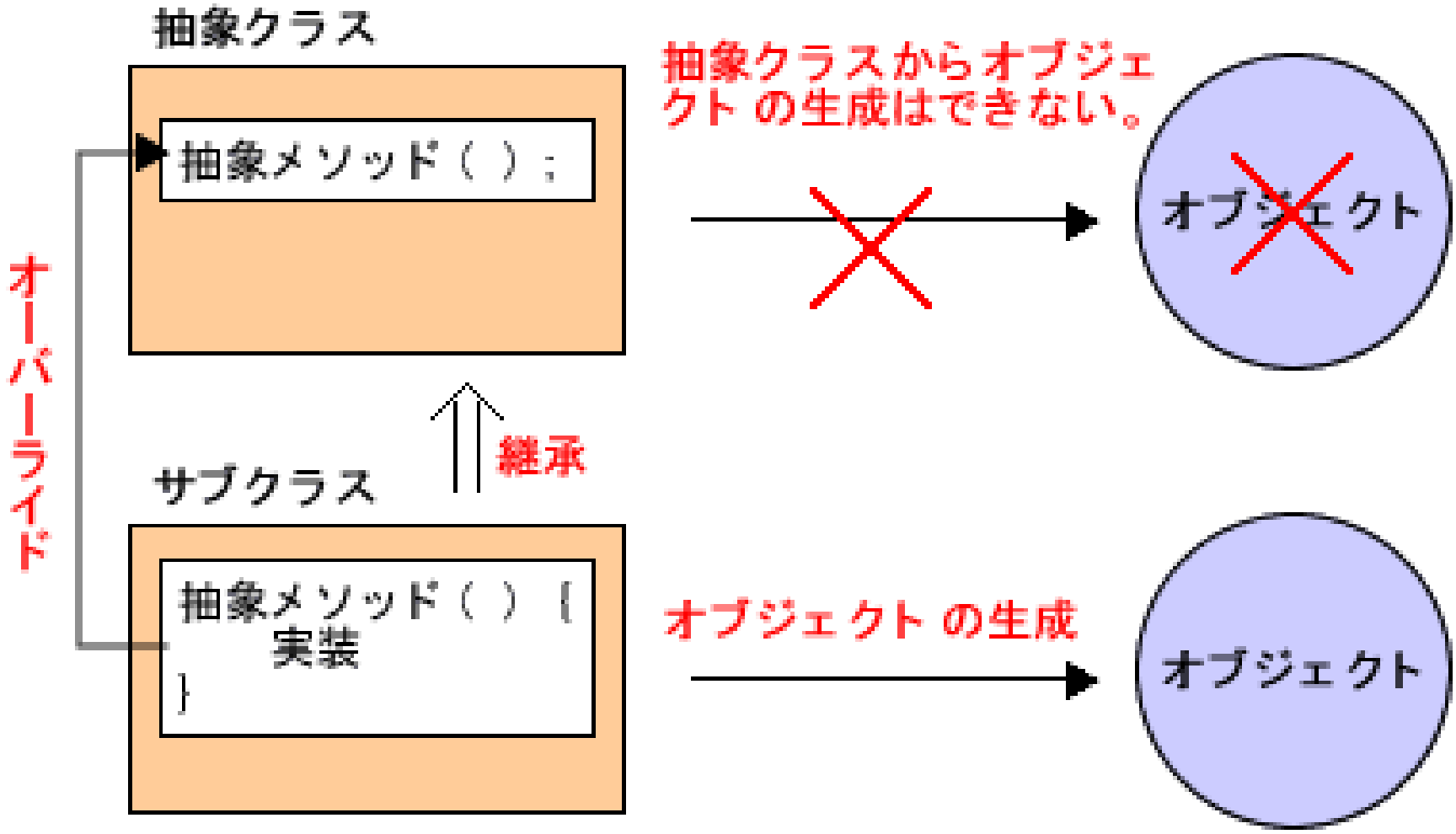
Javaプログラミング（6）
抽象クラス，インタフェース

成蹊大学理工学部
情報科学科

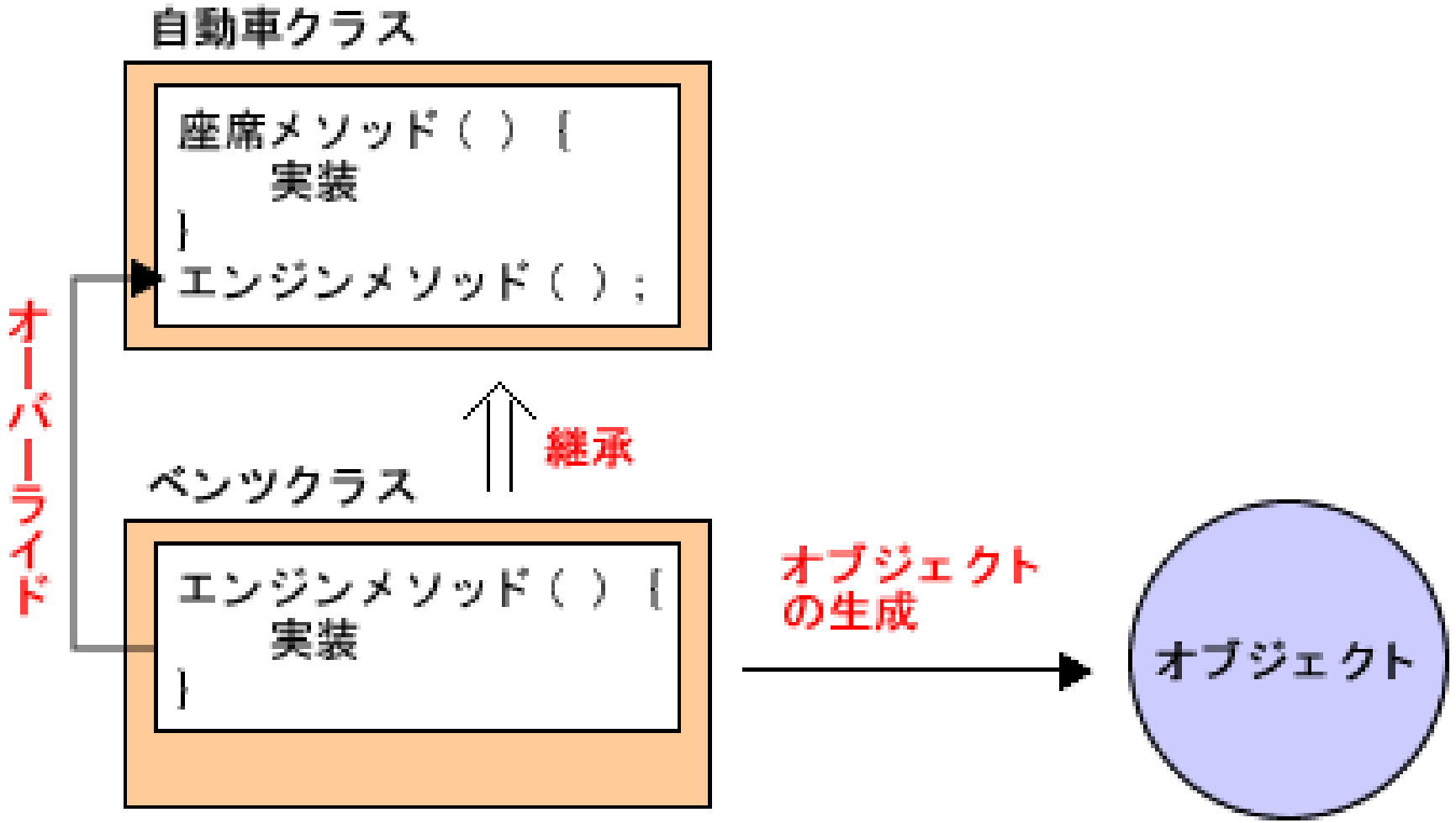
抽象クラス

- メソッドの名前と引数の型だけが決まっていて，本体の実装がないメソッドを「抽象メソッド」，あるいは「abstractメソッド」と呼ぶ.
- 抽象メソッドを含むクラスを「抽象クラス」，あるいは「abstractクラス」と呼ぶ.
- 抽象クラスはインスタンスを作ることができない.

抽象クラスと継承


抽象クラス
抽象メソッド（ ）；
抽象クラスからオブジェクトの生成はできない。
オブジェクト
オーバーライド
継承
サブクラス
抽象メソッド（ ） {
実装
}
オブジェクトの生成
オブジェクト

抽象クラスと継承の例


自動車クラス
座席メソッド（ ） {
実装
}
エンジンメソッド（ ）;
オーバーライド
継承
ベンツクラス
エンジンメソッド（ ） {
実装
}
オブジェクト
の生成
オブジェクト

抽象クラス，抽象メソッドの宣言

- 抽象クラスの宣言
  [修飾子] abstract class クラス名 { クラス本体 }
  （例）**abstract class** Animal {. . . }
- 抽象メソッドの宣言
  [修飾子] abstract 戻り値の型 メソッド名（引数型 引数名）;
