

9

부품구조: 다형성과 인터페이스를 활용한 프로그래밍

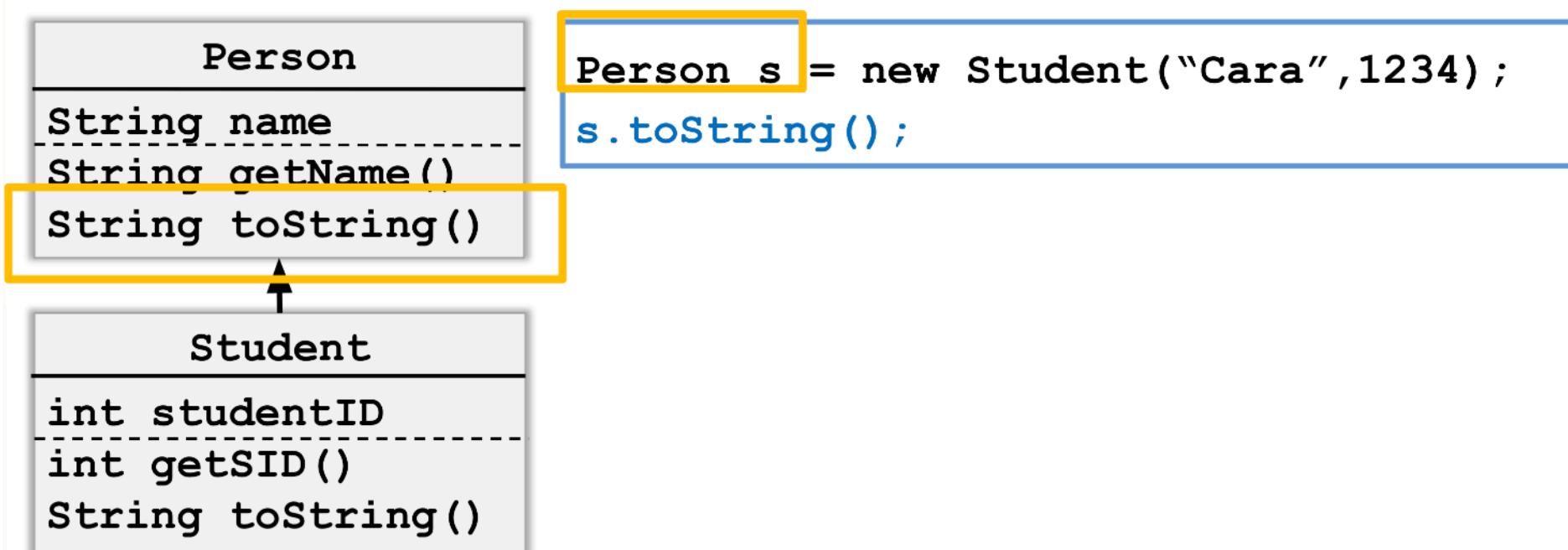
다형성 (Polymorphism)

- 여러 타입의 값들을 다루는 하나의 인터페이스를 제공하는 것

```
Person p = new Person("Tim");
System.out.println(p.toString()); // Tim
Person p = new Student("Cara");
System.out.println(p.toString()); // Student: Cara
Person p = new Faculty("Mia");
System.out.println(p.toString()); // Faculty: Mia
```

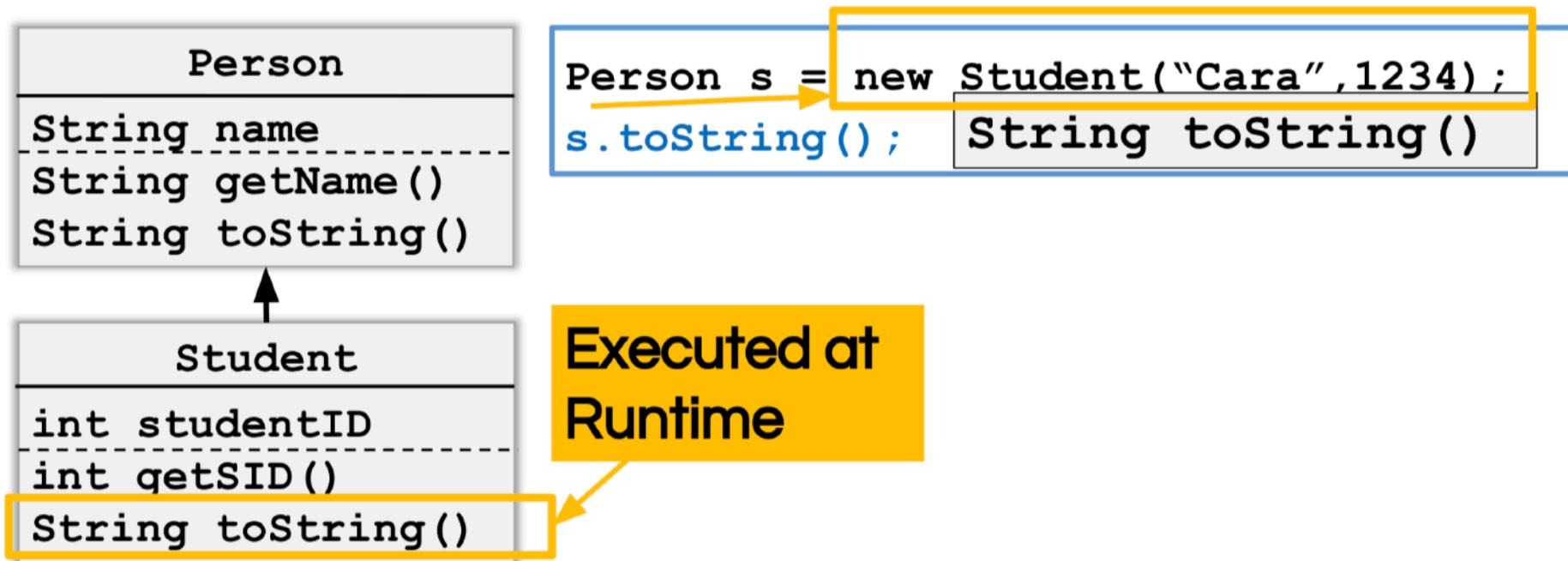
컴파일 시 규칙

- 컴파일러는 변수의 선언된 타입 (reference type) 만 안다.
- 메소드 호출에 대해 선언된 타입 클래스 내부에 해당 메소드가 존재하는지 체크

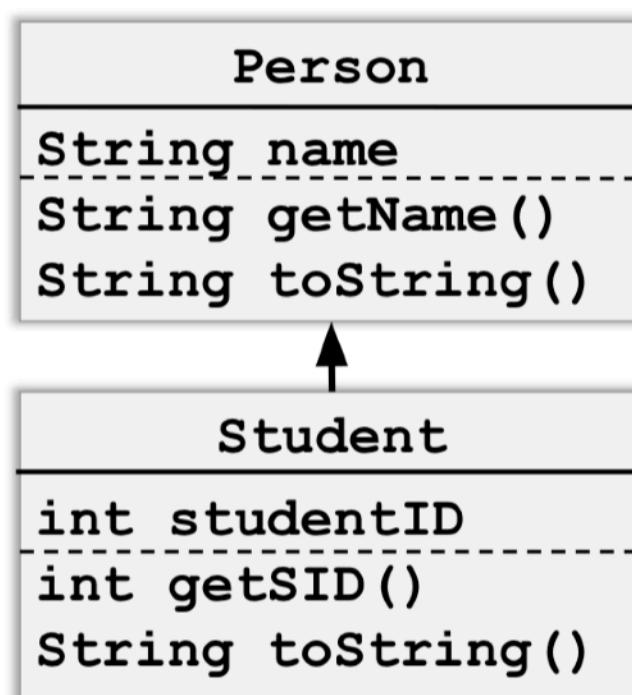


실행 시 규칙

- 호출된 메소드를 찾기 위해 객체의 정확한 타입 클래스 내부를 탐색한다.



다음은 컴파일 규칙에 위반

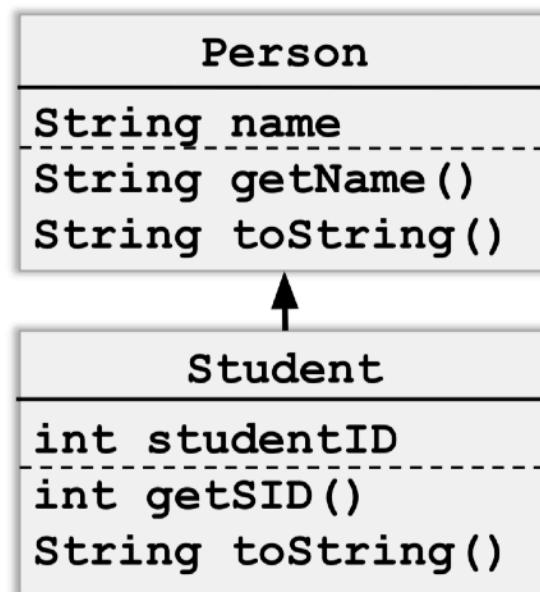


```
Person s = new Student("Cara", 1234);
s.getSID();
```

Compile Time
Error!

