Java演習
第7回

2024/5/29

横山大作

講義前・後の質問

- 横山まで
- dyokoyama@meiji.ac.jp

提出課題5(再掲)

- 文字列をため込んでいるクラスPartsがある
  - 文字列の個数はint Parts.NUM
  - 文字列を取り出すにはString Parts.item(int i)
- この文字列から、任意の2つを取り出してくっつけた文字列を作る
  - 同じ文字列をくっつける場合も含むことにする
- くっつけた文字列がParts.NUM * Parts.NUM個できる
- くっつけた文字列の中で、同じ文字列になっている（＝文字の並びが同じ）ものはあるか？あれば、1行に1つずつ表示せよ
  - ただし、3回以上重複していることはないとする
  - 文字列の全体が一致することが必要。一部が一致するものは含まない

ヒント(再掲)

- 文字列の連結は+でしたね
- 最初に、2個の文字列をくっつけた文字列の配列を作ってしまうと簡単そう
- そのあとで、配列の中の2個が等しいかチェックする
- 「等しい」の意味に注意！

- 答えは手作業でも十分わかるはずなので、正しい答えが出ているか、きちんとチェックしよう

解答例

```
public class StringTest {
    public static void main(String[] args) {
        String[] ps = new String[Parts.NUM * Parts.NUM];
        int p = 0;
        for (int i = 0; i < Parts.NUM; i++) {
            for (int j = 0; j < Parts.NUM; j++) {
                ps[p] = Parts.item(i) + Parts.item(j);
                p++;
            }
        }

        for (int i = 0; i < ps.length; i++) {
            for (int j = i + 1; j < ps.length; j++) {
                if (ps[i].equals(ps[j])) {
                    System.out.println(ps[i]);
                }
            }
        }
    }
}
```

もちろん

- ps[i*Parts.NUM + j] = Parts.item(i) + Parts.item(j);
  でも良いですね。
- 前半と後半でループの変数を変えてもいいですね
  - 例えば後半をp,qの組にするとか
  - 良く似ているのに意味が違うので
- for (int i = 0; i < ps.length - 1; i++) {
  for (int j = i + 1; j < ps.length; j++) {

のように、ループの範囲をあらかじめ狭くしても良いですね。

whileループでの書き方

```
for (int i = 0;
     i < Parts.NUM;
     i++) {

    // 何かやるべきこと

}
```

```
int i = 0;
while ( i < Parts.NUM )
{

    // 何かやるべきこと

    i++;
}
```

- 機械的に書き換えられる

whileループでよくあるミス

今回の課題には関係しないけど

```
for (int i = 0;
     i < Parts.NUM;
     i++) {

    // 何かやるべきこと
    if (XXX) { continue; }

    // 何かやるべきこと2

}
```

```
int i = 0;
while (i < Parts.NUM)
{

    // 何かやるべきこと
    if (XXX) { continue; }

    // 何かやるべきこと2

    i++;
}
```

- 右は無限ループする可能性がある
  - i++忘れ

今回の内容

- オブジェクト指向
  - カプセル化
  - アクセス修飾子
- 継承
  - 考え方
    - なぜ必要か、何をしようとしているのか
  - 基本的な書き方

- 提出課題6

カプセル化(第13章)

- オブジェクト指向の大事な役割に「内側を使わせない」というものがある
  - 内側の情報を勝手に別の人が書き換えると困る
  - 内側の実装を別の人が使っていると修正できない
  - 「カプセル化(encapsulation)」と呼ぶ
    - カプセルの中に閉じ込めて、外に見せない（隠蔽する）
- アクセス修飾子はカプセル化を支援するための機能

アクセス修飾子(p.472)

- public
  - すべてのクラスからアクセスできる
- protected
  - サブクラスからアクセス可能（「継承」でまた詳しく説明します、今は忘れても可）
- 指定なし(デフォルト)
  - 同じパッケージ内ならばアクセスできる
- private
  - 同じクラスからしかアクセスできない

アクセス修飾子を付けるところ

- クラス
- フィールド
- メソッド

- いずれも、その定義の前に付ければよい

```
public class Test {
    private int x;
    private void function(String arg) { …
```

- クラスはちょっと違う(後述)