  （例）**abstract void** taberu(String esa);

抽象クラスの例

```
//抽象クラスAnimal
abstract class Animal {
        //フィールド宣言
        String subType;
        String esa;
        //抽象メソッドtaberuの宣言
        abstract void taberu(String esa);
        //抽象メソッドではないメソッド
        void shokuji () {
                for (int i=0;i<3;i++) {
                        taberu("esa"+i);
                }
        }
}
```

抽象クラスの拡張例

```
//Animalの拡張クラス1
public class Dog extends Animal {
        String barkVoice;
        //コンストラクタ
        Dog(){
                this.subType="dog";
                this.esa="noinfo";
                this.barkVoice="bowwow";
        }
        Dog(String esa, String voice) {
                this.subType="dog";
                this.esa=esa;
                this.barkVoice=voice;
        }
        //抽象メソッドのオーバーライド
        void taberu(String esa){
                System.out.println(esa+"を食べました");
        }
...
```

抽象クラスの拡張例(Cont.)

```
. . .
        //抽象メソッドではないメソッドのオーバーライド
        void shokuji(){
                for (int i=0;i<3;i++) {
                        taberu(this.esa+i);
                }
        }
        public static void main (String[] args) {
                Dog dog1 = new Dog("dogfood","wanwan");
                dog1.shokuji();
        }
}
```

