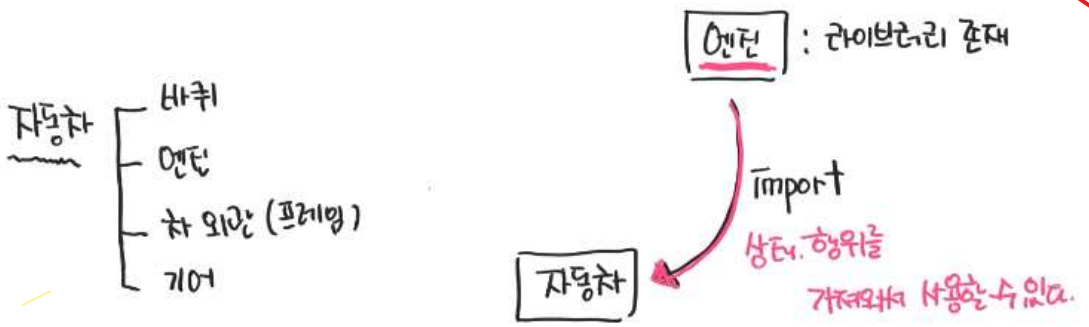

```
package ch05;
class Player{
  // 상태 = 필드
  String name;    // 이름
  private int thirty;        // 목마름 → 외부 클래스에서 접근 불가능
  public Player(String name, int thirty)
  {
    this.name = name;
    this.thirsty = thirsty;
  }
  // private thirty 변수에 접근 가능한 메서드 필요
  int 목마름상태확인(){
    return this.thirsty;
  }

  // 행위 = 메서드
  void 물마시기(){
    System.out.println("물마시기 행위");
    this.thirsty = this.thirsty - 50;
  }
```

```
}
public class OOPEx01{
   public static void main(String[] args){
      Player p1 = new Player("홍길동", 100);
      System.out.println("이름은 : " + p1.name);
//      System.out.println("갈증지수 : " + p1.thirsty);   → 접근불가능하므로 오류
      System.out.println("갈증지수 : " + p1.목마름상태확인());

      // 1. 첫번째 시나리오 = 마법(×)
      // p1.thirsty = 50;  // 원인없이 갈증지수 변경 → 마법
      // System.out.println("갈증지수 : " + p1.thirsty);
      // ▶ 위 처럼 필드를 원인 없이 상태가 강제로 변경되는 것은 객체지향 프로그램이 아니다.

      // 2. 두번째 시나리오 = 상태가 행위를 변경함(×) - 신입이 실수 할 수 있다.
      //p1.물마시기;
      //p1.thirsty = 50;  // 메서드 실행없이 직접 변수를 변경해서 변경 가능함.
      //System.out.println("갈증지수 : " + p1.thirsty);

      // 3. 세번째 시나리오 = 접근제어자를 통해 접근을 제어함.
      p1.물마시기;  // 실수할 수가 없음.
      System.out.println("갈증지수 : " + p1.목마름상태확인());
   }
}
```

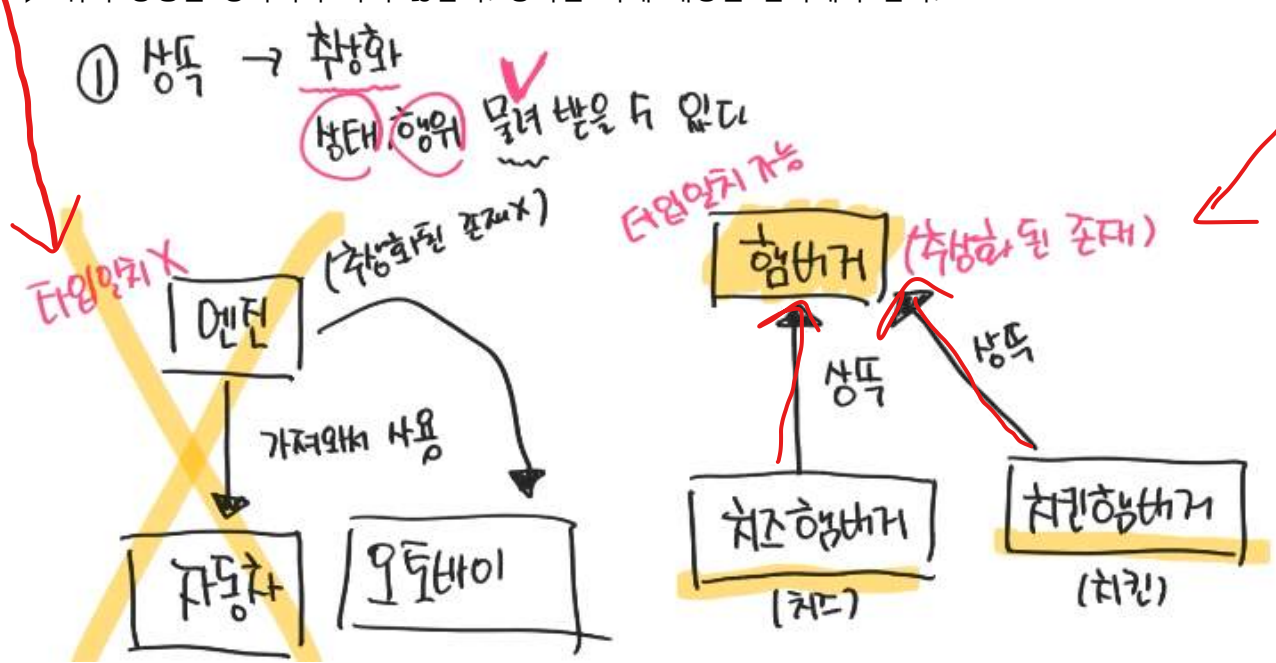
## 37장. 상속과 콤포지션

▷ 상속보다는 extend 즉, 확장하다라는 의미가 더 적절함.

▷ 콤포지션 : 결합, 잘 만들어진 클래스를 재사용하는 것.




▶ 위의 상황을 상속이라 하지 않는다. 상속은 아래 내용을 만족해야 한다.