타입 변환 (Type Casting)

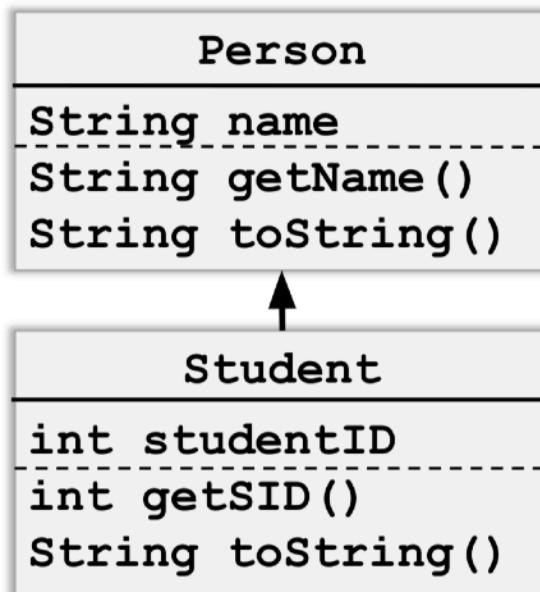
- 무시적 변환 (예: int 형에서 double 형으로)
 - Superclass super = new Subclass();
- 명시적 변환 (예: double 형에서 int 형으로)
 - Subclass sub = (Subclass)super;
 - (주의 필요) 컴파일러는 개발자가 옳게 했다고 믿음



```

Person s = new Student("Cara", 1234);
s.getSID();
( (Student)s ).getSID();
  
```

This works!



```

Person s = new Person("Tim");
( (Student)s ).getSID();
  
```

Runtime Error!
java.lang.ClassCastException:
 From Person to Student

실행 시간 타입 검사

- instanceof : is-a 관계가 성립되는지 실행 시 체크

```
if (s instanceof Student)
{
    // only executes if s is-a Student at runtime
    (Student) s).getSID();
}
```

```

public class Person {
    private String name;
    public Person(String name)
    {this.name = name;}
    public boolean isAsleep(int hr)
    { return 22 < hr || 7 > hr; }
    public String toString() { return name; }
    public void status( int hr ) {
        if (this.isAsleep(hr))
            System.out.println( "취침중: " + this );
        else
            System.out.println( "깼음: " + this ); }
}
public class Student extends Person
{
    public Student(String name) {super(name);}
    public boolean isAsleep( int hr ) // override
    { return 2 < hr && 8 > hr; }
}

```

○ 실행결과?

```

Person p;
p = new Student("지윤");
p.status(1);

```

- A. “깼음: 지윤”
- B. “취침중: 지윤”
- C. 실행 오류
- D. 컴파일 오류

```

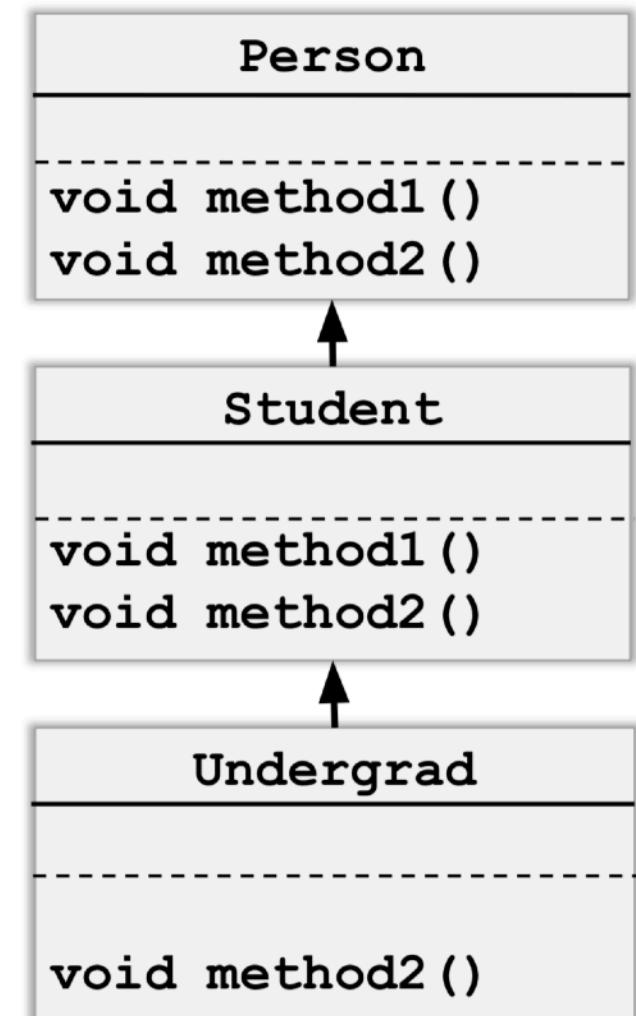
public class Person {
    public void method1() {
        System.out.print("Person 1 ");
    }
    public void method2() {
        System.out.print("Person 2 ");
    }
}
public class Student extends Person {
    public void method1() {
        System.out.print("Student 1 ");
        super.method1();
        method2();
    }
    public void method2() {
        System.out.print("Student 2 ");
    }
}
public class Undergrad extends Student {
    public void method2() {
        System.out.print("Undergrad 2 ");
    }
}

```

```

Person u = new Undergrad();
u.method1();

```



```
public class Person {  
    public void method1() {  
        System.out.print("Person 1 ");  
    }  
    public void method2() {  
        System.out.print("Person 2 ");  
    }  
}  
  
public class Student extends Person {  
    public void method1() {  
        System.out.print("Student 1 ");  
        super.method1();  
        method2();  
    }  
    public void method2() {  
        System.out.print("Student 2 ");  
    }  
}  
  
public class Undergrad extends Student {  
    public void method2() {  
        System.out.print("Undergrad 2 ");  
    }  
}
```

```
Person u = new Undergrad();  
u.method1();
```

```
public class Person {  
    public void method1 () {  
        System.out.print("Person 1 ");  
    }  
    public void method2 () {  
        System.out.print("Person 2 ");  
    }  
}  
  
public class Student extends Person {  
    public void method1 () {  
        System.out.print("Student 1 ");  
        super.method1 ();  
        method2 ();  
    }  
    public void method2 () {  
        System.out.print("Student 2 ");  
    }  
}  
  
public class Undergrad extends Student {  
    public void method2 () {  
        System.out.print("Undergrad 2 ");  
    }  
}
```

```
Person u = new Undergrad();  
u.method1();
```

```
public class Person {  
    public void method1() {  
        System.out.print("Person 1 ");  
    }  
    public void method2() {  
        System.out.print("Person 2 ");  
    }  
}  
  
public class Student extends Person {  
    public void method1() {  
        System.out.print("Student 1 ");  
        super.method1();  
        method2();  
    }  
    public void method2() {  
        System.out.print("Student 2 ");  
    }  
}  
  
public class Undergrad extends Student {  
    public void method2() {  
        System.out.print("Undergrad 2 ");  
    }  
}
```

```
Person u = new Undergrad();  
u.method1();
```

```

public class Person {
    public void method1() {
        System.out.print("Person 1 ");
    }
    public void method2() {
        System.out.print("Person 2 ");
    }
}
public class Student extends Person {
    public void method1() {
        System.out.print("Student 1 ");
        super.method1();
        method2();
    }
    public void method2() {
        System.out.print("Student 2 ");
    }
}
public class Undergrad extends Student {
    public void method2() {
        System.out.print("Undergrad 2 ");
    }
}

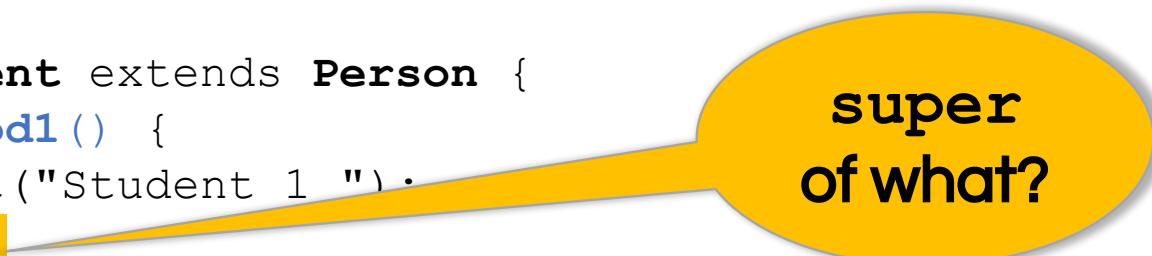
```

```

Person u = new Undergrad();
u.method1();

```

Output so far:
Student 1



**super
of what?**

```

public class Person {
    public void method1() {
        System.out.print("Person 1 ");
    }
    public void method2() {
        System.out.print("Person 2 ");
    }
}
public class Student extends Person {
    public void method1() {
        System.out.print("Student 1 ");
        super.method1();
        method2();
    }
    public void method2() {
        System.out.print("Student 2 ");
    }
}
public class Undergrad extends Student {
    public void method2() {
        System.out.print("Undergrad 2 ");
    }
}

