

プログラミング言語論

第6回

オブジェクト指向プログラミング
ポリモーフィズム

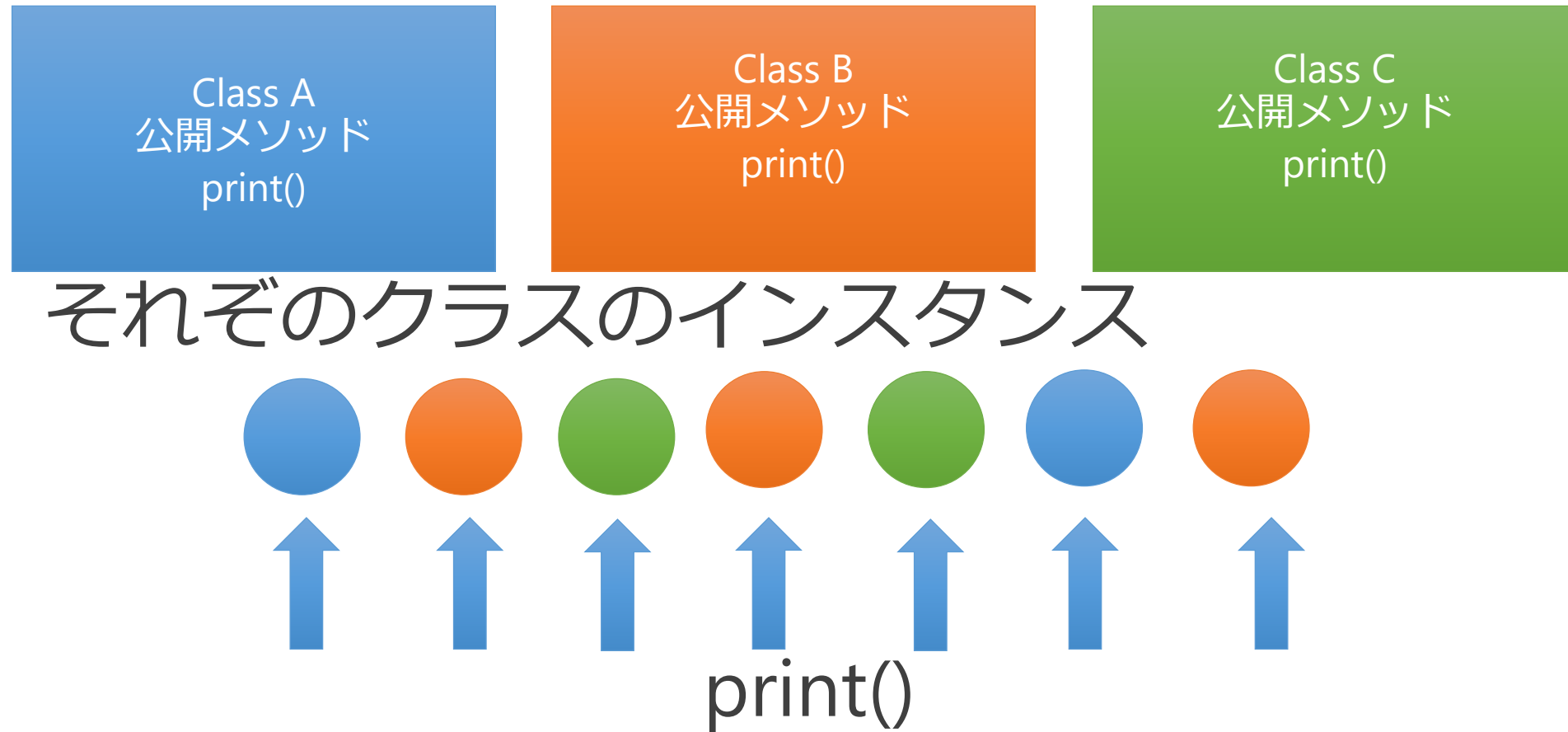
□ 同じ名前で別の関数へアクセスする仕組み

- メソッドに複数の（多様な）振る舞いを対応させるというオブジェクト指向の概念
- 類似したクラスに対するメッセージの送り方を共通にする仕組み

□ 日本語で多態性（たたいせい）、多相性

- 実世界の例：「線を引く」メソッドは、道具（オブジェクト）が鉛筆でも、シャープペンでも、ボールペンでも、有効

ユーザプログラム側は、インスタンスが属するクラスを意識する必要がない



同じ名前の関数を作る仕組み

□オーバーロード

- 関数の引数の数や種類を変えて同じ名前の関数を作成
- 1つのクラスの中に、引数の数や種類を変えて同じ名前を持つメソッドを作成

□オーバーライド

- 継承したサブクラスにおいて、スーパークラスの関数を上書きすること
- 上書きされた関数の扱いによって様々な実装がある
- （トじゃなくてド）

※間違いやすいので違いに注意

- C言語の絶対値計算：名前で変える必要

- abs/labs/fabs

- C++のオーバーロードされた関数群:

