

Java演習

第8回

2024/6/5

横山大作

講義前・後の質問

- 横山まで
- dyokoyama@meiji.ac.jp

提出課題6: 円と三角形(再掲)

- CircleクラスとTriangleクラスがあった
 - Circle:
 - 色、半径を覚えている（フィールドがある）
 - showCircle()メソッドで表示できる
 - Triangle:
 - 色、向きを覚えている
 - printTriangle()メソッドで表示できる
- 統一的に扱いたい
 - どちらのクラスもFigureの一種である、としてほしい
 - printFig()メソッドで表示できるようにしたい

方針

- 共通しているのは？
 - `color`はどちらのクラスも同じもの（同じ型）
 - （表示の中にちょっと共通部分あるかも）
- 共通しているものを「親クラス」 **Figure**
- それ以外を「子クラス」 **Circle, Triangle**
- になるように継承関係を作ればよい

基本構造

```
class Figure {  
}  
  
class Circle extends Figure {  
}  
  
class Triangle extends Figure {  
}
```

- 継承関係はこうなるはずですね

基本構造

```
class Figure {  
    String color;  
}  
  
class Circle extends Figure {  
    int r;  
}  
  
class Triangle extends Figure {  
    string dir;  
}
```

- フィールドはこうなるはず
 - 共通するフィールドは親クラスへ

基本構造

```
class Figure {  
    String color;  
}  
  
class Circle extends Figure {  
    int r;  
    Circle(String color, int r) {  
        this.color = color; this.r = r;  
    }  
    void showCircle() { 省略  }  
}  
class Triangle extends Figure {  
    string dir;  
}
```

- この段階でもうコンストラクタやshowCircle()などは動いているはず
 - colorは子クラスでも使えるので、特に他を書き換える必要なし

コンストラクタ？

- ここで**Figure**にコンストラクタを作って困った人はいますか？
 - きちんと作らないと動かなかったかも
 - 説明不足でごめんなさい
 - 今日の講義の中で説明します

もう一つ要求

- `printFig()`メソッドで表示できるようにしてほしい
 - それぞれ、`showCircle()`とか`printTriangle()`みたいに違う名前だったのを統一したい
- オーバーライドを使う

オーバーライド

```
class Figure {  
    String color;  
    void printFig() { }  
}  
  
class Circle extends Figure {  
    int r;  
    Circle(String color, int r) {  
        this.color = color; this.r = r;  
    }  
    void showCircle() { 省略 }  
    void printFig() { showCircle(); }  
}
```

- 親クラスにオーバーライド「される」メソッドが必要
 - 子クラスには具体的なメソッドが必要
 - この例は、もともとあったshowCircle()を呼び出す形に書いた

- もちろん、`showCircle()`を残しておかなくても良い
 - `printFig()`だけにして、中身が`showCircle()`になっているのもよし
- `class Triangle`も同様に作る

さらなる共通化

- **printFig()**の中の、色に関する文字列表示部分は共通化可能かも
 - 例えば下の例とか

```
class Figure {  
    String colorStr() { return color + "色の"; }  
}  
  
class Circle extends Figure {  
    void printFig() {  
        System.out.println(colorStr() + "半径" + r + "の円");  
    }  
}
```

この資料の内容

- 継承の詳細
 - 機能、使い方
- インタフェースの詳細
 - 機能、使い方
- 提出課題7

継承の復習

- ちょっとずつ違うクラスの差分だけを書けるように作られた仕組み
- 親クラスと子クラスという関係がある
- 親の機能は子でも使える
 - フィールド・メソッドは継承される
- 子に特有の機能を子で定義する
 - フィールドの追加、メソッドの追加、メソッド乗っ取り
- メソッドを乗っ取る時は「オーバーライド」と呼ぶ
 - 参考：オーバーロードは引数が違うときの呼び方
- `extends` というキーワードで書く

継承されたときのインスタンス(p.361)