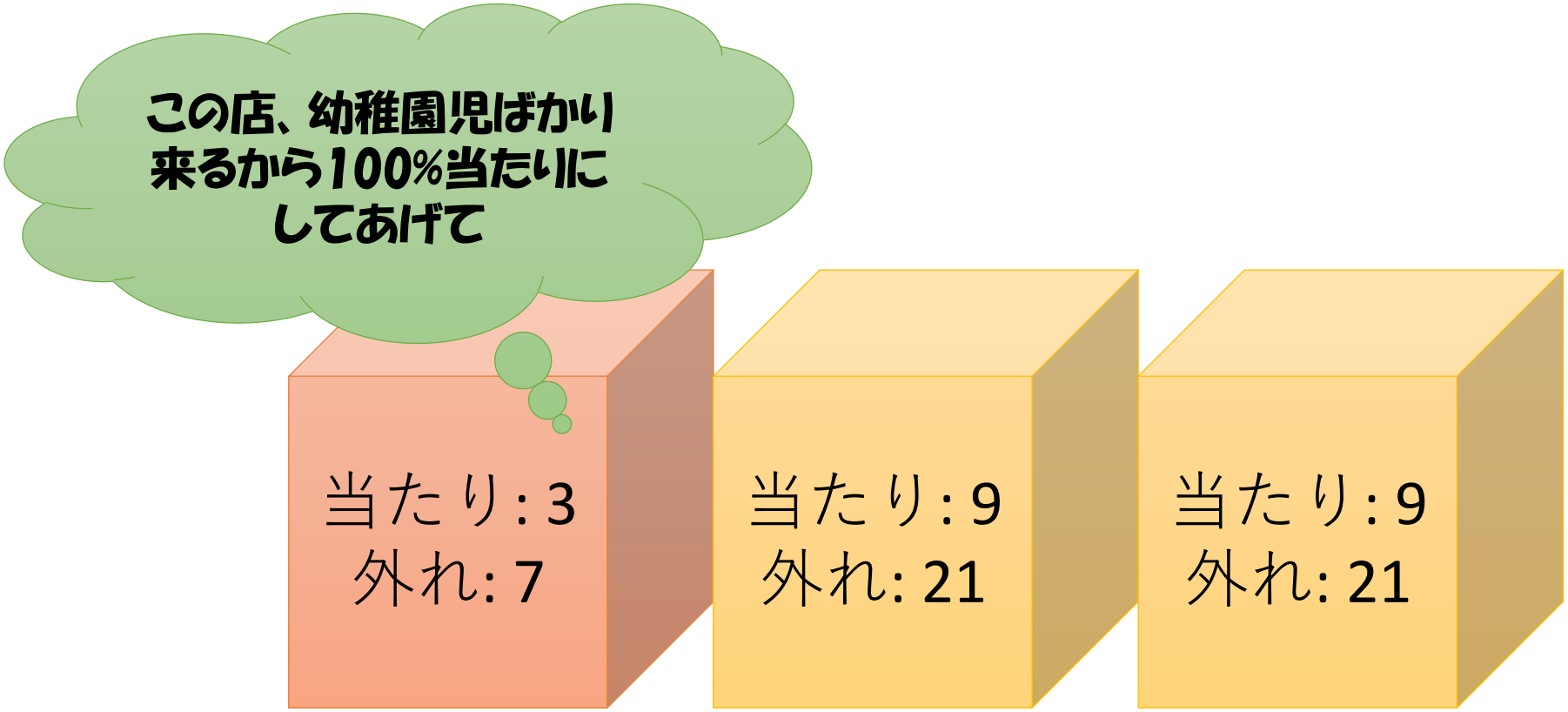
フィールドは基本的にprivateがおススメ

- カプセル化のため
- 値にアクセスするために”setter” ”getter”を作るのが普通（p.477)
  - 面倒であっても利点もある
    - setする前に値が正しいかチェックできる(validation)
- eclipseでもsetterなどを自動的に作る機能アリ