```

```

Person u = new Undergrad();
u.method1();

```

Output so far:
Student 1

Static binding!

super 는 부모 클래스를 가리킴

```
public class Person {  
    public void method1() {  
        System.out.print("Person 1 ");  
    }  
    public void method2() {  
        System.out.print("Person 2 ");  
    }  
}  
  
public class Student extends Person {  
    public void method1() {  
        System.out.print("Student 1 ");  
        super.method1();  
        method2();  
    }  
    public void method2() {  
        System.out.print("Student 2 ");  
    }  
}  
  
public class Undergrad extends Student {  
    public void method2() {  
        System.out.print("Undergrad 2 ");  
    }  
}
```

```
Person u = new Undergrad();  
u.method1();
```

Output so far:

Student 1 Person 1

```
public class Person {  
    public void method1 () {  
        System.out.print("Person 1 ");  
    }  
    public void method2 () {  
        System.out.print("Person 2 ");  
    }  
}  
  
public class Student extends Person {  
    public void method1 () {  
        System.out.print("Student 1 ");  
        super.method1();  
        method2();  
    }  
    public void method2 () {  
        System.out.print("Student 2 ");  
    }  
}  
  
public class Undergrad extends Student {  
    public void method2 () {  
        System.out.print("Undergrad 2 ");  
    }  
}
```

```
Person u = new Undergrad();  
u.method1();
```

Output so far:
Student 1 Person 1

which
method2 () ?

```

public class Person {
    public void method1() {
        System.out.print("Person 1 ");
    }
    public void method2() {
        System.out.print("Person 2 ");
    }
}
public class Student extends Person {
    public void method1() {
        System.out.print("Student 1 ");
        super.method1();
        this.method2();
    }
    public void method2() {
        System.out.print("Student 2 ");
    }
}
public class Undergrad extends Student {
    public void method2() {
        System.out.print("Undergrad 2 ");
    }
}

```

```

Person u = new Undergrad();
.method1();

```

Output so far:
Student 1 Person 1

**type of object
at runtime**

this 는 현재 클래스의 인스턴스를 가리킴

Dynamic binding!

Undergrad extends **Student**

```

public class Person {
    public void method1() {
        System.out.print("Person 1 ");
    }
    public void method2() {
        System.out.print("Person 2 ");
    }
}
public class Student extends Person {
    public void method1() {
        System.out.print("Student 1 ");
        super.method1();
        this.method2();
    }
    public void method2() {
        System.out.print("Student 2 ");
    }
}
public class Undergrad extends Student {
    public void method2() {
        System.out.print("Undergrad 2 ");
    }
}

```

```

Person u = new Undergrad();
u.method1();

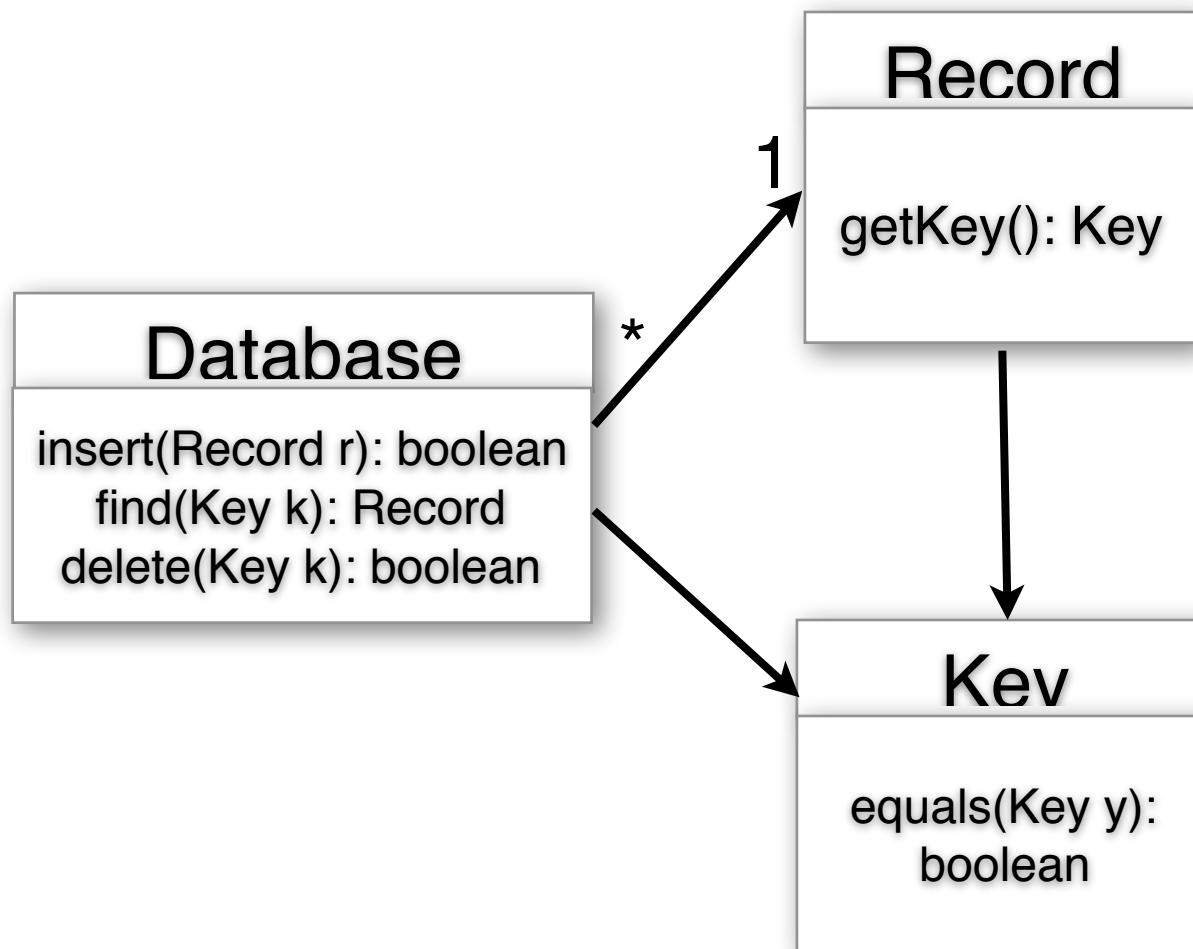
```

Final output:

Student 1 Person 1 Undergrad 2

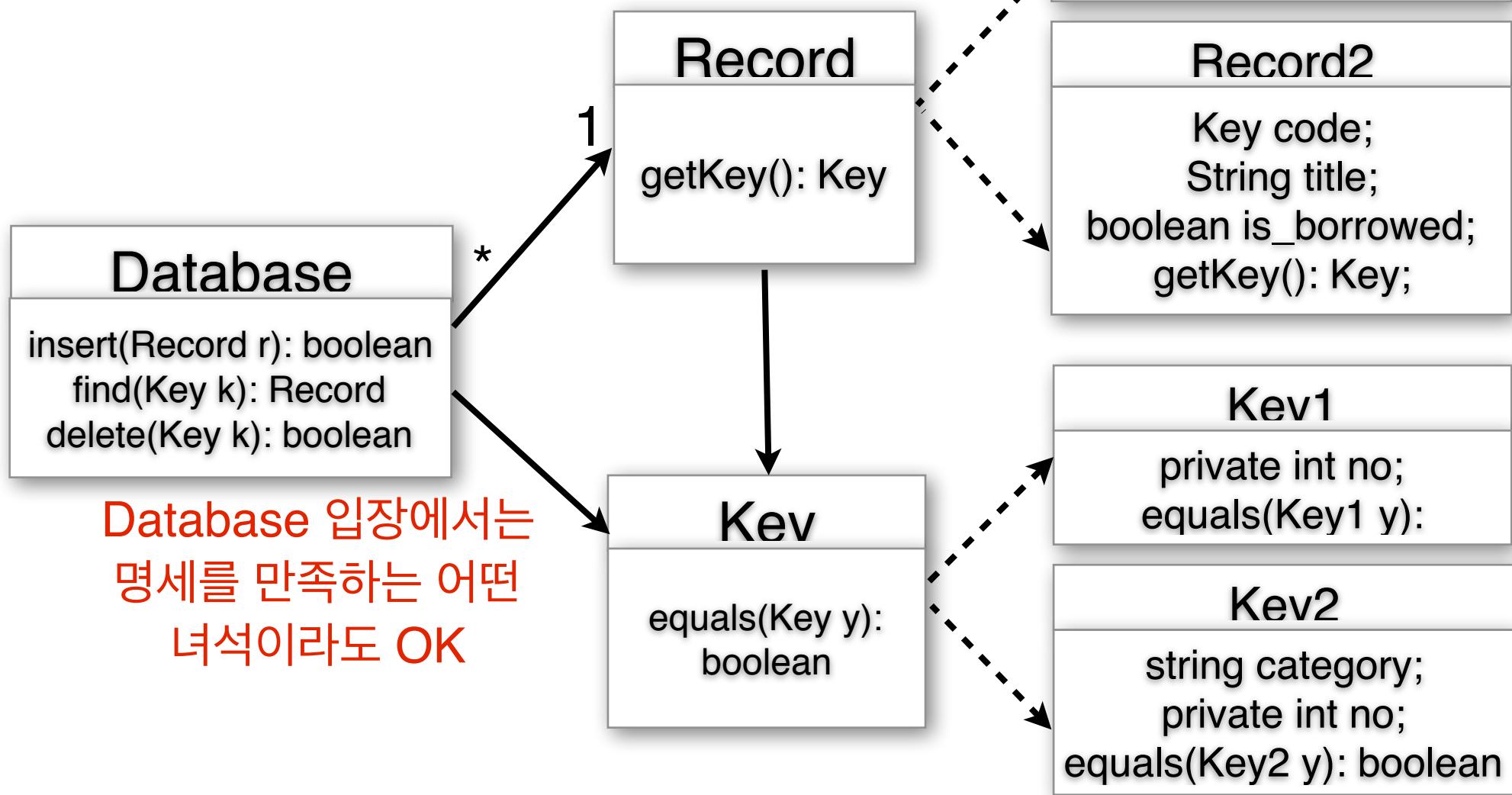
추상클래스와 인터페이스 (Abstract Classes and Interfaces)