- 親クラスのインスタンスができる
- その外側に子クラスがかぶさる
ようなイメージ
 - もっと入れ子になっていくことも可能
- データ、メソッドは親クラスのものも見える
- 子供にない名前は親を探す
 - メソッドは、外側に見つかったらそれ以上中を探さないから、上書きされたように見える

親クラスへのアクセス(p.363)

- 親クラスのメソッドは消えていない
- 親の処理を明示的に呼び出すことができる
- `super`キーワード

superキーワード(p.365)

- 親クラスを明示的に使いたいときに記述する
 - 親クラスのメソッドを使いたい場合など
- 自分のクラスを示す**this**と同等

```
class Hero extends Person {  
    void attack() {  
        super.attack(); // Person.attack() を呼んでいる  
        super.attack(); // Person.attack() を呼んでいる  
    }  
}
```

親のメソッドを
ちよっとだけ変
更するときとか

- 例えば**Person.attack()**が変更された時でも、**Hero.attack()**は「その」2倍ダメージ、と正しく動く
- 子クラスを示すものはあるか？
 - ない
 - コードを書いているときに親クラスは必ずあるが、子クラスは将来できるかどうかはわからない

継承とコンストラクタ (p.367)

- インスタンスは内側に親クラスのインスタンスを含んでいる感じだった
- コンストラクタは内側から順番に動いていく
- `Circle()`のコンストラクタの直前で実は`Figure`のコンストラクタが動いている

```
class Figure {  
    String color;  
}  
  
class Circle extends Figure {  
    int r;  
    Circle(String color, int r) {  
        this.color = color;  
        this.r = r;  
    }  
}
```

この時点では
`Figure`が出来上がっている

親クラスのコンストラクタ

- 親クラスのコンストラクタは**super(引数)**で呼べる
 - 自分のコンストラクタ**this(引数)**と似てる
 - 子のコンストラクタの中でだけ呼べる
 - 変なタイミングでは呼べない
- 何も書かなければ、暗黙のうちに**super()**（デフォルトコンストラクタ）がコンストラクタの先頭で呼ばれている

```
class Circle extends Figure {  
    int r;  
    Circle(String color, int r) {  
        super();  
        this.color = color;  
        this.r = r;  
    }  
}
```

自動的に呼ばれていた

こうなるとエラー

- なぜだか考えてみよう
- Circleのコンストラクタの頭には `super()` がある
- Figureに引数なしのコンストラクタある？

```
class Figure {  
    String color;  
    Figure(String color) {  
        this.color = color;  
    }  
}  
  
class Circle extends Figure {  
    int r;  
    Circle(String color, int r) {  
        this.color = color;  
        this.r = r;  
    }  
}
```

正しい親クラスのコンストラクタ呼び出し

- **Figure**は引数**1**個のコンストラクタで初期化しないとダメ
- **Circle**では親(**Figure**)を正しく初期化する必要がある
 - **super(色)**を正しく呼び出す

```
class Figure {  
    String color;  
    Figure(String color) {  
        this.color = color;  
    }  
}  
  
class Circle extends Figure {  
    int r;  
    Circle(String color, int r) {  
        super(color);  
        this.r = r;  
    }  
}
```

colorの初期化は親クラスのコンストラクタがやってくれている

コンストラクタのその他

- どのクラス階層でも必ず何かコンストラクタが呼ばれる
- `super()`はコンストラクタの先頭にしか書けない
 - 引数があってもなくても
 - 親インスタンスが完全に出来上がってから子インスタンスの初期化が始まる
- コンストラクタは継承されない
 - 親クラスにあるコンストラクタと同じ引数なんだけど...という場合にも、継承されていないので子クラスでも定義する必要あり

正しい/間違った継承 (p.373)

- 共通部分があれば何でも継承関係にすればよい
わけではない
- 継承は「～の一種である」(is-a関係)という関係にのみ使うことをお勧めする
 - class 座標 { int x; int y; }
 - class 勝敗 {int x; int y; }
 - (x, y)のペアなんだから継承関係にしたらまとまる？
 - そんなわけではない
- 現実のモデルだと思って、「～の一種である」関係だと思える場合なら継承してよい