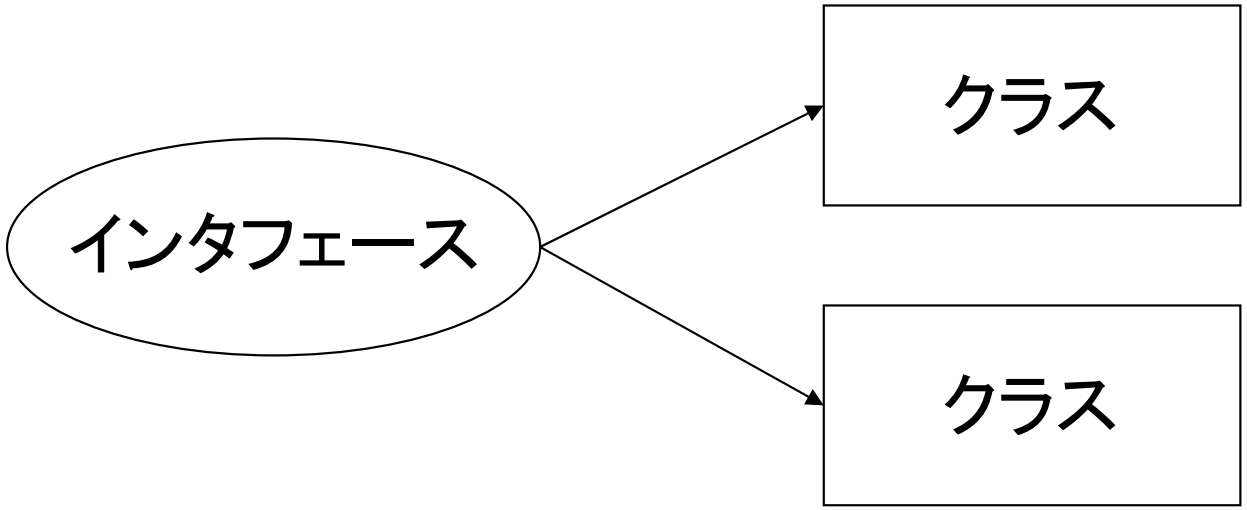
実行結果

```
dogfood0を食べました
dogfood1を食べました
dogfood2を食べました
```

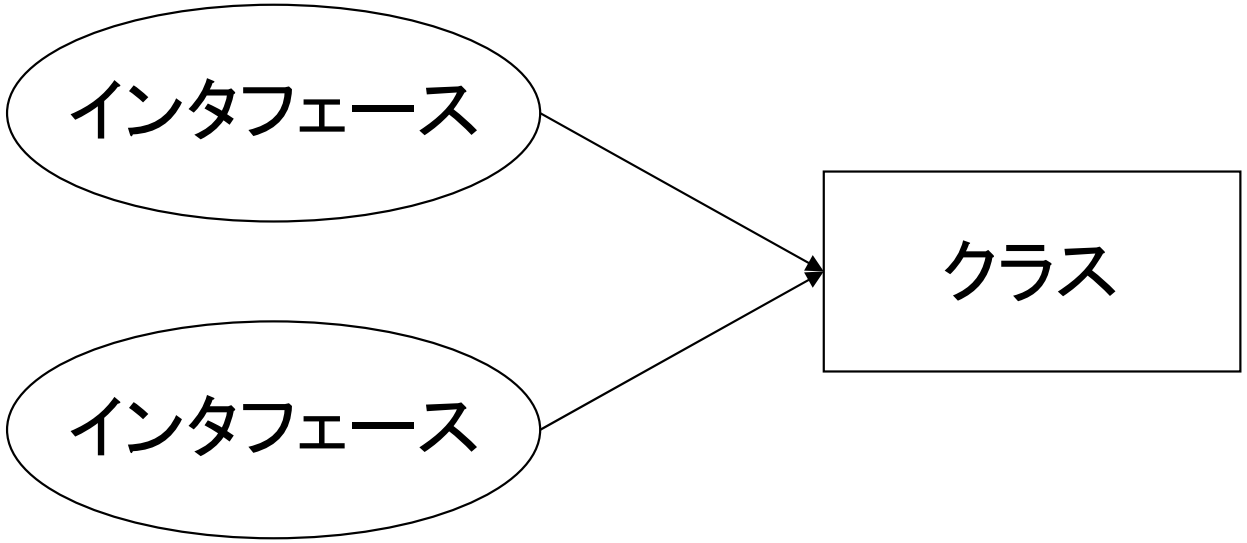
インタフェース

- クラスとの類似点
  - 参照型である
  - フィールドとメソッドを持つ
  - 拡張ができる（スーパーインタフェース，サブインタフェースを持つ）
- クラスとの相違点
  - インスタンスを作ることができない
  - メソッドは必ず抽象メソッドである（メソッドは実装されていない）
  - フィールドは必ず定数(public static final) である