데이터베이스 다시보기



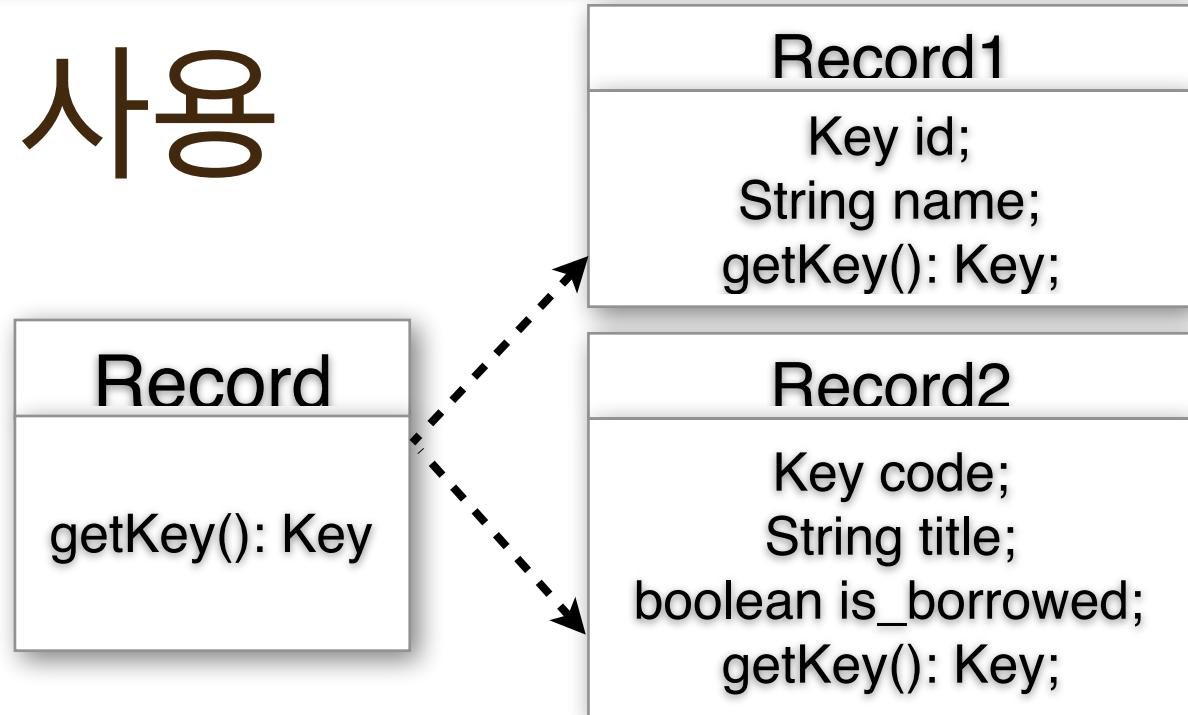
Record와 Key를 구현하지 않고도 Database를 구현할 수 있었다. 어떻게?

데이터베이스 사용



데이터베이스 사용

Java에서는 이러한 명세를 인터페이스(interface)라고 한다.
인터페이스는 구현이 없다.



Java에서는 이러한 클래스를 Record 인터페이스를 구현했다고 한다.
구현은 인터페이스에 명시된 모든 필드, 메소드를 타입에 맞게 제공해야 한다.

인터페이스 정의

- 인터페이스 정의
 - interface <이름> { <정의나열> }
 - 필드 정의는 기존과 동일, 단, 초기화 금지
 - 메소드 정의는 구현 없이 타입과 이름만 써 준다.
 - 예, 인수이름은 상관 없으나 중복되지 않도록
 - interface Key { boolean equals(Key y); }
 - interface Record { Key getKey(); }

인터페이스 또한 하나의 타입

- 클래스와 동일하게 타입으로 사용할 수 있다.
 - 단, new는 사용할 수 없다. 구현이 없으므로.
- 예,
 - 변수 선언, Key k;
 - 인수 전달, void method (Key k) { ... }
 - 반환, Key getKey() { ... }

인터페이스 구현

- 클래스를 정의할 때 뒤에 `implements <인터페이스>` 를 붙여 주면 그 인터페이스를 구현한다는 뜻이다.

```
class IntegerKey implements Key {  
    private int key;  
    public IntegerKey(int i) { key = i; }  
    public int getInt() { return key; }  
  
    boolean equals(Key k) { Key에 명세된 대로 equals 메소드를 작성  
        return key == ((IntegerKey)k).getInt();  
    }  
}
```

인터페이스 구현

- 여러 클래스를 같은 인터페이스로 구현할 수 있다.

```
class CodeKey implements Key {  
    private String category;  
    private int code;  
    public CodeKey(String s, int i) { category=s; code=i; }  
    public String getCategory() { return category; }  
    public int getCode() { return code; }  
  
    boolean equals(Key k) {  
        CodeKey ck = (CodeKey) k;  
        return category.equals(ck.getCategory()) &&  
            code == ck.getInt();  
    }  
}
```

Key에 명세된 대로 equals 메소드를 작성

같은 인터페이스를 구현한 객체는 혼용 가능

- IntegerKey 객체, CodeKey 객체 모두 Key 객체이다.

```
class findLocationDatabase {  
    ...  
    private int findLocation(Key k)  
    {  
        for (int i=0; i<base.length; i++)  
            if(base[i] != null && k.equals(base[i].getKey()))  
                return i;  
        return NOT_FOUND;  
    }  
    ...  
}
```

클래스, 객체, 인터페이스, 상속, 구현 !?

- 뭐가 이리도 복잡해요? 장난감 로봇으로 이해해 보자.

인공지능 로봇:

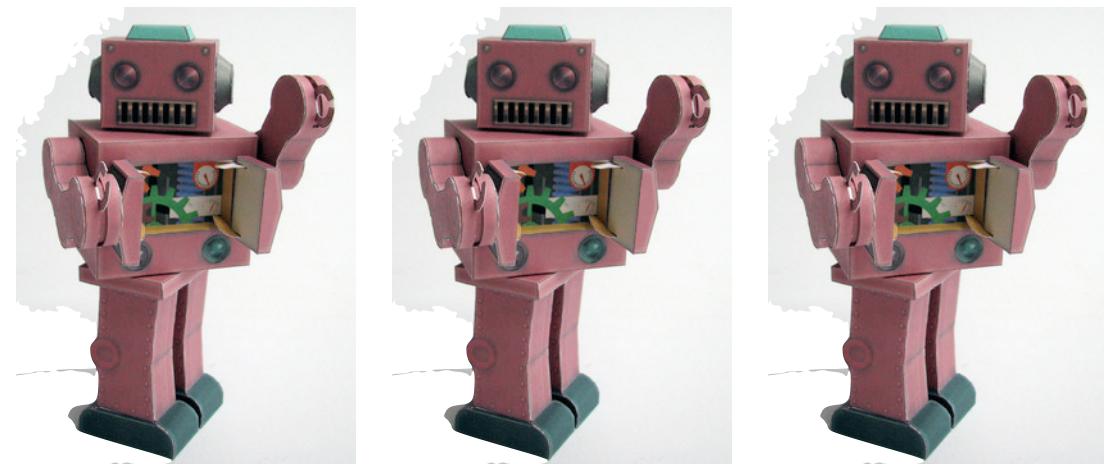
- * 대화
- * 장애물을 피하면서 이동

인터페이스

클래스



객체들

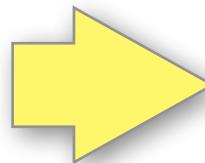


상속: 구현을 재사용

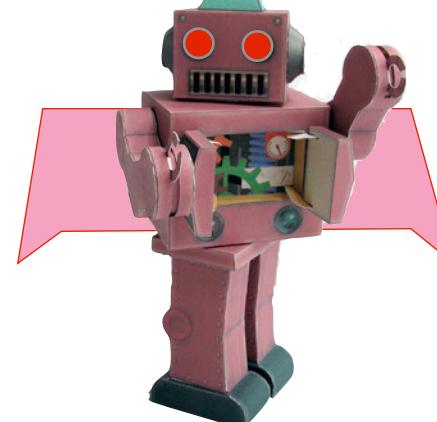
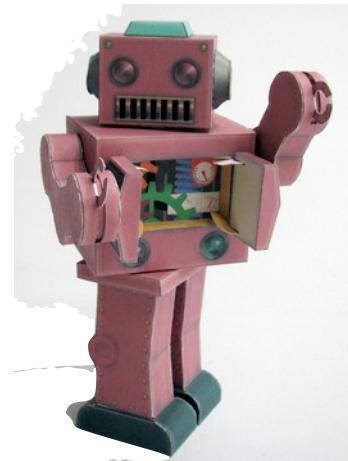
클래스