ポリモーフィズム

- オーバーライドの例にあったように「インスタンスの型によって実際の処理を変える」こと
 - 例外処理もそうでした
- オブジェクト（インスタンス）が処理を決めるのだ、というオブジェクト指向の考え方

```
Figure fig = getfigure();  
fig.calcArea();
```

本当は何型？○？△？

ポリモーフィズム

扱いは親クラス
で一般的に

```
double allArea(Figure[] figs) {  
    double s = 0;  
    for (int i = 0; i < figs.length; i++) {  
        s += figs[i].calcArea();  
    }  
    return s;  
}
```

インスタンスに
お任せ

- 実際に呼ばれるのは、インスタンスにくっついたメソッド
 - ○だの△だのの違いはインスタンスの方が知っている

ポリモーフィズムじゃない書き方だと

```
double allArea(Figure[] figs) {  
    double s = 0;  
    for (int i = 0; i < figs.length; i++) {  
        if (figs[i] instanceof Circle) {  
            s += figs[i].circleArea();  
        } else if (figs[i] instanceof Triangle) {  
            s += figs[i].triangleArea();  
        }  
    }  
    return s;  
}
```

使う側が**fig**が何者なのかを判断して、使うべきメソッドを変えている

- クラス増えていったら大変！

(実際にはキャストが必要)

ポリモーフィズムじゃない書き方だと（別パターン）

```
double allArea(Circle[] circles, Triangle[] triangles) {  
    double s = 0;  
    for (int i = 0; i < circles.length; i++) {  
        s += circles[i].circleArea();  
    }  
    for (int i = 0; i < triangles.length; i++) {  
        s += triangles[i].triangleArea();  
    }  
    return s;  
}
```

型ごとにリストを分けて
いったら当然まとめ
ては扱えない

- これもクラス増えていったら大変！

クラスを設計するときの考え方2通り

- 共通部分をくくりだしたものが親
- 何か改造を加えたものが子
- どちらの場合もあり得る

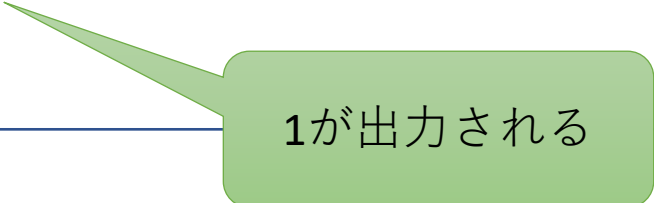
設計は想定力

- 共通部分をくくりだすところでも既に設計想定がある
- 「いろいろな図形を、まとめて扱おう」
- 「色へのアクセスは必要そうだ」
- 「面積も計算する要求があるな」
- 使われ方を想定しないとクラス設計はできない
 - 正しい設計は1つとは限らない

フィールドの再定義

- フィールドも実は継承の時に同じ名前を付けられる
- しかし、メソッドのように乗っ取りができていないわけではない
 - 「実際の型」ではなく、「変数の型」によってどのフィールドになるかが決まっている
 - ポリモーフィズムできない

```
class A {  
    int f = 1;  
}  
  
class B extends A {  
    int f = 2;  
}  
  
class Test {  
    public static void main(String[] args) {  
        A p = new B();  
        System.out.println(p.f);  
    }  
}
```



1が出力される

こんなこともできる

```
class A {  
    int f = 1;  
}
```

```
class B extends A {  
    String f = "abc";  
}
```

型が違うけど同じ
名前のフィールド
作れてしまう

```
class Test {  
    public static void main(String[] args) {  
        B b = new B();  
        System.out.println(b.f);  
        A a = b;  
        System.out.println(a.f);  
    }  
}
```