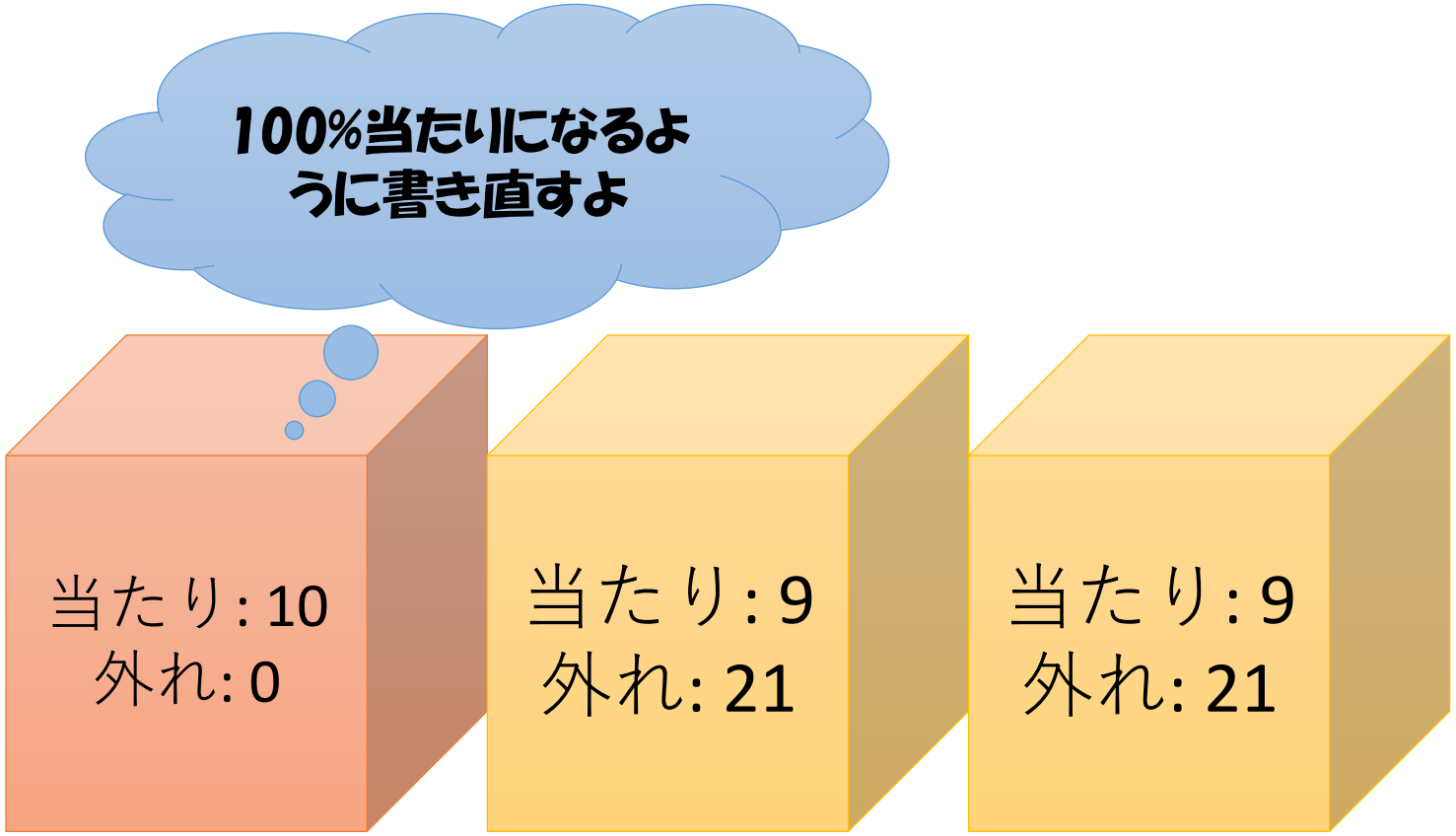
```
class Person {
  private String name;

  // setter
  public void setName(String name) {
    this.name = name;
  }

  // getter
  public String getName() {
    return name;
  }
}
```