상속



하위클래스



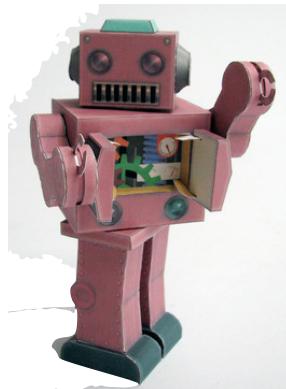
기존기능+
레이저 눈+
날개

인터페이스 구현: 요구조건만 만족

인터페이스

인공지능 로봇:

- * 대화
- * 장애물을 피하면서 이동



상속 vs 구현

- 상속과 인터페이스 구현은 유사한 속성이 있다.
 - class B extends A
 - class B implements A
 - A x = new B();
- 하지만, 상속과 인터페이스 구현은 엄연히 다른 개념임을 명심하라.
 - 상속은 코드의 재사용
 - 인터페이스 구현은 명세를 만족하는 코드 구현

예제, 탐험 게임

- 플레이어들이 방을 탐험하는 게임을 작성하고 싶다. 방에 들어갈 때는 암호를 말해서 맞으면 들어갈 수 있다.
- 방은
 - 플레이어가 들어갈 수 있어야 한다.
 - 플레이어가 나갈 수 있어야 한다.
 - 누가 방에 있는지 알려 주어야 한다.
- 플레이어는
 - 말할 수 있어야 한다.
 - 방을 탐험할 수 있어야 한다.

인터페이스

```
public interface RoomBehaviour {  
    public boolean enter(PlayerBehaviour p);  
    public void exit(PlayerBehaviour p);  
    public PlayerBehaviour occupantOf();  
}  
  
public interface PlayerBehaviour {  
    public String speak();  
    public boolean explore(RoomBehaviour r);  
}
```

단출한 방

```
public class BasicRoom implements RoomBehaviour {  
    private PlayerBehaviour occupant;  
    private String rooms_name;  
    private String secret_word;  
  
    public BasicRoom(String name, String password) {  
        occupant = null; rooms_name = name; secret_word = password;  
    }  
    public boolean enter(PlayerBehaviour p) {  
        boolean result = false;  
        if(occupant == null && secret_word.equals(p.speak())) {  
            occupant = p; result = true;  
        }  
        return result;  
    }  
    public void exit(PlayerBehaviour p) {  
        if(occupant == p) occupant = null;  
    }  
    public PlayerBehaviour occupantOf() { return occupant; }  
}
```

탐험가

```
public class Explorer implements PlayerBehaviour {  
    private String name, secret_word;  
    private RoomBehaviour where_I_am;  
  
    public Explorer(String n, String w) {  
        name = n; secret_word = w; where_I_am = null;  
    }  
    public String speak() { return secret_word; }  
    public void exitRoom() {  
        if(where_I_am != null) {  
            where_I_am.exit(this);  
            where_I_am = null;  
        }  
    }  
    public boolean explore(RoomBehaviour r) {  
        if(where_I_am != null) exitRoom();  
        boolean went_inside = r.enter(this);  
        if(went_inside) where_I_am = r;  
        return went_inside;  
    }  
    public RoomBehaviour locationOf() { return where_I_am; }  
}
```

구동 코드

```
RoomBehaviour[] ground_floor = new RoomBehaviour[4];
ground_floor[0] = new BasicRoom("kitchen", "pasta");
ground_floor[3] = new BasicRoom("lounge", "swordfish");
Explorer harpo = new Explorer("Harpo Marx", "swordfish");
Explorer chico = new Explorer("Chico Marx", "tomato");
boolean success = harpo.explore(ground_floor[3]);
```

인터페이스를 키우고 싶어요

- 보물이 있는 방으로 확장하고 싶다.
- 보물획득 메소드가 있어야 한다.

```
public interface TreasureProperty {  
    public String contentsOf();  
}  
public interface TreasuryRoomBehaviour {  
    public boolean enter(PlayerBehaviour p);  
    public void exit(PlayerBehaviour p);  
    public TreasureProperty yieldTreasure(PlayerBehaviour p);  
    public PlayerBehaviour occupantOf();  
}
```

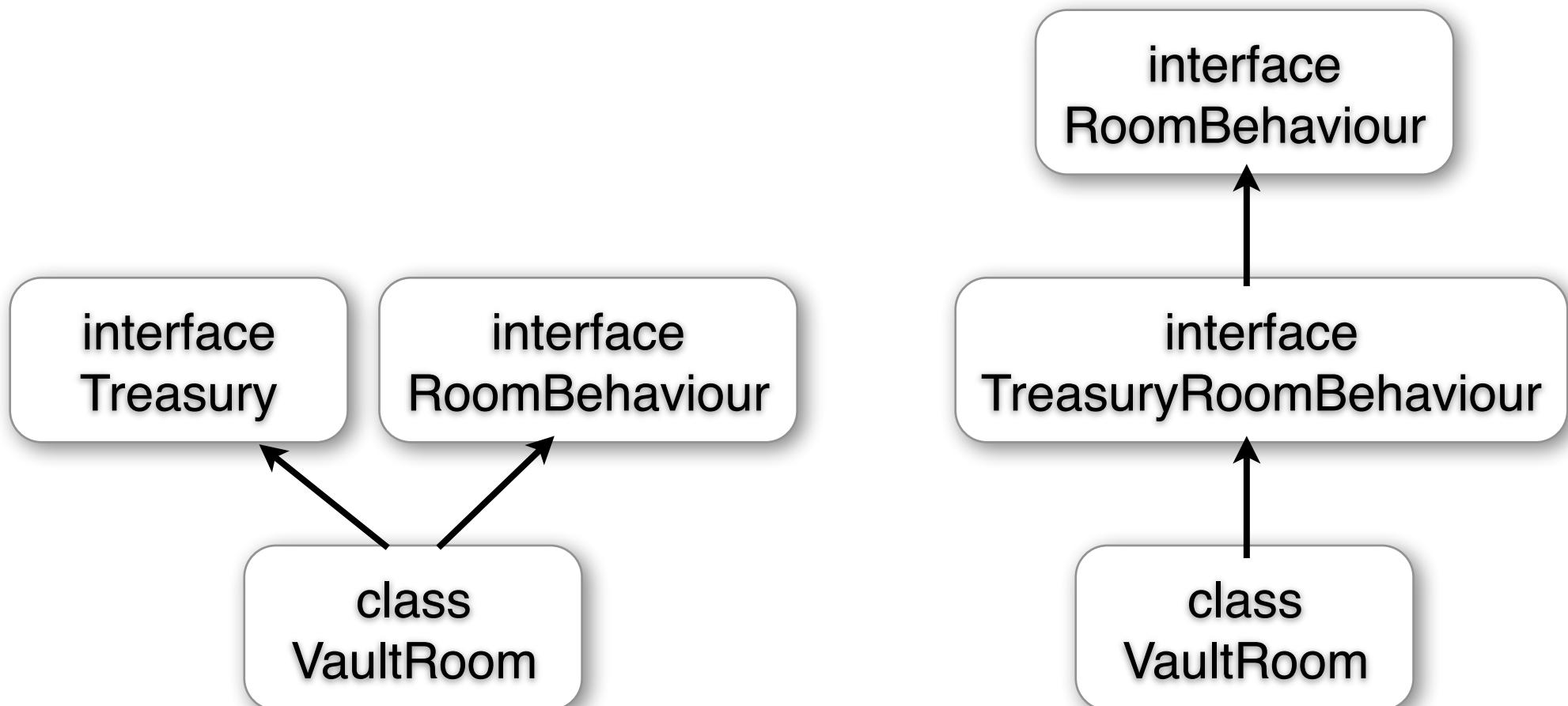
첫 번째 방법: 인터페이스 동시 구현

```
public interface RoomBehaviour {  
    public boolean enter(PlayerBehaviour p);  
    public void exit(PlayerBehaviour p);  
    public PlayerBehaviour occupantOf();  
}  
  
public interface Treasury {  
    public TreasureProperty yieldTreasure(PlayerBehaviour p);  
}  
  
public class VaultRoom implements Treasury, RoomBehaviour {  
    // 방의 속성과 보물이 있는 속성을 동시에 만족하는 코드  
    ...  
}
```