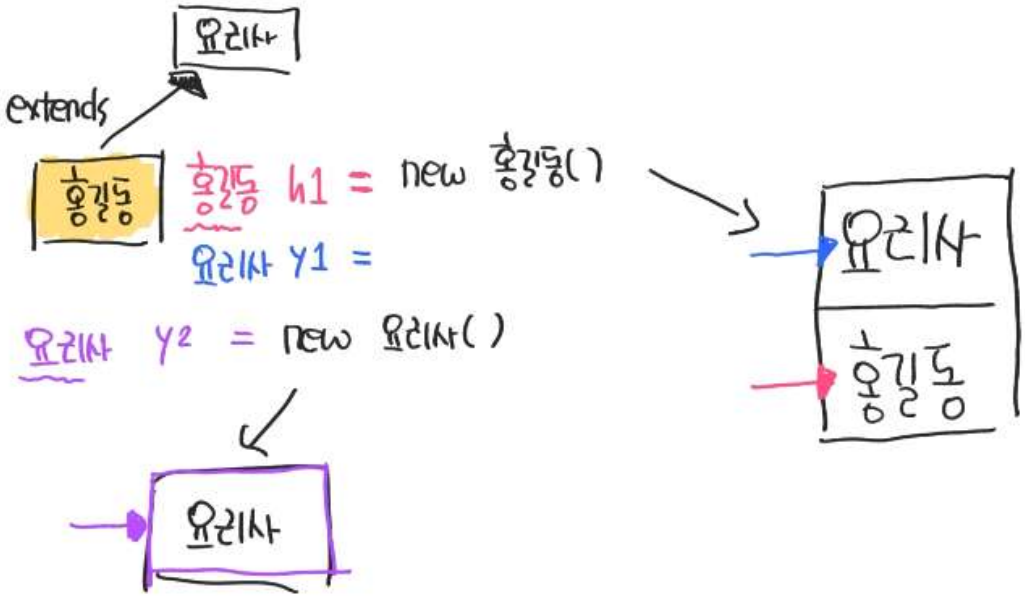



```
package ch05;
class Engine{
  int power = 2000;
}
class Car{  // 자동차는 엔진이 아니기 때문에 상속할 수 없다.
  // 콤포지션!!  결합 → 잘 만들어진 것을 가져다 사용하는 것.
  Engine e;
  public Car(Engine e){
    this.e = e;
  }
```

```
}

class Hamburger{
  String name = "기본햄버거";
  String 재료1 = "번";
  String 재료2 = "양상추";
}
// 상속은 상태와 행위를 물려받을 수 있지만 꼭 타입이 일치되어야 한다.
class CheeseHamburger extends Hamburger{ // 치즈햄버거 = 햄버거이다.
  // 겹치지 않는 상태(필드)만 물려받는다.
  String name = "치즈햄버거";
}

class ChickenHamburger{
  String name = "치킨햄버거";
  Hamburger h;
  public ChickenHamburger(Hamburger h){
    this.h = h;
  }
}

public class OOPEx02{
  public static void main(String[] args){
    Engine e1 = new Engine();
    Car c1 = new Car(e1);
    System.out.println("자동차의 엔진 마력은 : " + c1.e.power);

    CheeseHamburger ch1 = new CheeseHamburger();
    System.out.println("햄버거의 이름은 : " + ch1.name);
    System.out.println("재료 : " + ch1.재료1);
    System.out.println("재료 : " + ch1.재료2);

    Hamburger h1 = new Hamburger();
    ChickenHamburger chk1 = new ChickenHamburger(h1)
    System.out.println("햄버거의 이름은 : " + chk1.name);
    System.out.println("재료 : " + chk1.h.재료1);
    System.out.println("재료 : " + chk1.h.재료2);
  }
}
```

## 38장. 다형성

▷ 상속받은 클래스가 다양한 성격을 갖는 클래스로 만들어 지는 것. 즉, 요리사 직업을 상속받은 홍길동을 heap 메모리에 띄우면, 홍길동은 홍길동 본인 혹은 요리사 두 가지 성격을 다양하게 갖는다. 이를 다형성이라 한다.




```
package ch05;
// 단지 요리사
class 요리사{
   String name = "요리사";
}
// 홍길동 or 요리사
class 홍길동 extends 요리사{
   String name = "홍길동";
}

public class OOPEx03{
   public static void main(String[] args){
      홍길동 h1 = new 홍길동();      // (홍길동, 요리사)
      System.out.println(h1.name);

      요리사 y1 = new 홍길동();      // (홍길동, 요리사) → 요리사 바라봄.
      System.out.println(y1.name);

      //홍길동 h2 = new 요리사();  // 불가능 함.  메모리에 요리사만 뜨기 때문에
   }
}
```

# 39장. 오버로딩

▷ Overloading = 과적재 : 특정 클래스 내의 행위(메서드)를 구성할 때 같은 이름으로 구성이 가능하다. 단, 매개변수의 갯수 또는 타입이 달라야 한다.

```
class 클래스{
   달리기(){
   }
   달리기(int speed){
   }
   달리기(double speed){
   }
   달리기(int speed, double power){
   }
}
```

```
package ch05;
class 임꺽정{
   void 달리기(){
      System.out.println("달리기");
   }
   // 오버로딩
   void 달리기(int speed){
      System.out.println("달리기2");
   }
   // 오버로딩
   void 달리기(double speed){
      System.out.println("달리기3");
   }
   // 오버로딩
   void 달리기(int speed, double power){
      System.out.println("달리기3");
   }
}

public class OOPEx04{
   public static void main(String[] args){
      임꺽정 e = new 임꺽정();
      e.달리기();
      e.달리기(2);
      e.달리기(5.0);
```

```
        e.달리기(1,  5.0);
    }
}
```

# 40장. 오버로딩의 한계

```
package ch05;
class 전사{  // 검
  String name = "전사";
  void 기본공격(){
    System.out.println("검으로 공격하기");
  }
  void 기본공격(궁수 e1){
    System.out.println("검으로 " + e1.name+ " 공격하기");
  }
}
class 궁수{  // 활
  String name = "궁수";
  void 기본공격(){
    System.out.println("활로 공격하기");
  }
  void 기본공격(광전사 e1){
    System.out.println("활로 " + e1.name + " 공격하기");
  }
}
class 광전사{  // 도끼
  String name = "광전사";
  void 기본공격(){
    System.out.println("도끼로 공격하기");
  }
  void 기본공격(전사 e1){
    System.out.println("도끼로 " + e1.name + " 공격하기");
  }
}

public class OOPEx04{

  public static void main(String[] args){
    전사 u1 = new 전사();
    궁수 u2 = new 궁수();
    광전사 u3 = new 광전사();

    u1.기본공격(u2);
    u2.기본공격(u3);
    u3.기본공격(u1);
```

```
  }
}
```

▷ 현재 상태는 전사는 궁수만, 궁수는 광전사만, 광전사는 전사만 공격이 가능하다.