"abc"が出力される

1が出力される

- 要するに、**B.f**は**A.f**とは関係ない
 - 変数の型の方が優先されてアクセスされる

なぜ？

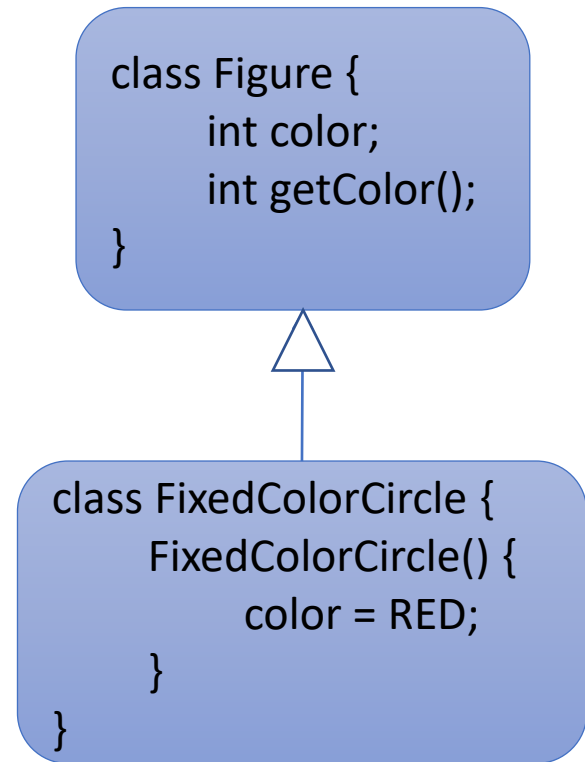
- Javaではそういう決まりだから...
- 変数へのアクセスが「変わるかも」と思うのはさすがに無駄が多くない？

```
class A {  
    int f = 1;  
    void print_f() { System.out.println(f); }  
}  
  
class B extends A {  
    String f = "abc";  
}
```

このfが変わるかもと備えておくことはしない、とJavaでは決めた

やりたかったこと？

- FixedColorCircleがFigure.colorを書き換えれば良いのでは？
 - コンストラクタなどで
- 同じフィールドを定義することは「シャドウ変数」と呼ばれ、推奨されない
 - 混乱のもと



staticメソッドの再定義

- staticメソッドもオーバーライドされない
 - 単に別物ができるだけ
- 同様に推奨されない

```
class Hero extends Person {  
    static void m() {  
        System.out.println("m()をオーバーライドしたつもり");  
    }  
}  
  
class Test {  
    public static void main(String[] args) {  
        Person p = new Hero();  
        p.m();  
    }  
}
```

オーバーライド
前の**Person.m()**が
呼ばれる

@Override アノテーション

- オーバーライドするよ、と明示的にコンパイラに教える書式
 - Java5以降
 - スペルミスなどで該当するメソッドがないとコンパイラがエラーで教えてくれる
 - **static**メソッドでオーバーライドできないチェックもしてくれる

```
class Child extends Parent {  
    @Override  
    void func() {  
        ...  
    }  
}
```

継承の制限(p.358)

- 意図的にクラス等の継承を禁止できる
- クラスに**final**を付けると子クラスが作れなくなる
 - `final class Ape {}`とか。`extend`できなくなる。
- メソッドに**final**を付けると、そのメソッドがオーバーライドできなくなる
 - `final void sleep() {}`とか。
- 参考：フィールドに**final**を付けたときは「変更不可」になるので少し意味が違う
 - 定数値を作るときに使う

オーバーライドの制限(p.359)

- アクセス修飾子についてちょっと復習 (p.472)
 - `protected` の意味が分かるようになったはず
- 継承でアクセス範囲を狭くすることはできない
 - 親で使えていたメソッドを使えなくすることは許さない
 - 例：親で指定なしのメソッド
子供では指定なし, `protected`, `public`にはできる。
`private`にはできない。
- `final`キーワードが付くと以降オーバーライドできない
- `static`修飾子がからむとオーバーライドできない

ポリモーフィズムの実現方法

- 継承
 - 親クラスのメソッドを子クラスでオーバーライドしているとき
- インタフェース
 - 「このメソッドが必ずあるはず」と宣言されているとき

インタフェース

- このインタフェースを「実装」(implements)しているクラスは必ずこのメソッドを持つ、という宣言
- 例: 図形は必ず面積を返す関数 `int calcArea()` を持つ
- 例: 比較できるものは必ず `int compareTo()` を持つ
 - Comparable インタフェース