- absで全部OK

- abs(int a), abs(double a), abs(float a), abs(short a)...が定義

- 関数のオーバーロードで解決

- 関数の「数の型」に関するポリモーフィズム

- 引き数が違えば、同じ名前で違う挙動をする関数がいくつも書ける
- 例えば、`func(int a)`と`func(short a)`で全く違う動作を定義可能
 - `int`の`func`は二乗を返し、`short`の`func`は2倍を返すなど
 - ただし、そのようなトリッキーなコードはバグの要因になるので**オーバーロード時は関連のある操作を書くこと**

- 動物クラス：「鳴く」というメソッドコール
 - 犬クラス：鳴く→「ワンワン」
 - 猫クラス：鳴く→「ニャー」
 - カラスクラス：鳴く→「カーカー」

- 命令は「鳴く」という同一のものに対して、オブジェクトによって異なる挙動をする実装がしたい

- スーパークラスのオーバーライドの対象となるメソッドを**仮想関数**で宣言
 - C++ではメソッドにvirtualを追加して宣言
- サブクラスでオーバーライド
 - サブクラスでの再定義の時は virtual は不要

□ 仮想関数とは、サブクラスでオーバーライドされる予定の関数のこと（**実体はそのクラスでは実装しない**ので、「仮想」）

□ 純粋仮想関数

- 定義を持たない仮想関数
- C++での書式: `virtual 戻り値型 関数名(引数) = 0;`
- 派生クラスでは必ず実体を実装する必要がある

□ 抽象クラス

- ひとつでも純粋仮想関数があれば抽象クラス

□ インターフェース

- 純粋仮想関数だけのクラス→普通のメソッドも、メンバ変数も一切無い
- Javaで言うインターフェースの実現方法の一種

```
#include <stdio.h>
```

```
class Person  
{  
public:  
    virtual void set_name(const char* name) = 0;  
    virtual void show_name(char* name) = 0;  
};
```

```
int main()  
{  
    Person p;  
    printf("test¥n");  
}
```



動かない。抽象クラスだろ、
これは！とコンパイラに
怒られる

- インターフェース（純粹仮想関数だけからなるクラス）は、それからの派生クラスが持つべき、**公開メソッドを規定**する役割を持つ
- 実体の実装は、派生クラスで受け持つ(**実装が強制**される)
- 前のページの例だと、Personから継承された派生クラスは「必ずset_nameとshow_nameというメソッドを持つ」ことが**保証される**(プロトコルの定義を与える)
- ポリモーフィズムを利用するプログラムは、インスタンスがどの派生クラスに属するかを意識せずにset_nameとshow_nameを利用することができる

- サブクラスで、スーパークラスと全く同じ名前でメソッドを作ればいい。
- オーバーロードを違って引数の数を同じにする。
- 純粹仮想関数の場合、必ずオーバーライドしないとコンパイルエラーとなる。
- この機能を使って、派生クラスを作っているプログラマに「必ずこのメソッドを実装しなさい」というアラートが出せる。

```
#include <stdio.h>
class Person
{
public:
    virtual void set_name(const char* name) = 0;
    virtual void show_name() = 0;
};
class Student: public Person
{
private:
    const char* name;
```

```
public:
    void set_name(const char* _name)
    {
        name = _name;
    }
    void show_name()
    {
        printf("name=%s¥n", name);
    }
};
int main()
{
    Student p;
    p.set_name("Meiko Hanako");
    p.show_name();
}
```

例：派生クラスを複数作ってみる

14

```
#include <stdio.h>
class Fruits
{
public:
    virtual void show_message() = 0;
};
class Apple: public Fruits
{
public:
    void show_message()
    {
        printf("apple¥n");
    }
};
```

```
class Orange: public Fruits
{
public:
    void show_message()
    {
        printf("orange¥n");
    }
};
class Banana: public Fruits
{
public:
    void show_message()
    {
        printf("banana¥n");
    }
};
```

```
int main()
{
    Apple a;
    Orange b;
    Banana c;
    a.show_message();
    b.show_message();
    c.show_message();
}
```

```
int main()
{
    Fruits* x[5];
    Apple a,b;
    Orange c;
    Banana d,e;
    x[0] = &a;
    x[1] = &b;
    x[2] = &c;
    x[3] = &d;
    x[4] = &e;
    for (int i = 0; i < 5; i++) {
        x[i]->show_message();
    }
}
```

apple
apple
orange
banana
banana

と表示される

この部分のコードは
Fruits の派生クラスの
インスタンスならどれ
でも対応できる(汎用的)