▶ 이를 다른 방식으로 바꿔 누구나 공격할 수 있도록 하고 싶다.

```
package ch05;
class 전사{  // 검
  String name = "전사";
  void 기본공격(){
    System.out.println("검으로 공격하기");
  }
  void 기본공격(궁수 e1){
    System.out.println("검으로 " + e1.name+ " 공격하기");
    void 기본공격2(광전사 e1){   // 이름을 달리주어 새로 생성
    System.out.println("검으로 " + e1.name+ " 공격하기");
  }

  // 그래서 오버라이딩을 하면 함수를 이해하고 외우기 쉽다.
  // 그리고, 하나의 함수로 다양한 기능을 처리할 수 있다.
  void 기본공격(광전사 e1){   // 이름을 달리주어 새로 생성
    System.out.println("검으로 " + e1.name+ " 공격하기");
  }
}
class 궁수{  // 활
  String name = "궁수";
 /void 기본공격(){
    System.out.println("활로 공격하기");
  }
  void 기본공격(전사 e1){
    System.out.println("활로 " + e1.name + " 공격하기");
  }
  void 기본공격(광전사 e1){
    System.out.println("활로 " + e1.name + " 공격하기");
  }
}
class 광전사{  // 도끼
  String name = "광전사";
  void 기본공격(){
    System.out.println("도끼로 공격하기");
  }
```

```
  void 기본공격(전사 e1){
    System.out.println("도끼로 " + e1.name + " 공격하기");
  }
  void 기본공격(광전사전사 e1){
    System.out.println("도끼로 " + e1.name + " 공격하기");
  }
}

class 마법사{  // 마법
  String name = "마법사";
  void 기본공격(전사 e1){
    System.out.println("마법으로 " + e1.name + " 공격하기");
  }
}

class 엘프{  // 활
  String name = "엘프";
  void 기본공격(전사 e1){
    System.out.println("엘프로 " + e1.name + " 공격하기");
  }
}

class 흑마법사{  // 마법
  String name = "흑마법사";
  void 기본공격(전사 e1){
    System.out.println("흑마법으로 " + e1.name + " 공격하기");
  }
}
→ 마법사, 엘프, 흑마법사가 더 늘어남.
→ 새로운 객체가 생길 때 마다 공격할 수 있는 메서드를 추가해 주어야 함.

public class OOPEx05{
  //static void attack(전사 u1){
  //  u1.기본공격();
  //}
  public static void main(String[] args){
    전사 u1 = new 전사();
    궁수 u2 = new 궁수();
    광전사 u3 = new 광전사();
```

```
        u1.기본공격(u2);
        u2.기본공격(u3);
        u3.기본공격(u1);
        //u1.기본공격2(u3);   // 함수의 이름을 달리 주면 외우기 힘들다.
    }
}
```

※ 오버로딩은 어느 정도 경우의 수가 제한되어 있다면 좋지만, 경우의 수가 많으면 힘들다....

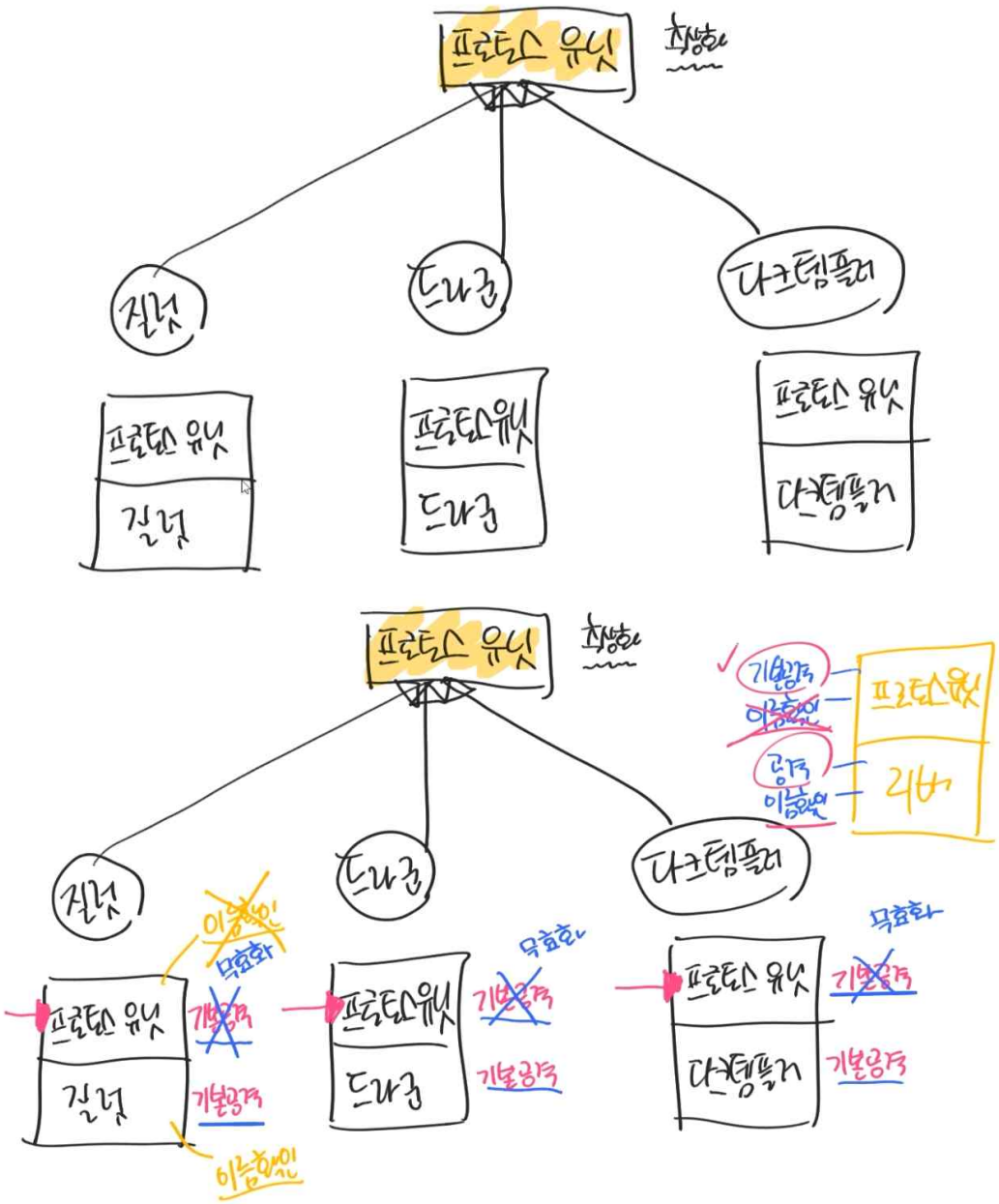
## 41장. 오버라이딩

▷ 스타크래프트로 ...