例: くじ引きシステム

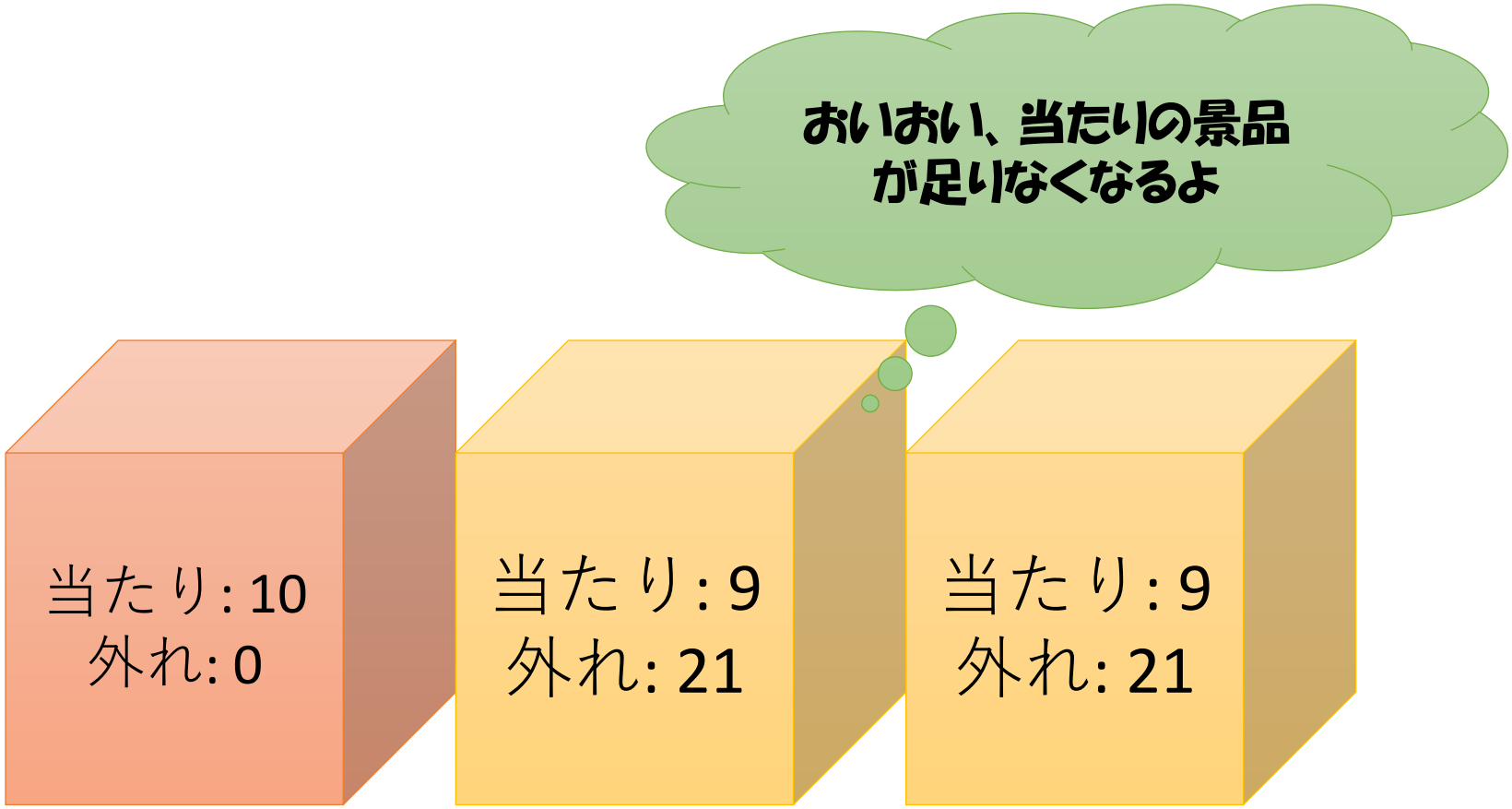
- 商店街のくじ引きシステム
- 店ごとに当たりくじを公平に分けた



設定変更



設定変更

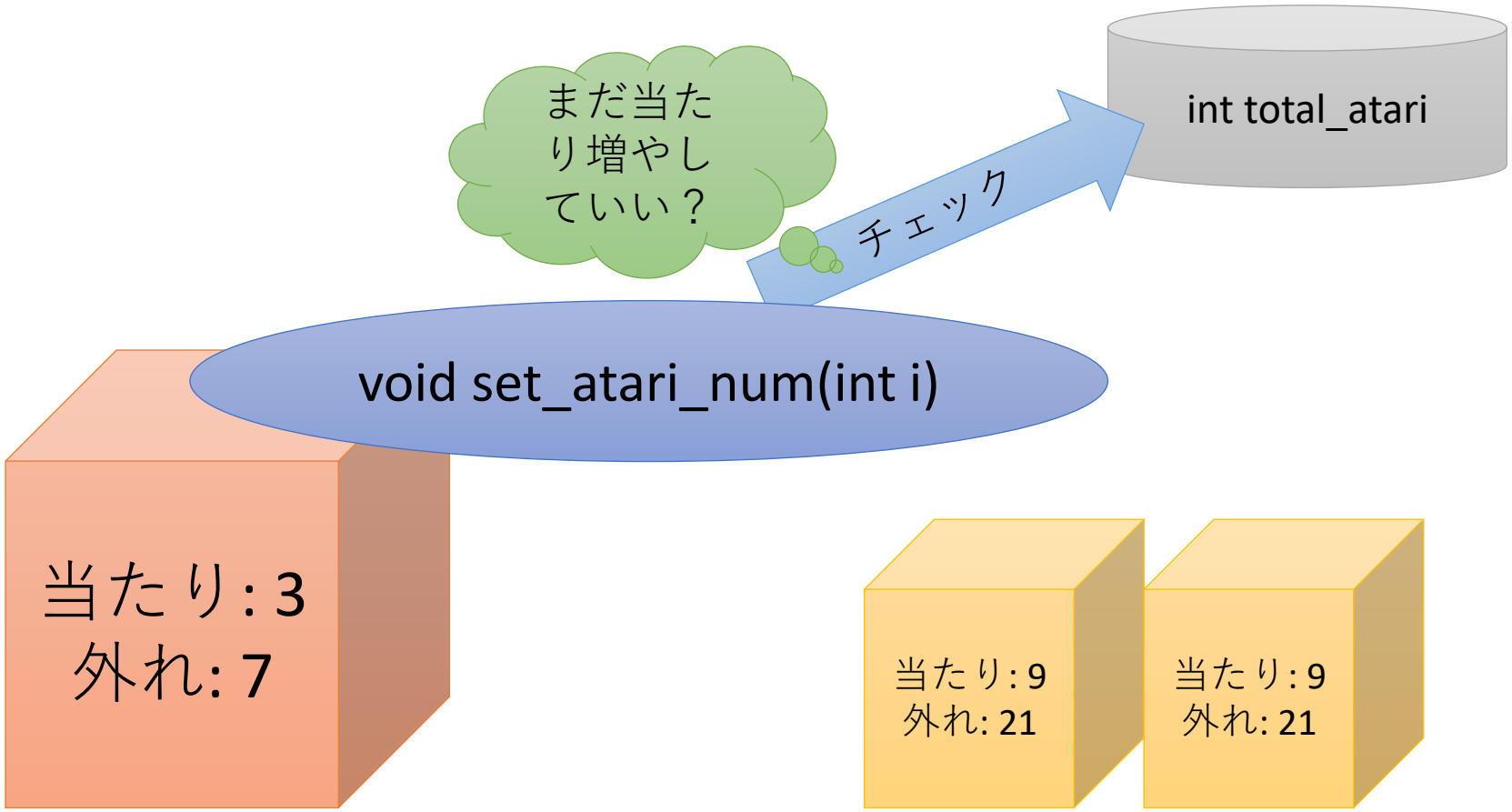


言いたいこと

- データを勝手に書き換えると、裏側にあった意図が崩れるかもしれない
- 特に、複数人で作業しているときは起きがち

- これを防ぐには、責任の境界を決めることが大事
  - くじの内訳は箱がきちんと管理する。外の人は勝手に触らない。
  - 変えたいときは変える方法を「箱が」準備するので、それを使う

設計例


まだ当たり増やしていい？
チェック
int total_atari
void set_atari_num(int i)
当たり: 3
外れ: 7
当たり: 9
外れ: 21
当たり: 9
外れ: 21

カプセル化の効果

- 責任の境界を明確にできる
- 責任を超えて情報を書き換えることを防ぐ

- 今回の場合、どこがprivate?