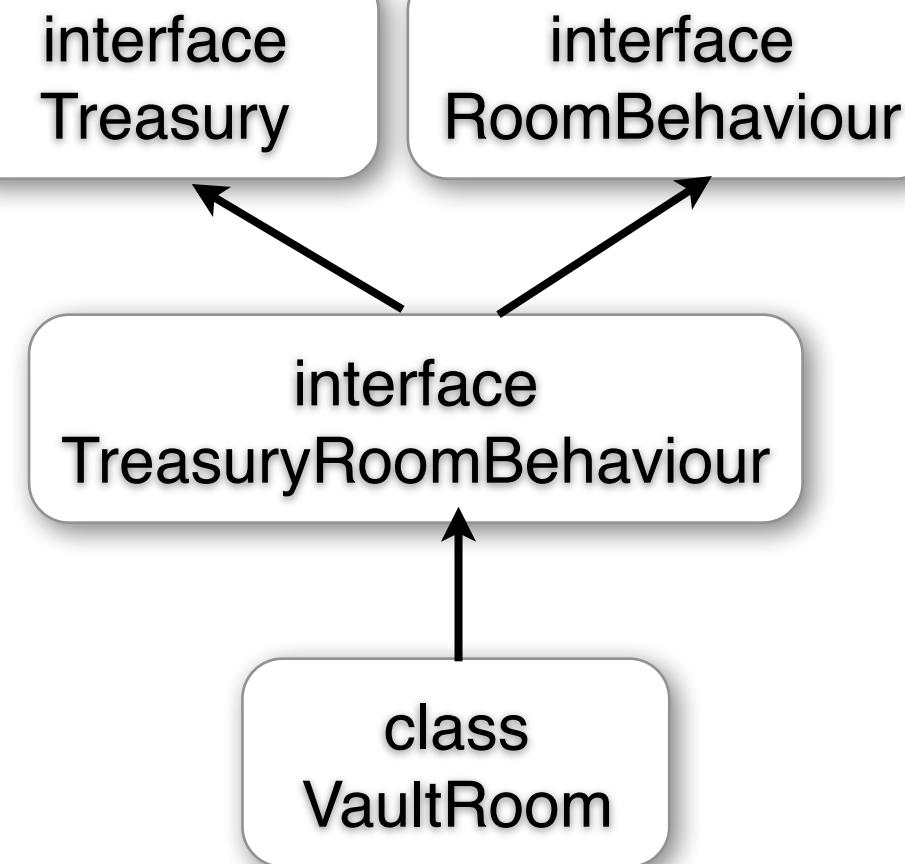
두 번째 방법: 인터페이스를 확장

```
public interface RoomBehaviour {  
    public boolean enter(PlayerBehaviour p);  
    public void exit(PlayerBehaviour p);  
    public PlayerBehaviour occupantOf();  
}  
  
public interface TreasuryRoomBehaviour extends RoomBehaviour {  
    public TreasureProperty yieldTreasure(PlayerBehaviour p);  
}  
  
public class VaultRoom implements TreasuryRoomBehaviour {  
    // 보물 있는 방의 속성을 만족하는 코드  
    ...  
}
```

두 방법은 다르다



그럼, 이건 어떨까?



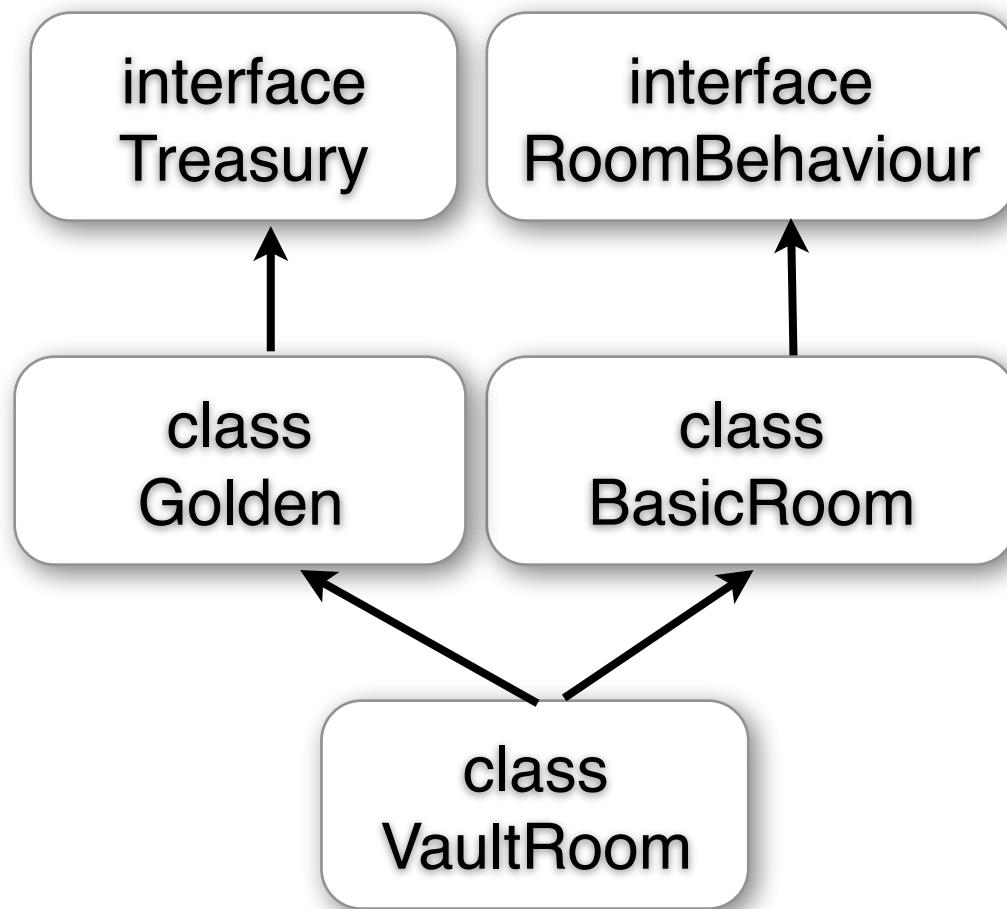
```
public interface
TreasuryRoomBehaviour extends
Treasury, RoomBehaviour {

}

public class VaultRoom implements
TreasuryRoomBehaviour {
    // 보물 있는 방을 만족하는 코드
    ...
}
```

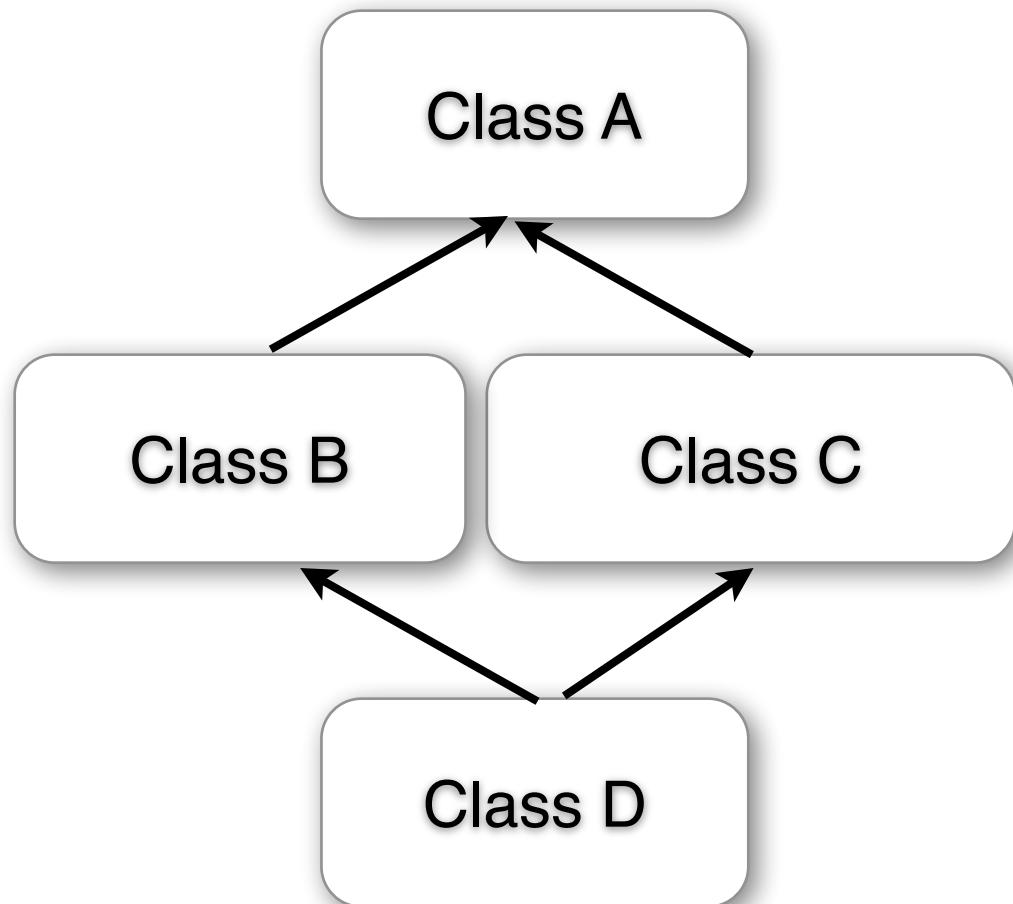
선택은 편의에 따라

이건 어떨까?