질럿 : 높은 계급에 오르지 못한 용사

드라군 : 육체를 기계 속에 집어넣어서 조정

다크템플러 : 은폐형 정예 돌격대

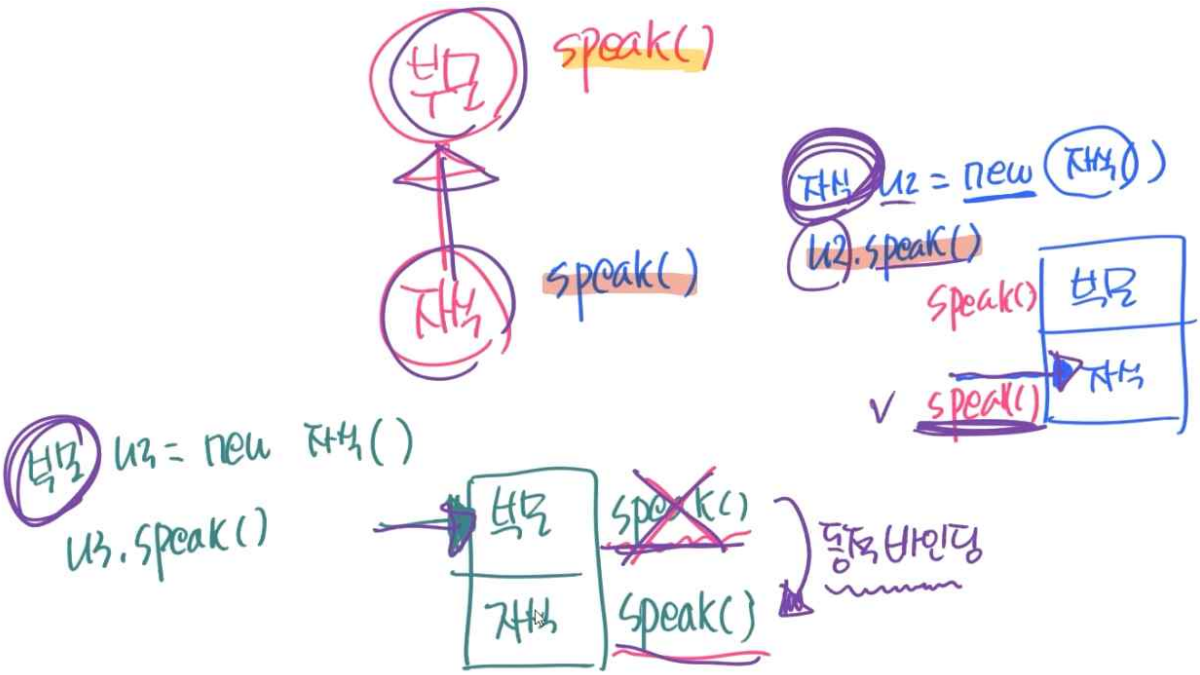



```
package ch05;

class 프로토스유닛{
   String name = "프로토스유닛";
   void 기본공격(프로토스유닛 e1){   // 무효화 됨
      System.out.println("프로토스유닛 메서드");
   }
   String 이름확인(){   // 무효화 됨.
      return "?";
   }
```

```
}

//class 질럿{
class 질럿 extends 프로토스유닛{
  String name = "질럿";
  // 오버라이드 = 부모의 메서드를 무효화 한다.
  void 기본공격(프로토스유닛 e1){
    System.out.println("질럿 메서드");
    //System.out.println(this.name + "이 " + e1.name + "을 공격합니다.");
    System.out.println(this.name + "이 " + e1.이름확인() + "을 공격합니다.");
  }
  String 이름확인(){
    return this.name;
  }
}
class 드라군 extend 프로토스유닛{
  String name = "드라군";
  void 기본공격(프로토스유닛 e1){
    System.out.println("드라군 메서드");
    System.out.println(this.name + "이 " + e1.이름확인() + "을 공격합니다.");
  }
  String 이름확인(){
    return this.name;
  }
}

class 다크템플러 extend 프로토스유닛{
  String name = "다크템플러";
  void 기본공격(프로토스유닛 e1){
    System.out.println("다크템플러 메서드");
    System.out.println(this.name + "이 " + e1.이름확인() + "을 공격합니다.");
  }
  String 이름확인(){  // 오버라이드 = 부모의 메서드를 무효화한다.
    return this.name;
  }
}
```

```
=========================================================
// 신입 → 리버라는 유닛을 하나 만들어봐(오버라이드 해서 만들어!)
// 팀장 → 프로토스유닛으로 상속(공격메서드, 이름을 확인하는 메서드가 필요해)
//       → name 이라는 변수 하나 만들어!! - 리버
// 누구를 공격하라는 거죠? → 다 공격해야 해... (프로토스 유닛)
// 테스트해봐 → 질럿으로 리버한번 공격해봐
// 팀장님 ? 가 뜨는데요?
// 아 그거 너 메서드 이름 머라고 적었어?  →  이름체크라고 했는데요?
// 리버가 질럿을 공격하게 해봐.
// 이 친구야 기본공격이라고 이름 좀 바꿔봐!!
// 팀장은 신입이 안물어보고 코딩을 하게 할 수 있는 방법이 뭘까?
=========================================================
class 리버 extends 프로토스유닛{
  String name = "리버";
  void 공격(프로토스유닛 e1){
    System.out.println(this.name + "이 " + e1.name + "을 공격합니다.");
  }
  //String 이름체크(){
  //  return name;
  //}
  String 이름확인(){
    return name;
  }
}

public class OOPEx06{
  public static void main(String[] args){
    //질럿 u1 = new 질럿();
    //드라군 u2 = new 드라군();
    //다크템플러 u3 = new 다크템플러();
    프로토스유닛 u1 = new 질럿();
    프로토스유닛 u2 = new 드라군();
    프로토스유닛 u3 = new 다크템플러();

    u1.기본공격(u2);
    u2.기본공격(u3);
    u3.기본공격(u1);
```

```
==========================================================
    프로토스유닛 u4 = new 리버();
    u1.기본공격(u4);        → 출력결과 :  질럿이 ?를 공격합니다.
    u4.기본공격(u1);
==========================================================
  }
}
```



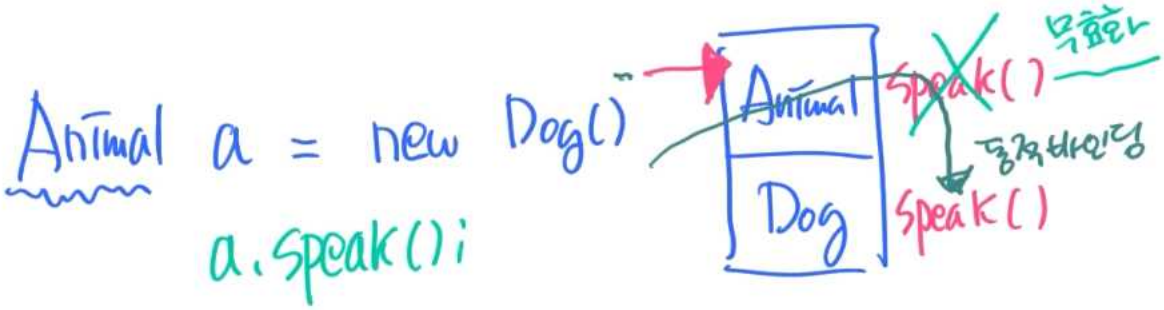
## 42장. 추상클래스

▷ 추상클래스 : 추상적인 것.

① new 할 수 없다. → 즉 인스턴스를 생성할 수 없다.

② 일반 메서드도 가질 수 있다.