インタフェースの適用

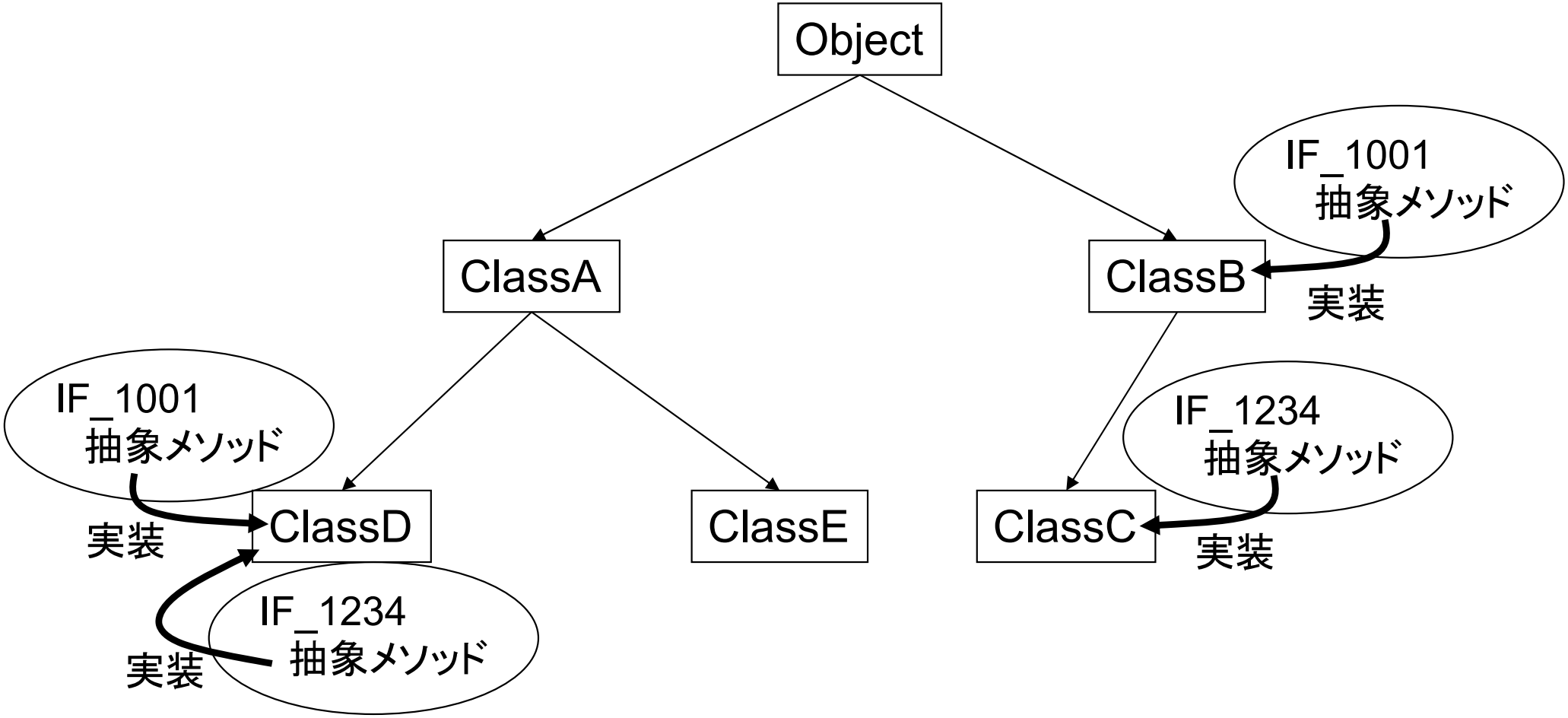


継承とは無関係に複数のクラスに適用可能



1つのクラスが複数のインタフェースを実装可能

インタフェースの適用


Object
ClassA
ClassB
IF_1001
抽象メソッド
実装
IF_1001
抽象メソッド
実装
ClassD
ClassE
IF_1234
抽象メソッド
実装
ClassC
IF_1234
抽象メソッド
実装

