1. 상속과 합성

객체지향 프로그래밍에서 코드 **재사용** 방법으로 크게 **상속**과 **합성**이 있다.

〈상속(inheritance)〉

```
class Father { int height = 180; }                  //아버지 키 180
class Son extends Father                            //상속
{                                  재사용
  public void how_tall() { System.out.println(height + 10); }  //출력 190
}
public class Test {
  public static void main(String args[]) {
    Son s = new Son();    //객체 생성
    s.how_tall();         //메서드 호출
  }
}
```

〈합성(composition)〉

```
class Father { int height = 180; }        //아버지 키 180
class Son
{                                  재사용
  private Father f = new Father();                          //아버지 객체 포함 (합성)
  public void how_tall() { System.out.println(f.height + 10); }  //출력 190
}
public class Test {
  public static void main(String args[]) {
    Son s = new Son();    //객체 생성
    s.how_tall();         //메서드 호출
  }
}
```

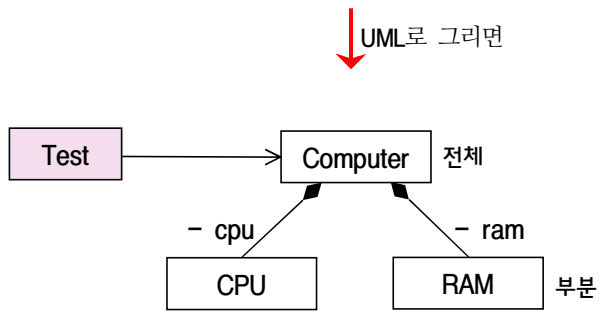
// 상속과 합성 비교

	공통점	바인딩	속도	결합	유지보수	관계	적용
상속	재사용	컴파일시간	빠르다	강결합	어렵다	is-a	꼭 필요한 곳
합성	재사용	실행시간	느리다	약결합	쉽다	has-a	권장 사항

2 http://cafe.daum.net/pass365(홍재연)

예제	컴퓨터가 객체 CPU와 RAM을 포함하고 있는 구조

```
class CPU { int c = 3; }          //CPU 속도는 3GHz
class RAM { int r = 4; }          //RAM 용량은 4TB
class Computer
{
  private CPU cpu;                //클래스 Computer와 CPU 사이의 연관
  private RAM ram;                //클래스 Computer와 RAM 사이의 연관
  public Computer()               //생성자(합성관계) - 핵심부분!
  {
    this.cpu = new CPU();         //Computer 내에서 직접 CPU의 객체 생성(합성)
    this.ram = new RAM();         //Computer 내에서 직접 RAM의 객체 생성(합성)
  }
  public void dispCPU() { System.out.println(cpu.c + "GHz"); }   //출력 3GHz
  public void dispRAM() { System.out.println(ram.r + "TB"); }    //출력 4TB
}
public class Test
{
  public static void main(String args[])
  {
    Computer com = new Computer();                  //기본생성자 호출
    com.dispCPU();                                  //메서드 dispCPU() 호출
    com.dispRAM();                                  //메서드 dispRAM() 호출
  }
}
```



탐구

피터 코드의 상속 규칙

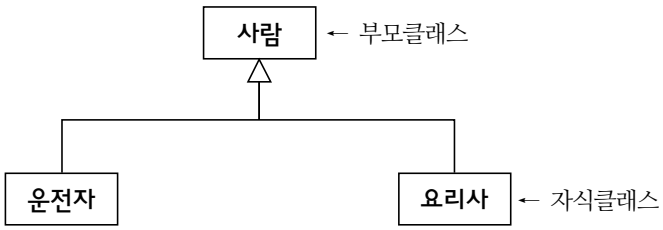
피터 코드의 상속 규칙은 상속 오용을 막기 위한 제한 규칙이다.(5가지)
피터 코드의 상속 규칙은 상속을 올바르게 사용할 수 있도록 판단할 수 있는 규칙이다.

〈피터 코드의 상속 규칙〉

① 자식클래스와 부모클래스 사이는 **역할 수행 관계**가 아니어야 한다.
② 한 자식클래스의 객체는 다른 자식클래스의 **객체로 변환**할 필요가 절대 없어야 한다.
③ 자식클래스가 부모클래스의 책임을 무시하거나 재정의하지 않고 **확장 수행**해야 한다.
④ 자식클래스가 단지 부모클래스의 **일부 기능을 재사용**할 목적으로는 상속하지 않아야 한다.
⑤ 자식클래스가 역할, 트랜잭션 등을 **세분화(부모에서 자식을 추출하는 과정)**해야 한다.

- 피터 코드의 5가지 상속 규칙을 모두 만족할 때, 상속을 적용해야 한다.
- 즉, 어느 한 규칙이라도 만족하지 않으면 상속을 적용하지 않아야 한다.

〈피터 코드의 상속 규칙을 이용한 상속 적용 여부〉



규칙 1	• 운전자와 요리사는 각각 어떤 순간에는 **사람의 역할을 수행**한다. • 피터 코드의 상속 규칙 위배 (운전자와 요리사는 사람을 상속받으면 안 됨)
규칙 2	• 요리사가 자신이 일하는 회사로 출퇴근하는 시점에는 운전자가 된다.**(객체 변환 필요)** • 피터 코드의 상속 규칙 위배 (운전자와 요리사는 사람을 상속받으면 안 됨)
규칙 3	• 사람, 운전자, 요리사 클래스에 대한 속성과 연산 정보가 없다.(분석 불가)
규칙 4	• 기능만을 재사용할 목적으로 상속 관계를 표현하지는 않았으므로 규칙 준수 • 피터 코드의 상속 규칙 준수
규칙 5	• 부모클래스가 역할, 트랜잭션 등을 표현하지 않았다. • 피터 코드의 상속 규칙 위배 (운전자와 요리사는 사람을 상속받으면 안 됨)

• **결론** : 운전자와 요리사는 사람을 상속받으면 안 된다.**(상속 대안으로 위임을 사용해야 한다)**