③ 가구 같은 경우를 추상클래스라 한다.




```
package ch05;

//class Animal{
abstract class Animal{
   //void speak(){};
   abstract void speak(); // 추상메서드 몸체 {  } 가 없다. 어차피 무효화 될거니까..
   void hello(){
      System.out.println("!!!!");
   }
}

class Dog extends Animal{
   // 오버라이드 (Animal의 speak() 가 무효화된다.
   void speak(){
      System.out.println("멍멍");
   }
}
class Cat extends Animal{
   // 오버라이드 (Animal의 speak() 가 무효화된다.
   void speak(){
      System.out.println("야옹");
   }
}
```

```
=========================================================
// 신입 → Bird를 만들어봐(Animal 을 상속해서 만들어!)
// 출력은 "짹짹" 하도록 해!
// 팀장한테 물어볼 필요가 없다.
=========================================================
class Bird extends Animal(){  // Animal이 갖고 있는 메서드를 구현하라고 오류 뜸 – 강제성
  // 아래 코드가 자동 생성됨.
  // 추상메서드를 부모가 갖고 있으면 자식은 추상메서드를 무조건 구현해야 한다.
  @Override
  void speak(){
    System.out.println("짹짹");
  }
}

public class OOPEx07{
  public static void main(String[] args){
    Animal a1 = new Dog();
    Animal a2 = new Cat();
    a1.speak(); // 동적바인딩 된다. 왜? 부모의 메서드가 무효화되니까!!

    //Animal a3 = new Animal(); // abstract로 선언하면 new를 이용해 인스턴스로 생성 못함..오류.
=========================================================
    Animal a3 = new Bird();
    a3.speak();
=========================================================
  }
}
```

## 43장. 추상클래스 미완성 설계도

▷ 타입을 일치시켜서 미완성 설계도를 만든다.
▷ 일부 추상적인 메서드를 포함하기 때문에 미완성 설계도라 한다.
▷ 자식이 완성해야 하는 메서드가 존재한다는 의미이기도 하다.


```
package ch05;

abstract class 육식동물{
   void 걷기(){
      System.out.println("걷다!");
   }
   // 미완성 설계도
   abstract void 공격();
}

class 뱀 extends 육식동물{
   @Override
   void 공격(){
      System.out.println("독으로 공격");
   }
}
class 사자 extends 육식동물{
   @Override
   void 공격(){
      System.out.println("이빨로 공격");
   }
}

public class OOPEx08{
   public static void main(String[] args){
      육식동물 u1 = new 사자();
      육식동물 u2 = new 뱀();
```

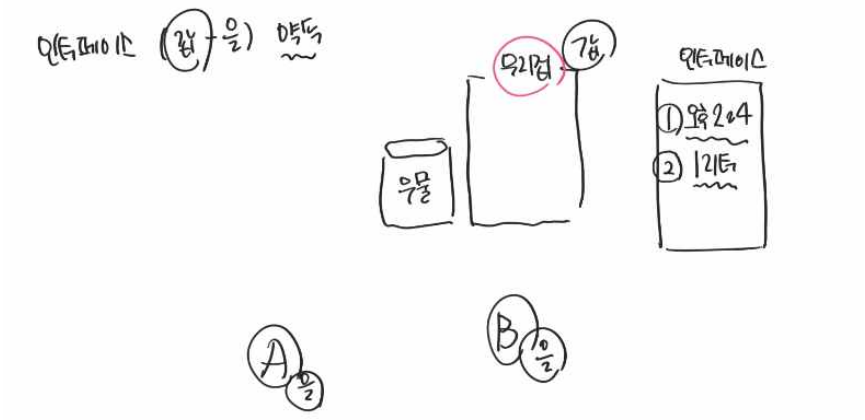
```
        u1.걷기();
        u1.공격();
        u2.걷기();
        u2.공격();
    }
}
```

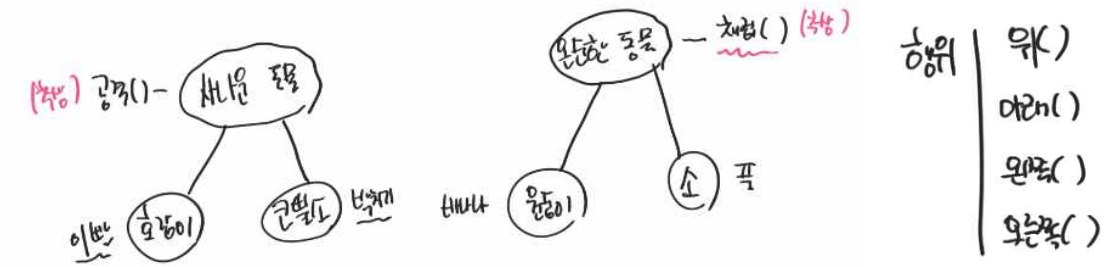
# 44장. 인터페이스

▷ 인터 : 교차로, 페이스 : 직면하다, 얼굴

▷ 인터페이스 : 만든 사람의 일방적인 약속이다. 즉, 행위에 대한 제약을 거는 것이다.

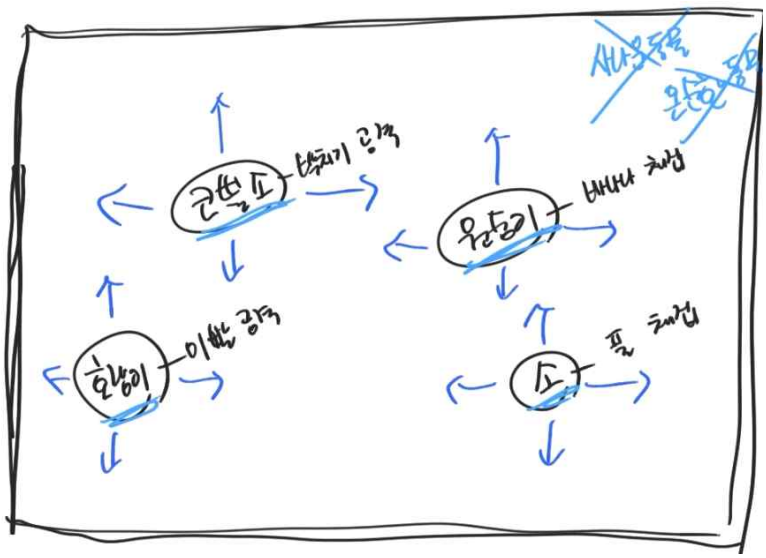
→ 인터체인지 진입 시 사용자의 선택보다는 만든 사람의 필요에 의한 것 밖에 선택할 수 없다.





▷ 네가지 행위를 할 수 있는 인터페이스를 만들어서 구현한다.

▷ 사나운 동물과 온순한 동물은 추상클래스이고, 움직임에 대한 제약은 인터페이스 이다.



▷ 위에 네가지 동물이 존재하는 게임 맵이 존재할 때, 이들의 행동을 제약하는 내용을 인터페이스로 구현한다.