インタフェースの宣言と実装

- インタフェースの宣言
  [修飾子] interface <インタフェース名> {. . . }
  - インタフェースの修飾子はpublicのみ（なしも可）
- インタフェースの実装
  [修飾子] class <クラス名>　implements <インタフェース名リスト>{. . . }

インタフェースの例

```
//インタフェースの宣言
interface Naku {
        void printNakigoe();
}
//Nakuインタフェースの実装
public class Dog extends Animal implements Naku {
. . .
//インタフェース中のメソッドはpublic なので，publicで実装
        public void printNakigoe() {
                for (int i=0;i<3;i++) {
                        System.out.print(this.barkVoice+"!");
                }
        }
. . .
}
```

利用例

```
public static void main (String[] args) {
        Dog dog1 = new Dog("dogfood","wanwan");
        dog1.shokuji();
        dog1.printNakigoe();
}
```

実行結果

```
dogfood0を食べました
dogfood1を食べました
dogfood2を食べました
wanwan!wanwan!wanwan!
```

インタフェースの特徴

- フィールドには自動的に修飾子 public static finalが付けられる
- メソッドには自動的に修飾子 public abstract が付けられる
- インタフェースのメソッドは抽象メソッドでなければならない
- 実装クラスでは複数のインタフェースを実装 (implements) できる
- 実装クラスではインタフェースの全てのメソッド宣言を実装しなければならない
- インタフェースはインスタンスを作れないが，インタフェース型の変数は定義可能
- 別のインタフェースを多重継承 (extends) できる
  interface インタフェース名 extends スーパーインタフェース名

インタフェース型の変数

- インタフェースはインスタンスを作ることはできないが，インタフェース型の変数を使うことはできる．

Naku obj = new Naku();
×インタフェースはインスタンスを作れない

Naku obj = new Dog(“zanpan”, “kyankyan”);
○変数名にすることはできる

これを使って，
obj.printNakigoe();
○実装されたメソッドを使うこともできる

抽象クラスとインタフェース

- 類似点
  - どちらもインスタンスを作ることができない
    - ただし，抽象クラスはコンストラクタは持つことができる
- 相違点
  - インタフェース中のメソッドはすべて抽象メソッド．抽象クラスは，抽象メソッドと抽象でないメソッドの両方を持つことができる．
  - 複数の抽象クラスをextendsできないが，複数のインタフェースをimplementsすることはできる．
    Javaではこれを使って擬似的に多重継承を実現している
    （例）

```
class Dog extends Animal implements
                Hashiru, Naku, Nemuru{ ...
```

等と書くことが可能

  - インタフェースは継承関係とは無関係に様々なクラスに適用することができる（自由に機能（メソッド）の付与が可能）

クラスとインタフェースの比較

	クラス	インタフェース
インスタンス	作れる	作れない
メソッド	いろいろ	必ずpublic abstract
フィールド	いろいろ	必ずpublic static final
拡張	1つだけ class X extends Y {	複数指定可能 interface X extends Y, Z {
インタフェースの利用	複数指定可能 class X implements A, B {	―