이런 것을 다중상속
(multiple inheritance)라
고 한다.
문제점이 있어 Java에서
는 지원하지 않는다.
(다이아몬드 문제 때문)

다중 상속의 문제점



다이아몬드 문제:

A의 method m
B,C가 method m 재정의
D가 B,C 상속 후 m 호출

=> 어느 m이 호출되어야?

여기까지 정리

- 하나의 클래스가 여러 인터페이스 구현 가능
- 인터페이스를 확장 가능
- 여러 인터페이스를 단일 인터페이스로 조합 가능

- 하지만, 여러 클래스를 상속받는 것은 불가능

상속, 구현을 통해서 얻을 수 있는 이득

- 여러 클래스를 한 번에 처리 할 수 있다.
- 서로 다른 클래스에 대해 기본적으로 별도 처리 필요
 - $m(B\ x)\{ \dots \}$
 - $m(C\ x)\{ \dots \}$
 - $m(D\ x)\{ \dots \}$
- 만약 ...의 내용이 별로 다르지 않으면 하나로 구현할 수 있다
 - $m(A\ x)\{ \dots \}$
 - 단, B, C, D가 A를 상속 받았거나, A를 구현했어야 가능

하위타입 (Subtype)

- 타입을 값의 집합으로 보고
- 타입에 값의 포함관계로 상하위 관계를 줄 수 있다.
- 예,
 - 수학에서 정수 < 실수
 - 한국사람 < 사람 < 포유류 < 생물
 - 컴퓨터공학과 학생 < 학생
- 다른 말로, 하위가 할 수 있는 일이 더 많다.

하위타입으로 치환 가능

- 하위타입으로 치환해도 말이 된다.
 - 생물은 죽는다 ==> 사람은 죽는다
 - 사람은 음식을 먹는다 ==> 한국사람은 음식을 먹는다
- 반대가 항상 성립하는 것은 아니다.
 - 사람은 입이 있다 ==> 생물은 입이 있다
 - 한국사람은 한국어를 한다 ==> 사람은 한국어를 한다

Java에서의 하위타입

- 상속 받거나 인터페이스를 구현하면 하위타입
 - class B extends A
 - class B implements A
 - B < A
- 상위타입을 하위타입으로 치환가능?
 - 하위타입의 접근가능 필드와 메소드가 상위타입보다 많으므로
 - a.method(); 를 b.method(); 로 치환해도 문제없음
 - b.method(); 를 a.method(); 로 치환하면 문제 발생 가능
 - **상위타입 변수에 하위타입 객체를 넣어도 문제없음**
 - **하지만 하위타입 변수에 상위타입 객체를 넣는 것은 불허**

다시 이야기하면

- A x = y;
- y의 타입이 B라고 할 때 $A \geq B$ 이면 OK.
- 인수 전달의 경우
 - void m(A x)에 대해 m(y)를 호출하면 y가 x에 저장되는 것과 같다.
 - 즉, y의 타입이 B라고 할 때 $A \geq B$ 이면 OK.

다시 레코드로 돌아와서

- IntegerKey 객체, CodeKey 객체 모두 Key 객체이다.

```
class findLocationDatabase {  
    ...  
    private int findLocation(Key k)  
    {  
        for (int i=0; i<base.length; i++)  
            if(base[i] != null && k.equals(base[i].getKey()))  
                return i;  
        return NOT_FOUND;  
    }  
    ...  
}
```

이상한 일!

```
interface Key {  
    boolean equals(Key y);  
}
```

키끼리 비교할 수 있어야 한다.

```
class IntegerKey implements Key {  
    private int key;  
    public IntegerKey(int i) { key = i; }  
    public int getInt() { return key; }  
  
    boolean equals(Key k) {  
        return key == ((IntegerKey)k).getInt();  
    }  
}
```

정수 키끼리 비교하고 싶은데...

boolean equals(IntegerKey k)로 하면 안될까요?

다시 Key로 돌아 와서

```
interface Key {           equals 메소드가 모든 Key 객체를 받을
    boolean equals(Key y); 수 있어야 한다.
}
```

```
class IntegerKey implements Key {           Key k를 IntegerKey k로 바꾸면
    ...
    boolean equals(Key k) {           다른 Key 객체를 k로 받을 수 없음
        return key == ((IntegerKey)k).getInt();
    }
    ...
}
```

우리의 의도는 명세에서 Key y가 모든 Key 객체가 아닌 구현하는 클래스, 예를 들어, IntegerKey를 구현할 때는 IntegerKey y, CodeKey를 구현할 때는 CodeKey y이길 바라는 것이다.

강제 타입 변환을 더 안전하게 하려면

- 강제 타입 변환은 위험하다.
 - boolean equals(Key k) { ... (IntegerKey) k ... }
 - IntegerKey ik; CodeKey ck;
 - ik.equals(ck);
- 안전하게 검사하는 방법은?
 - if(k **instanceof** IntegerKey) {
 ... (IntegerKey) k ...
}
else { 오류처리 }

예, IntegerKey.equals

```
interface Key {  
    boolean equals(Key y);  
}  
  
class IntegerKey implements Key {  
    ...  
    boolean equals(Key k) {  
        if(k instanceof IntegerKey)  
            return key == ((IntegerKey)k).getInt();  
        else  
            return false;  
    }  
    ...  
}
```

통칭 (Generic)

- Key 인터페이스를 만족하려면, “자기와 같은” Key 객체와 equals로 비교할 수 있어야 한다.
- 이를 명세하려면 타입에 인수가 필요
- Key<T>
 - “T” 객체와 equals로 비교할 수 있다.
- Key<T extends Key<T>>
 - “Key<T>를 구현한 T” 객체와 equals로 비교할 수 있다.
- class IntegerKey implements Key<IntegerKey> { ... }
- IntegerKey와 비교하는 것만 작성하면 된다.

예, IntegerKey.equals

```
interface Key <T extends Key<T>> {  
    boolean equals(T y);  
}  
  
class IntegerKey implements Key <IntegerKey> {  
    ...  
    boolean equals(IntegerKey k) {  
        return key == k.getInt();  
    }  
}
```

Database

```
public interface Key <T extends Key<T>>{  
    boolean equals(T k);  
}  
public interface Record <K extends Key<K>>{  
    K getKey();  
}  
public class Database <K extends Key<K>, R extends Record<K>>{  
    private R[] base;  
    ...  
    public Database (int initial_size) { ... }  
    private int findLocation(K k) { ... }  
    private int findEmpty() { ... }  
    public R find(K k) { ... }  
    public boolean delete(K k) { ... }  
    public boolean insert(R r) { ... }  
}
```

주의: new R[10]이 허용되지 않아 (R[]) new Record[10]을 사용해야 함

추상 클래스 (Abstract Class)

- 구현이 덜 된 클래스
 - 메소드 중 일부가 정의되지 않은 클래스
- 메소드 구현을 공유하는 경우, 인터페이스보다 유리
 - 하위 클래스들이 메소드 중 일부를 동일하게 사용 => 상속을 통한 코드 재사용
 - 하위 클래스들이 메소드 중 일부는 반드시 구현해야 함 => 인터페이스 만족

예제, 카드 게임

- class Card와 class CardDeck은 예전 것을 사용
- class Dealer는 게임을 진행
 - 특정 플레이어 차례가 되면 플레이어에게 카드를 원하는가 묻고, 그 렇다고 하면 카드통에서 뽑아 카드를 준다.
- interface CardPlayerBehaviour는 게임을 함
 - 다음 interface를 만족

```
public interface CardPlayerBehaviour {  
    public boolean wantsACard();  
    public void receiveCard(Card c);  
}
```

컴퓨터 플레이어, 사람 플레이어

- 컴퓨터 플레이어나 사람 플레이어나 카드를 받을 때는 똑같이 수행된다.
 - 카드를 받아서 자기 손에 준다.
- 하지만, 카드를 받을래? 물었을 때
 - 사람 플레이어의 경우 입력창을 통해 답을 구한다.
 - 컴퓨터 플레이어의 경우 내부 알고리즘을 통해 계산해서 답을 구한다.
- 컴퓨터 플레이어와 사람 플레이어의 상위 "추상" 클래스 CardPlayer를 작성해서 상속 받도록 하자.