継承とインタフェース

- どちらも同じようなことができる
- 継承は「インスタンスが作れるもの」の関係
- インタフェースは「満たすべき性質」

継承のイメージ

- リンゴを継承したバウムクーヘン
 - 中身がある



インタフェースのイメージ

- 一口香（長崎名物）
 - ガワだけ



継承とインタフェース

- 継承されたものはインスタンスが作れる
 - `Figure fig = new Circle();`
 - 子の型のインスタンスを親の型の変数で指せる
- インタフェースはインスタンスは作れない
 - `interface HasArea;`
 - `HasArea fig = new HasArea();` とはできない
 - `HasArea fig = new Circle();` はできる
 - インタフェースは変数の型にはなり得る

継承の気持ち

- 世の中のものはすべて階層的に説明できる！
- オントロジー
- 20の質問(20の扉)
 - <http://www.20q.net/index.html>
 - 「それは動物ですか？」 「水の中にいますか？」 みたいな質問から物事を当てる
 - 参考: 広辞苑は25万語

実際には...

- きれいに階層化できるとは限らない
 - Treeになってない
- そもそも階層化とは異なる観点の整理がしたい
 - 例: 「順序関係があるもの」という観点
 - StringもIntegerもStudentsもすべてComparable
 - クラスの階層とはちょっと違う切り口で切りたい
- という要求を満たすためにInterfaceが用意された

使い分け

- 現実世界のモデルとして階層的に整理できると
思ったら継承
 - is-a関係か？
- クラス階層とは異なる性質だと思ったらインタ
フェース
- 絶対的な正解はない

インタフェース型 (p.411)

```
interface HasArea {  
    double calcArea();  
}  
  
class Arrow implements HasArea {  
    // 三角形部分と四角形部分がある...  
    double calcArea() { return 三角形部分面積 + 四角形部分面積; }  
}
```

- 継承とは別の方法でポリモーフィズムできる
 - HasArea型の変数 が作れる
 - 親クラス型で扱うのではなく、インタフェース型で扱えばよい

インタフェースの詳細

- 未実装はエラー
 - `implements`するクラスではすべてのメソッドを作らないとコンパイルできない
- `final static`なフィールドが定義できる
 - 定数を使いやすく提供
- `public abstract`が暗黙の裡に付く
 - `abstract`は「ここでは実体がない」という宣言
- 使うときは複数のインタフェースを**`implements`**することが可能
 - クラス継承だと親は**1**つだけなので、より柔軟

参考: インタフェースのデフォルト実装

- インタフェースがメソッドの実装を持てる
 - `static`メソッドも持てる
- Java8より
- 互換性のため
- やや細かい使い方のため という要素が強い
 - そんなに使う場面はないかも
 - 普通にインタフェースを実装した抽象クラスを作れば良い

抽象クラス (p.400)

- インタフェースは「実装を持たない、メソッドのシグネチャのみ宣言する」
- 同様のことがクラス内でも書ける
 - クラスとメソッドの宣言部分に**abstract**キーワードを付ける
- ある意味、インタフェース付きのクラスのようなもの

ちょっと気持ち悪かったところ

```
class Figure {  
    int x;  
    int y;  
    int color;  
    int getColor() { return color; }  
    double calcArea() { return 0.0; }  
}
```

```
class Circle extends Figure {  
    double r;  
    double calcArea() { return r * r * Math.PI; }  
}
```

```
class Triangle extends Figure {  
    double width;  
    double height;  
    double calcArea() { return width * height / 2; }  
}
```

なぜ0を返す？
このコードは呼ば
れない（呼ばれる
べきではない）の
になぜ作らなく
ちゃいけない？

抽象クラスなら

```
abstract class Figure {  
    int x;  
    int y;  
    int color;  
    int getColor() { return color; }  
    abstract double calcArea();  
}
```