ポリモーフィズムを利用したジェネリック(汎用的)なプログラミング

16

LTE: 2つのオブジェクトを比較し、左が右よりも小さいときにtrueを返す

インターフェース
Sortable
virtual bool LTE(.,.)=0

継承と実装

Class A(文字列)
公開メソッド
LTE()

Class B (整数)
公開メソッド
LTE()

Class C (実数)
公開メソッド
LTE()

Sortable
なオブジェクト

(= Sortableインター
フェースの派生クラスの
インスタンス)

ジェネリックなソートプログラム
Sortableなオブジェクトならば、
全て扱える

「型」に依存
しないコード

- 整数に対するソーティングのプログラム
- 実数に対するソーティングプログラム
- 文字列に対するソーティングプログラム
- Personオブジェクトに対するソーティング

ぜんぶ別々に書いていたら面倒くさい、
汎用性も低い、似たコードが散らばる

ポリモーフィズムを使えば、**ひとつのジェネリック**
(型に依存しない汎用的) プログラムを準備すればよい

以下の関数は型をprintfしてintの値を2倍にする関数である。同様の関数をshort, float, doubleでオーバーロードして定義し, main関数から呼び出した時の挙動を確認せよ。

```
int db(int value)
{
    printf("intです. ");
    return 2*value;
}
```

```
int main ()
{
    int a = 1;
    short b = 2;
    float c=3.f;
    double d=4.0;
    db(a);
    db(b);
    db(c);
    db(d);
    return 0;
}
```

- スライド中のサンプルコード(13,14,15ページ)を実行し，動作を確認せよ.
- Fruitsクラスの例で，自分で他の果物の派生クラスを作ってみてmその動作を確認せよ.

- Object クラスは抽象クラスであり、純粹仮想関数として `double area()` を持つ。
 - Objectクラスの派生クラスとして、Rectangle クラスと Circleクラスがある
 - 例えば、Rectangle クラスの場合は、対角の2点を保持、円なら半径や直径などを持つとする
 - area メソッドは、面積を返すメソッドである
- 以上のクラスについて動作テストするコードを書け

- (1) 次ページmin1.cpp を読み、中身を理解せよ。その上でプログラムを実行し、その動作を確認せよ。
- (2) ジェネリック関数find_min関数は不完全である。最小値の表示を行うように改良せよ。ただし、Double クラスに手を加えてはいけない。
- (3) Char (文字) クラスをComparableインターフェースの派生クラスとして実装せよ。
- (4) mainでCharクラスの配列に対して、find_minを適用してみよ。ただし、(1)で作ったfind_minに手を加えてはいけない。
- (5) 名前と年齢を含むPersonクラスを派生クラスとして定義せよ。年齢が最小のデータが表示されるプログラムを作成せよ。ただし、(1)で作ったfind_minに手を加えてはいけない。

```
#include <stdio.h>
class Comparable
{
public:
    virtual void print() = 0;
    virtual bool LTE(Comparable* a) = 0;
};
class Double: public Comparable
{
private:
    double val;
public:
    Double(double v) {val = v;}
    bool LTE(Comparable* a) {
        if (val <= (static_cast<Double*>(a)->val)) return true;
        else return false;
    }
    void print() {printf("%f\n", val);}
};
```

```
Comparable* x[10];
void find_min()
{
    for (int i = 0; i < 10; i++) {
        x[i]->print();
    }
}

int main()
{
    for (int i = 0; i < 10; i++) {
        x[i] = new Double((i-3)*(i-3)+1);
    }
    find_min();
}
```

- 本講義で紹介したポリモーフィズムは、実行時においてインスタンスの型によって、呼ばれるメソッドが選択される、**動的ポリモーフィズム**と呼ばれる
 - メリット：高い柔軟性
 - デメリット：関数呼び出しの際のオーバーヘッド、スタックを圧迫(vtable)
- 一方、コンパイル時にインスタンスの型により、メソッドが選択される場合、**静的ポリモーフィズム**と呼ぶ
 - メリット：高速、オーバーヘッドが少ない
 - デメリット：コンパイル時に型が決まっていないと使えない

- 静的ポリモーフィズムは、**テンプレート**で提供される。こちらで間に合うときは、テンプレートを利用する静的ポリモーフィズムでジェネリックなプログラムを書く（STLなどの利用も含め）
- 例えば、配列に含まれるオブジェクトの型がまちまちであるようなケースは実行時にメソッド選択する必要があるので、本講義で説明があった動的ポリモーフィズムを利用する。
- 次回はテンプレートについて説明する。