추상 클래스 카드플레이어

```
public abstract class CardPlayer implements CardPlayerBehaviour
{
    private Card[] my_hand;
    private int card_count;

    public CardPlayer(int max_cards) {
        my_hand = new Card[max_cards];
        card_count = 0;
    }
    public abstract boolean wantsACard();
    public void receiveCard(Card c) {
        my_hand[card_count] = c;
        card_count = card_count + 1;
    }
}
```

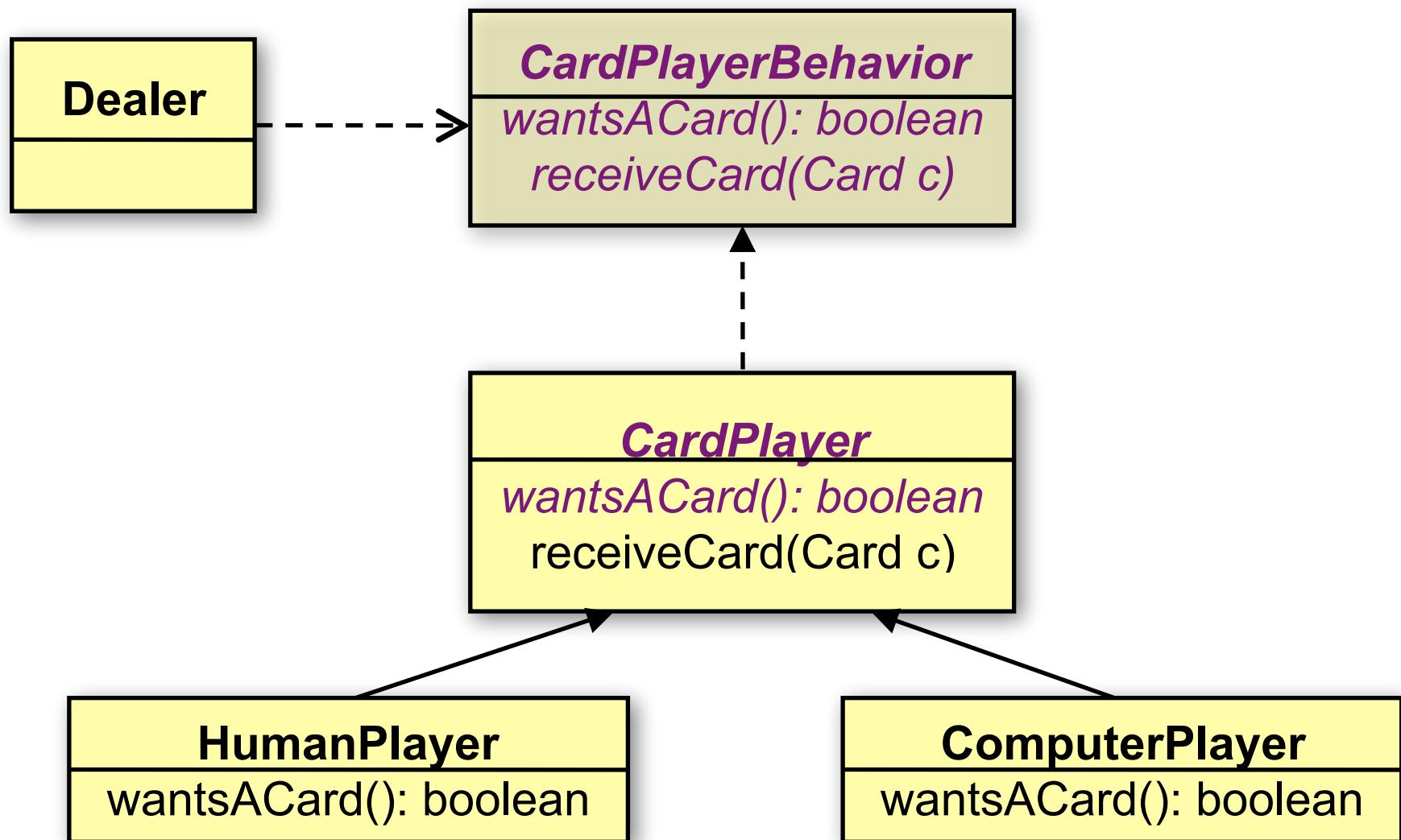
사람 플레이어

```
import javax.swing.*;  
  
public class HumanPlayer extends CardPlayer {  
    public HumanPlayer(int max_cards) {  
        super(max_cards);  
    }  
    public boolean wantsACard() {  
        String response = JOptionPane.showInputDialog  
            ("Do you want another card (Y or N)?");  
        return response.equals("Y");  
    }  
}
```

컴퓨터 플레이어

```
public class ComputerPlayer extends CardPlayer {  
    public ComputerPlayer(int max_cards) {  
        super(max_cards);  
    }  
    public boolean wantsACard() {  
        int sum = 0;  
        for(int i=0; i<card_count; i++)  
            sum += my_hand[i].getCount();  
        return sum < 15;  
    }  
}
```

카드 게임의 소프트웨어 구조



상속을 통한 클래스 계층 구조

- 동물

- 온혈동물

- 고양이과

- 사자
- 호랑이

- ...

- 말과

- 말
- 얼룩말

- 소과

- ...

- ...

- 냉혈동물

- ...

```
public abstract class Animal
{ // 동물에 관계된 필드와 메소드 }
```

```
public abstract class WarmBlooded
    extends Animal
```

```
{ // 온혈동물에 관계된 필드와 메소드 }
```

```
public abstract class Equine
    extends WarmBlooded
```

```
{ // 말과에 관계된 필드와 메소드 }
```

```
public class Horse extends Equine
{ // 말에 관계된 메소드를 채움으로써 완성 }
```

하나 더 예제

- Form
 - Point
 - Line
 - Straight
 - Jagged
 - Curved
 - Shape
 - Curved
 - Circle
 - Ellipse
 - Polygon
 - Triangle
 - Rectangle
 - ...

모든 도형에서 사용하는 필드와 메소드는 상위에서 정의하고 아래로 내려가면서 각 도형들만이 사용하는 메소드를 정의

프레임워크 (Framework)

- 프레임워크란 웬만한 것이 다 정의되어 있고 사용자가 일부만 채워주면 실체가 나오는 것을 말한다.
- 예, GraphicsWindow
 - setSize, setBackground, setVisible 등은 다 구현되어 있음
 - paint 메소드만 구현되어 있지 않음
 - paint 메소드만 구현하면 창이 완성
- Java에서는 추상 클래스로 프레임워크를 구현할 수 있다.

인터페이스 vs 추상 클래스

| 인터페이스 | 추상클래스 |
|--|---|
| 행위에 대한 명세만 정의 | 일부는 구현, 일부는 구현 없음 |
| 인터페이스를 구현한다는 것은, 명세를 만족하는 코드를 작성한다는 것이다. | 상속을 받으면 구현된 코드는 재사용하고, 구현 없는 코드만 채워 준다. |
| 클래스가 어떻게 구현되었는지 상관없이 그룹 짓거나 연결할 때 사용 | 클래스를 구현 할 때 일부 코드가 동일하게 사용되는 여러 클래스가 있을 때 사용. |
| <i>Interfaces list behaviours</i> | <i>abstract classes list codings</i> |

질문

- OK, 상속과 구현을 통한 하위 타입은 이해했어요. 그런데,
이런 것은 구현할 수 없나요?
 - 아무거나 다 들어갈 수 있는 배열 또는 쌍
 - 아무 배열을 받아서 그 크기를 반환하는 메소드

Object를 활용한 예제

- `Object[] a = new Object[10];`
 - 아무거나 들어가는 10개 짜리 배열
 - `a[0] = new Integer(10); a[1] = "abc";`
 - 그럼, `a[1]`의 타입은?
- `int arraySize(Object[] a) { return a.length; }`
 - 아무 배열이나 그 크기를 반환하는 함수
 - `...arraySize(new int[10]); ...arraySize(new String[5]);`
 - 그럼, `a[1]`의 타입은?

조심해야 하는 것

- 올라간 것은 다시 내려주어야.

```
Object[] a = new Object[10];
a[0] = new Integer(25);
System.out.println(a[0].getInteger() / 10);
// 오류가 발생. a[0]는 Object 타입으로
// getInteger 메소드가 없음.
```

```
System.out.println(((Integer)a[0]).getInteger() / 10);
// 강제 타입변환을 통해 정수로 내려주어야.
```

```
System.out.println(a[0]);
// 이건 문제 없음. Object 클래스에는 toString() 메소드가
// 있기 때문
```

패키지 (Package)

- 패키지는 클래스, 인터페이스들을 모아 놓은 폴더이다.
 - 예, java.util, java.awt, javax.swing
- import 명령어를 통해 패키지에 있는 것들을 사용할 수 있다.
- 클래스, 인터페이스보다 상위에 있는 부품구조로 단순히 폴더별로 모아 놓은 것 뿐이다.

정리

- Java에서는 다양한 부품구조를 제공한다.
 - 클래스 및 상속
 - 인터페이스: 스펙을 정의하고 이에 맞추어 구현
 - (다중) 인터페이스 구현
 - (다중) 인터페이스 확장
 - 추상클래스: 구현 중 일부를 비워두고 나중에 구현
 - 상속, 다중 불가
 - 패키지
- 부품들끼리의 관계는 하위타입을 따른다.
 - Java에서는 상속, 구현, 확장을 통해 관계 성립
 - 상위타입 변수에는 하위타입 객체를 저장할 수 있다.
 - 맨 위에 Object가 계신다.