## 45장. 인터페이스와 추상클래스의 차이

▷ 위의 내용을 코드로 정리

```
package ch05;
// 모든 동물들에게 행위의 제약을 주기 위해 Interface를 생성한다.
interface MoveAble{
  // 기본적으로 public abstract가 생략되어 있다.
  void 위();
  void 아래();
  void 왼쪽();
  void 오른쪽();
}
interface MoveAble2{  // 숨기 기능 추가
  // 기본적으로 public abstract가 생략되어 있다.
  void 위();
  void 아래();
  void 왼쪽();
  void 오른쪽();
  void 땅바닥으로숨기();
}

//abstract class 사나운동물{
abstract class 사나운동물 implements MoveAble{
  // 중요 : 인터페이스를 상속받은 추상클래스는 인터페이스 내에 있는 추상 메서드를
  // 구현하지 않아도 된다.
  // 일반 클래스는 추상메서드를 구현해야 한다.
  // 여기서 구현을 안하면 얘네들을 상속받는 클래스에서 구현해야 함 → 위임된다.
  abstract void 공격();  // 미완성 설계도
  @Override
  public void 왼쪽(){
    System.out.println("왼쪽으로 이동");
  }
  @Override
  public void 오른쪽(){
    System.out.println("오른쪽으로 이동");
  }
  @Override
  public void 위(){
    System.out.println("위로 이동");
  }
  @Override
```

```
    public void 아래(){
        System.out.println("아래로 이동");
    }
}
abstract class 온순한동물 implements MoveAble2{
    abstract void 채집();  // 미완성 설계도
    @Override
    public void 왼쪽(){
        System.out.println("왼쪽으로 이동");
    }
    @Override
    public void 오른쪽(){
        System.out.println("오른쪽으로 이동");
    }
    @Override
    public void 위(){
        System.out.println("위로 이동");
    }
    @Override
    public void 아래(){
        System.out.println("아래로 이동");
    }
    @Override
    public void 땅바닥으로숨기(){
        System.out.println("땅바닥으로 숨기");
    }
}

// 구현이 자식 클래스로 위임(위, 아래, 왼쪽, 오른쪽)
class 원숭이 extends 온순한동물{
    @Override  // Annotation = JVM이 실행시에 분석해서 처리 → JVM의 힌트
    // 즉, 컴파일시에 Override가 있으므로 부모클래스에서 채집 메서드를 찾고 없으면 오류 반환
    void 채집(){
        System.out.println("바나나 채집");
    }
    //@Override
    //public void 왼쪽(){
    //}
    //@Override
    //public void 오른쪽(){
```

```
  //}
  //@Override
  //public void 위(){
  //}
  //@Override
  //public void 아래(){
  //}
  // 각각의 상속받은 클래스에서 모든 인터페이서 내의 추상메서드를 구현하기에는 양이 많아지므
  // 그냥 사나운동물, 온순한동물 클래스에서 인터페이스를 구현하는 방법을 사용한다.
}

class 소 extends 온순한동물{
  @Override
  void 채집(){
    System.out.println("풀 채집");
  }
}
class 호랑이 extends 사나운동물{
  @Override
  void 공격(){
    System.out.println("이빨로 공격");
  }
}
class 코뿔소 extends 사나운동물{
  @Override
  void 공격(){
    System.out.println("몸통 박치기로 공격");
  }
}

============================================================
// 신입 → 말을 만들어봐 → 얜 온순한 동물이야.
============================================================
class 말 extends 온순한동물(){
  @Override
  void 채집(){
    System.out.println("풀을 먹다.");
  }
}
```

```
public class OOPEx09{
  //void 조이스틱(소 u1){  // 한꺼번에 움직이고 싶어서 만듦
  static void 조이스틱(온순한동물 u1){  // 한꺼번에 움직이고 싶어서 만듦
    u1.채집();
    u1.땅바닥으로숨기();
    u1.위();
    u1.아래();
    u1.오른쪽();
    u1.왼쪽();
    System.out.println("============================");
  }
  static void 조이스틱(사나운동물 u1){  // 한꺼번에 움직이고 싶어서 만듦
    u1.공격();
    u1.위();
    u1.아래();
    u1.오른쪽();
    u1.왼쪽();
    System.out.println("============================");
  }

  public static void main(String[] args){
    소 u1 = new 소();
    //u1.채집();
    //u1.땅바닥숨기();
    조이스틱(u1);
    원숭이 u2 = new 원숭이();
    조이스틱(u2);

    호랑이 u3 = new 호랑이();
    조이스틱(u3);   // 이때 오버로딩이 필요함.
=========================================================
    말 u5 = new 말();
    조이스틱(u5);
=========================================================
  }
}
```

**추가문제>**

**① 사나운 동물에 냄새맡기() 기능 추가 → 공통적으로 모두 "코로 냄새맡기" 출력**

**② 하이에나를 사나운 동물에 추가하기 → 이빨로 공격**

**문제1> 추상클래스 및 인터페이스를 이용해서 여러 가지 도형의 넓이를 구합니다.**

▷ 조건은 아래와 같습니다.

① 삼각형(Triangle), 사각형(Rectangle), 원(Circle)의 넓이를 구하는 프로그램을 작성하세요.

② 추상클래스 구현
- 모든 도형의 면적은 area 메서드를 사용하며,
- 데이터 타입은 double을 사용합니다.

③ 인터페이스 구현
- 가로(width), 세로(height)의 값은 정수치이며 인터페이스로 값을 할당할 수 있도록 구현합니다.
- 단, 원(Circle) 클래스를 구현할 때 위의 내용은 그냥 return 처리 합니다.

④ 도형의 면적을 구한 후 출력하는 메서드는 static으로 작성하며,

삼각형의 넓이는 25.0입니다.
사각형의 넓이는 50.0입니다.
원의 넓이는 78.53981633974483입니다. 와 같이 출력되도록 합니다.

**문제2> 추상클래스 및 인터페이스를 이용해서 여러 가지 도형의 넓이를 구합니다.**

▷ 길동이 집에는 티비가 두개가 있다. 그런데 정말 불편한 점이 있었다.
삼성 티비는 초록 버튼을 클릭하면 전원이 켜지고 파랑 버튼을 클릭하면 전원이 꺼졌다.
엘지 티비는 초록 버튼을 클릭하면 전원이 켜지고 빨간 버튼을 클릭하면 전원이 꺼졌다.
두 회사의 차이 때문에 안방에 있는 삼성 티비를 끌 대 파랑 버튼을 클릭했고, 거실에 있는 엘지 티비는 빨간 버튼을 클릭해서 티비를 껏다.
너무 너무 혼란 스러웠다.

길동이는 이 문제를 해결하고자 삼성과 엘지에 동시에 적용되는 리모콘을 직접 개발하기로 했다.

▶ 초록 버튼 : 전원 켜짐
▶ 빨간 버튼 : 전원 꺼짐

이렇게 통일하기로 마음먹었고 행위에 대한 제약을 주기 위해 인터페이스를 만들었다.
인터페이스의 이름은 RemoconAble 이라고 지었다.
인터페이스에서 사용할 메서드 이름은 greenButton()과 redButton(); 으로 결정하였다.

이제 길동이는 RemoconAble 인터페이스를 토대로 리모콘을 만들 수 있었고 new를 이용해서 필요한 리모콘을 각각 1개씩 회사별로 만들어 냈다. 그리고 테스트를 실행하였다.