実装なし。
すっきり。

```
class Circle extends Figure {  
    double r;  
    double calcArea() { return r * r * Math.PI; }  
}  
  
class Triangle extends Figure {  
    double width;  
    double height;  
    double calcArea() { return width * height / 2; }  
}
```

抽象クラス

- インスタンスを作ることはいできない
 - インタフェースがくっついているから
 - ガワしかない部分がある
- 継承して、**abstract**なメソッドの中身を実装したら初めてインスタンスが作れるようになる
 - 具体クラスにしたら**new**できる

ポリモーフィズムの方法3通り

- インタフェースを使う
- 抽象クラスを使う
- 具体クラスを継承して使う
 - これはこの講義の最初に見せた図形クラスの例
- 抽象クラスやインタフェースは、「ポリモーフィズムで動いてるよ、メソッドは差し替えられるよ」という意図をわかりやすくしている
 - あと、さっき説明した「必要がない親クラスのメソッド」の気持ち悪さを排除している

継承とキャスト(p.450)

- 子供は親型の変数に安全に（キャストせずに）代入できる
- 親を子供型変数に代入するのはキャストが必要
 - 安全にできるかどうかはわからない
 - 実行時例外の可能性あり
- 兄弟への変換はキャストしても無理
 - 論理的にあり得ないはずだから
- 「子」は「親」の一種、という関係にあることを注意して、考えてみよう
 - ペンギンは鳥である、鳥はペンギンでは（必ずしも）ない
 - ペンギンも鳩も鳥だが、ペンギンは鳩ではない

java.lang.Object (p.499)

- Javaのオブジェクトはすべてこのクラスを継承
- オブジェクトの基本機能がある
- 主なメソッドの表を眺めておきましょう
 - equals(), hashCode(), toString()などがある
- System.out.println()にオブジェクトを渡すと、toString()が呼ばれる
 - デフォルトはクラス名やメモリアドレスが表示されるだけ
 - 自作クラスでオーバーライドすれば読みやすくなる

提出課題7: 商品価格

- 商品を表すクラス **Food, Book** がある
 - いずれも名前と価格を保持
- 本と食品をセット売りすることがある
 - 価格は合計から **100円** 引き
- 色々な商品やセットをリストにして表示したい

方針

- 商品などのクラスとは別に、「価格表示ができるもの」というインタフェースを考える
 - interface PriceTag
 - できることは
 - String tagstr() // 表示のための商品名を返す
 - int price() // 表示のための価格を返す
- 商品などはPriceTagインタフェースを実装する
- PriceTagのArrayListで商品などを管理する
- セット商品はGoodsSetというクラスで表現する

課題の要求

- ひな型ソースには**Food**と**Book**がある
 - `tagstr()`と`price()`も用意されている
- `ShopTest.main()`は
 - カップラーメン、180円
 - Java演習の本、2000円
 - その2つのセットの3つをリストに入れ、順番に表示することを考えている
- 必要な他のクラスなどを作り、**Food**なども適切に書き換えて、動作するようにせよ

細かい要求

- セットの商品の文字列は
「XXX(食品名) と YYY(書籍名) のセット」
とする
- セット商品の価格は合計価格から**100円**引き

提出物

- 提出物はShopTest.java
 - 先頭に「**組番号、名前**」と、出力された文字列をコメントで記入
 - 採点ミスを減らすための用心。ご協力ください。
 - package javalec7 とする
- ✕切は6/11(火) 17:00

ヒント

- インタフェースを使うことを考えましょう
- インタフェースは「このメソッドがあること」を要求します
- インタフェースの要求するメソッドは**public**でなければなりません
 - インタフェースに書いてなくても暗黙的に要求されます
 - つまり、実装側のクラスで、その名前のメソッドの最初には**public**を付ける必要があります

この資料の内容

- 継承の詳細
 - `super`キーワード、コンストラクタの詳細
- インタフェース
 - 実装がないもの、型として使えるもの
 - メソッドの存在を保証
 - 抽象クラスもある
- `java.lang.Object`
- 提出課題7