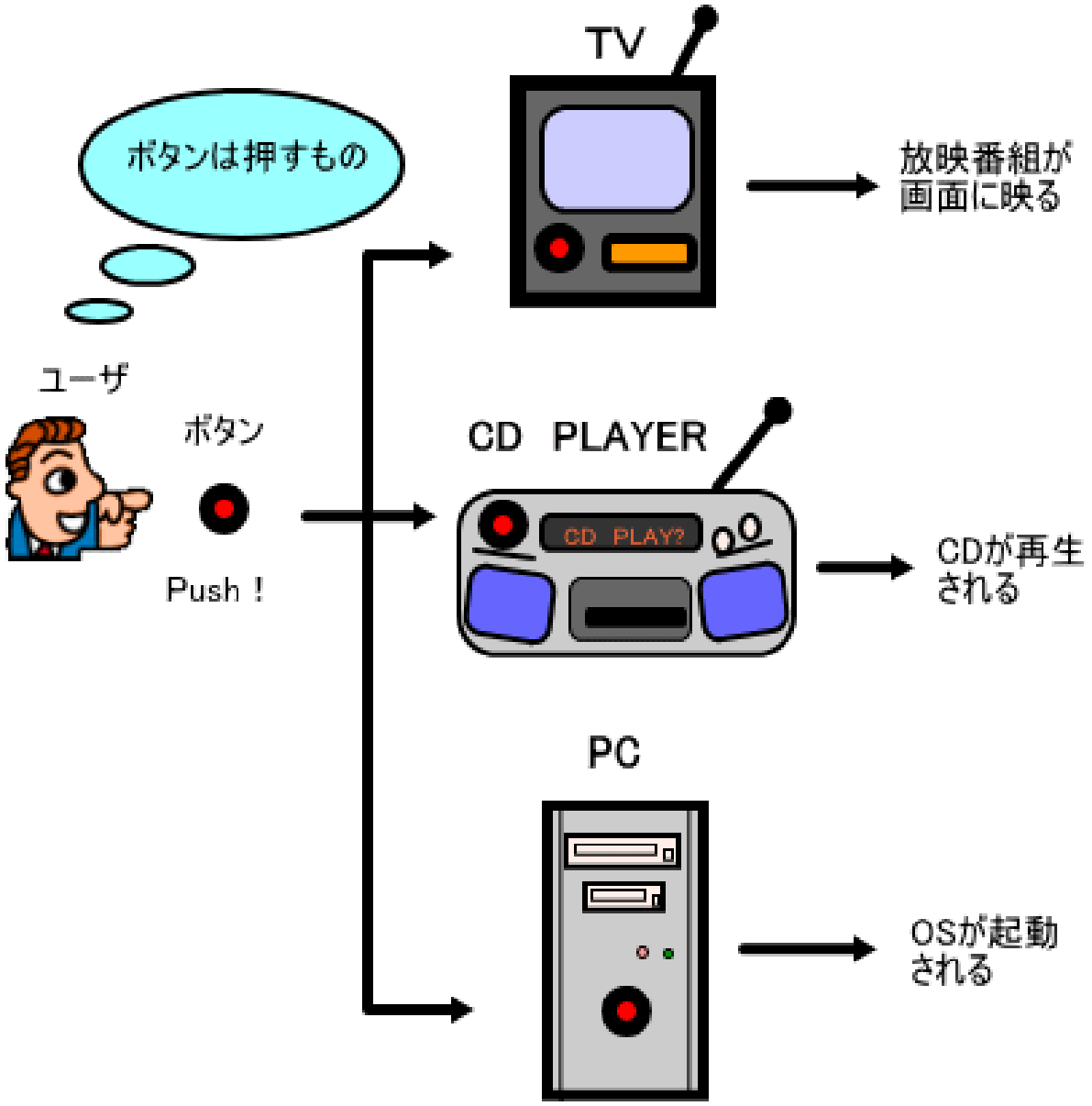
オーバーライドとオーバーロード

- オーバーライド
  - スーパークラスから継承したフィールドやメソッドを変更すること
  - スーパークラスで宣言されたものを「再宣言」，「上書き」すること
- オーバーロード
  - 同じ名前で引数の異なるメソッドを多数宣言すること
  - 同じ名前のメソッドを「多重定義」すること