LgTV와 SamsungTV가 작동하도록 해당 기능을 구현하시오.

> 출력 결과는 다음과 같습니다.

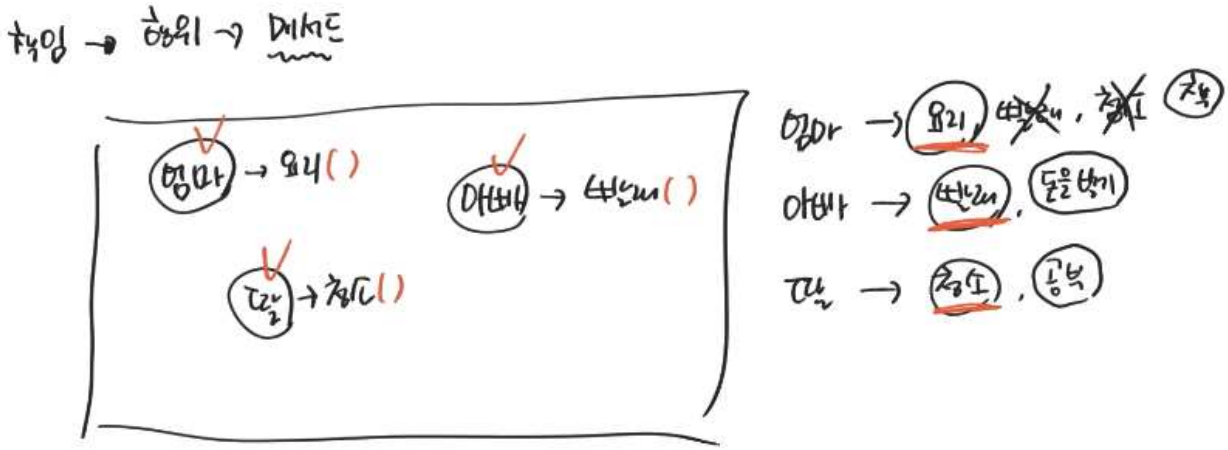
```
삼성 티비가 켜졌습니다.
삼성 티비가 꺼졌습니다.
LG 티비가 켜졌습니다.
LG 티비가 꺼졌습니다.
```

# 46장. SRP와 DIP 개념

## 1. SRP

▷ SRP : Single Responsibility Principal(단일책임원칙)

▷ 책임 → 행위 → 메서드



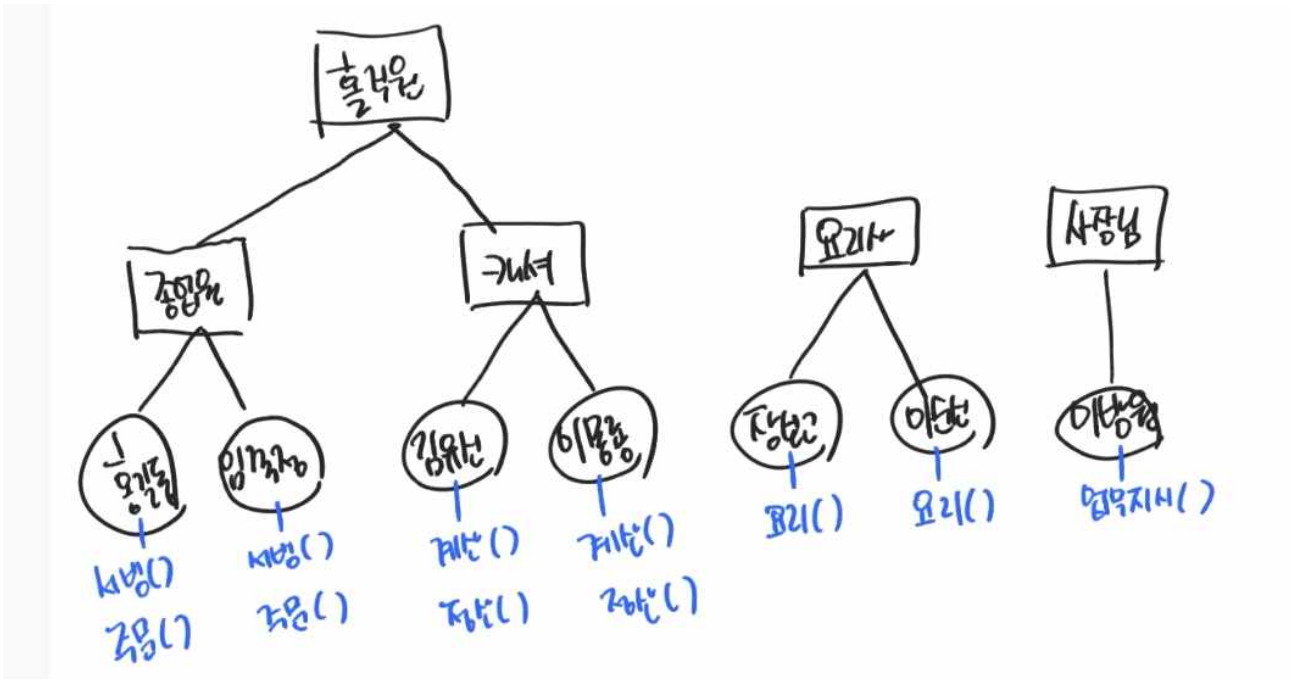

**2. DIP(Dependency Inversion Principal) : 의존성 역전 원칙**

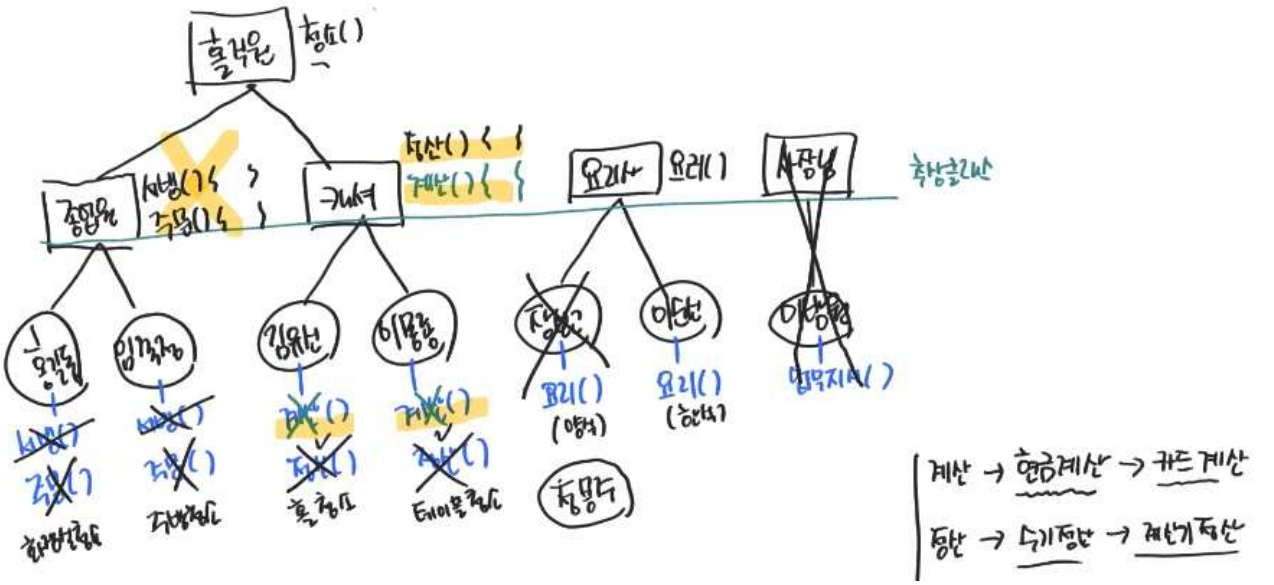
**▶ DIP 원칙이란 객체에서 어떤 class를 참조해서 사용해야하는 상황이 생긴다면, 그 class를 직접 참조하는 것이 아니라 그 대상의 상위요소(추상클래스 or 인터페이스)로 참조하라는 원칙.**