多態性（ポリモーフィズム）

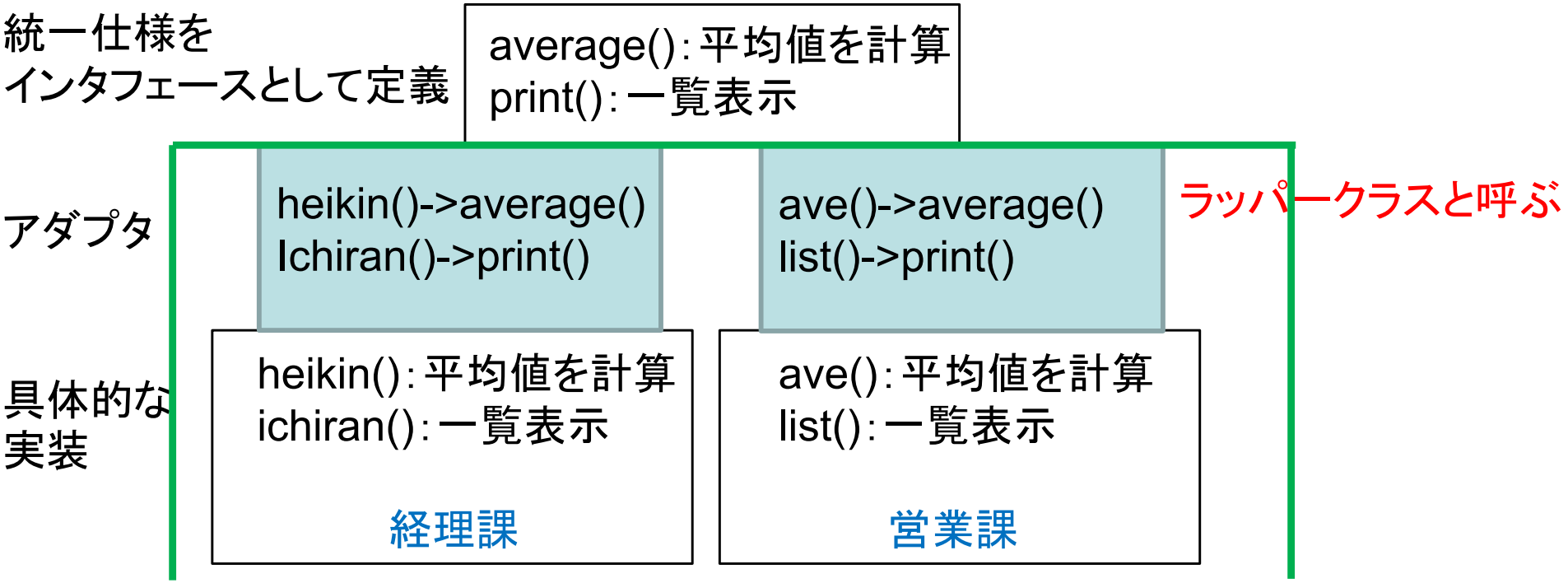
- オブジェクト指向プログラミングにおいて，異なるオブジェクトに対して，同じメッセージを送った場合，そのオブジェクトに合わせて，異なる処理が行われること
- Javaではオーバーライドやオーバーロードを使って多態性を実現している

多態性の概念


TV
ボタンは押すもの
放映番組が
画面に映る
ユーザ
ボタン
CD PLAYER
CD PLAY?
CDが再生
される
Push！
PC
OSが起動
される

インタフェースの有用性

- 社内の様々な部署でデータの一覧や平均値を出力する仕事がある
- しかし，部署によってメソッド名や仕様が違う
- 部署ごとにどのメソッドを使い分けるのは面倒



まとめ

- クラスの意味だけ定義しておく抽象クラス
- 機能の概要のみの集合体，インタフェース
- クラスの拡張とインタフェースの実装の組み合わせにより，擬似的に多重継承を実現
- 特性，機能が共通化できるところは（抽象）クラスとしてまとめる．詳細部分はサブクラスで実装
- 必要に応じてインタフェースを貼り付ける
- オブジェクト指向言語は保守性に優れている