**▶ SRP와 DIP 규칙을 지켜 프로그래밍을 하면 추후 수정이 용이해 진다.**

## 47장. SRP와 DIP 실습

▷ 실습 테스트




▷ 종업원, 캐셔, 요리사 라인은 추상클래스 라인이다.

▷ 우측과 같은 변경사항이 생기면 수정이 번거로우니 원래의 코드를 상위인 추상클래스 라인으로 올린다.

▷ 인터페이스로 제약 관계 생성 : 홀직원만 손님과 대화할 수 있는 talk() Method를 가짐.

```
package ch05;

interface CanAble{
   public abstract void talk();
}

abstract class 홀직원 implements CanAble{
   abstract void 청소();

   @Override
   public void talk(){
      System.out.println("손님과 대화하기");
   }
}

abstract class 종업원 extends 홀직원{
   void 서빙하기(){
      System.out.println("서빙하기");
   }

=============================================================
   // 변경 시나리오 3. 주문받기 → 키오스크설치 → 종업원이 주문받을 필요 없어짐.
   // → 주문받기 method를 삭제 또는 주석처리하기만 하면 됨.
=============================================================
   //void 주문받기(){
   //   System.out.println("주문받기");
   //}
}
abstract class 캐셔 extends 홀직원{
================================================
   // 변경 시나리오 2. 수기 정산하기 → 계산기 정산하기로 변경
================================================
   void 정산하기(){
      //System.out.println("수기 정산하기");
================================================
      System.out.println("계산기 정산하기");
================================================
   }
================================================
   // 변경 시나리오 1. 현금계산 → 카드계산으로 변경
```

```
================================================
  void 계산받기(){
    //System.out.println("현금 계산하기");
================================================
    System.out.println("카드 계산하기");
================================================
  }
}

abstract class 요리사{
  abstract void 요리(){
  }
}

class 홍길동 extends 종업원{
  // 홍길동과 임꺽정은 요리사인 장보고, 이순신을 의지해야 되는데 직접 의지하게 되면 나중에
  // 특정 요리사 퇴사시 문제가 생김. 그래서 의존은 상위 추상클래스인 요리사 클래스를
  // 의존해야함. → 의존성 역전원칙

  요리사 j;

  @Override
  void 청소(){
    System.out.println("화장실 청소");
  }
}
class 임꺽정 extends 종업원{
  요리사 j;
  @Override
  void 청소(){
    System.out.println("주방 청소");
  }
}
class 김유신 extends 캐셔{
  @Override
  void 청소(){
    System.out.println("홀 청소");
  }
}
class 이몽룡 extends 캐셔{
```

```
  @Override
  void 청소(){
    System.out.println("테이블 청소");
  }
}
=====================================================
  // 변경 시나리오 4. 장보고 퇴사 → 정몽주 입사
  // → 홍길동과 이몽룡이 각각의 장보고, 이순신 요리사를 의존하고 있지
  // 않고 요리사 추상클래스를 의존하고 있으며로 삭제해도 관계없음.
=====================================================
//class 장보고 extends 요리사{
//   @Override
//   void 요리(){
//      System.out.println("양식 만들기");
//   }
//}
=====================================================
class 정몽주 extends 요리사{
  @Override
  void 요리(){
    System.out.println("양식 만들기");
  }
}
=====================================================

class 이순신 extends 요리사{
  @Override
  void 요리(){
    System.out.println("한식 만들기");
  }
}
public class OOPEx10{
  public static void main(String[] args){
  }
}
```

## ▶ Chapter 5 연습문제

▷ 길동이 집에는 티비가 두개가 있다. 그런데 정말 불편한 점이 있었다.
삼성 티비는 초록 버튼을 클릭하면 전원이 켜지고 파랑 버튼을 클릭하면 전원이 꺼졌다.
엘지 티비는 초록 버튼을 클릭하면 전원이 켜지고 빨간 버튼을 클릭하면 전원이 꺼졌다.
두 회사의 차이 때문에 안방에 있는 삼성 티비를 끌 대 파랑 버튼을 클릭했고, 거실에 있는 엘지 티비는 빨간 버튼을 클릭해서 티비를 껏다.
너무 너무 혼란 스러웠다.

길동이는 이 문제를 해결하고자 삼성과 엘지에 동시에 적용되는 리모콘을 직접 개발하기로 했다.

▶ 초록 버튼 : 전원 켜짐
▶ 빨간 버튼 : 전원 꺼짐

이렇게 통일하기로 마음먹었고 행위에 대한 제약을 주기 위해 인터페이스를 만들었다.
인터페이스의 이름은 RemoconAble 이라고 지었다.

```
public void 초록버튼();
public void 빨간버튼();
```

이제 길동이는 RemoconAble 인터페이스를 토대로 리모콘을 만들 수 있었고 new를 이용해서 필요한 리모콘을 각 2개씩 회사별로 만들어 냈다.

해당 기능을 구현하시오.

```
package ch05;

interface RemoconAble{
   public void 초록버튼();
   public void 빨간버튼();
}

class Samsung implements RemoconAble{
   @Override
   public void 초록버튼(){
      System.out.println("전원이 켜졌습니다.");
   }
   @Override
   public void 빨간버튼(){
      System.out.println("전원이 꺼졌습니다.");
   }
}
```

```
class LG implements RemoconAble{
   @Override
   public void 초록버튼(){
      System.out.println("전원이 켜졌습니다.");
   }
   @Override
   public void 빨간버튼(){
      System.out.println("전원이 꺼졌습니다.");
   }
}

public class OOPExampleEx01{
   public static void main(String[] args){
   // 삼성 리모콘 2개 만들기
   Samsung s1 = new Samsung();
   Samsung s2 = new Samsung();
   // LG 리모콘 2개 만들기
   LG g1 = new LG();
   LG g2 = new LG();
   }
}
```