  - 当たり、外れの本数はprivate
  - set_atari_num()はpublic
  - 外の人ができることを制限している

クラス設計

- 今回の場合、他の整理の仕方も考えられる
  - set_atari_num(int)ではなく、set_atari_ratio(double)
    - 当たり確率を設定
  - total_atariは作らない、全当たり本数を足し算して計算
    - どこかを増やすときにはどこかを減らす
- こういう「どう整理するか」を考えるのが「設計」
  - 自由度がある
  - 外の人に使いやすく、かつ変なことをされないように守れる設計が良さそう
  - カプセル化はそれをサポートする手段

クラスにもアクセス修飾子付けられる

- パッケージ外から見えるかどうか
  - publicと「なし（パッケージprivate）」の２択のみ

- 再利用する、という観点からはpublic付けておきたい
  - まぁ一時的なクラス、ここでしか使わないクラスなら「なし」でもいいかな

継承

オブジェクト指向の例

- 何が似ている？

みんなミュージシャン

- それ以外の共通点はないですね

データ

ミュージシャン

- 名前
- 楽器名
- ギャラ

1人1人違うデータを同じ枠組みで整理するのがクラスだった

ちょっと増えた

- グループはある？

ちょっとした違い

- 演奏するか、踊るか

データ

ちょっと違うクラス

ミュージシャン

- 名前
- 楽器名
- ギャラ

ダンサー

- 名前
- ギャラ
- ジャンル



ちょっと違うクラスをまとめたい

- ギャラの計算をそれぞれのデータごとに別々に書くのは無駄
  - どうせ「ギャラ」というフィールドを読むだけ
- どうまとめる？

最小公倍数的？に考える？

ミュージシャンか
ダンサー

- 名前
- 楽器名
- ギャラ
- ジャンル

できなくはなさそう

```
for (a : alist) {
    total += a.guarantee;
}
```

名前	楽器名	ギャラ	ジャンル
葉加瀬太郎	バイオリン	X	
ユザーン	タブラ	Y	
レキシ	ボーカル	Z	
パパイヤ鈴木		W	J-POP

違う処理がしたいときは？

名前	楽器名	ギャラ	ジャンル
葉加瀬太郎	バイオリン	X	
ユザーン	タブラ	Y	
レキシ	ボーカル	Z	
パパイヤ鈴木		W	J-POP

```
for (a : alist) {
    if (a.instrument != null) {
        // ミュージシャン
        total += a.guarantee;
    }
    if (a.genre != null) {
        // ダンサー
        ...
    }
}
```

時間がたったらどうなる？

- プログラムを継続的に開発していくとしたら、何か困ったことが起きないだろうか？
- 考えてみよう

- データが増えていくと？
- 自分がプログラムを忘れると？

```
for (a : alist) {
    if (a.instrument != null) {
        // ミュージシャン
        total += a.guarantee;
    }
    if (a.genre != null) {
        // ダンサー
        ...
    }
}
```

- ミュージシャンにもジャンル入れたくなったら？
- 別のジャンル（タレントとか？）入れたくなったら？
- データの列が増えていったら？
  - ほとんど空の表にならない？

固まりごとに見分けたい

名前	楽器名	ギャラ
葉加瀬太郎	バイオリン	X

名前	楽器名	ギャラ
ユザーン	タブラ	Y

名前	楽器名	ギャラ
レキシ	ボーカル	Z

名前	ギャラ	ジャンル
パパイヤ鈴木	W	J-POP

こうしたくはない

名前	楽器名	ギャラ
葉加瀬太郎	バイオリン	X
ユザーン	タブラ	Y
レキシ	ボーカル	Z

名前	ギャラ	ジャンル
パパイヤ鈴木	W	J-POP

```
for (a : musician) {
    // ミュージシャン
    total += a.guarantee;
}
for (a : dancer) {
    // ダンサー
    total += a.guarantee;
}
```

- 同じことを何度も書く必要あり
- タレントという種類が増えたらどうする？

こうしたくもない


```
for (a : alist) {
    if (aがミュージシャン) {
        演奏してもらう
    } else if (aがダンサー) {
        ダンスしてもらう
    }
}
```

- 毎回データの「外で」種類を意識したくない
  - 種類が増えたらあちこち直さないといけない

オブジェクト指向

- 処理が同じところは共通化したい
- 処理が違うところだけ部分的に変えたい
  - 変えるのは外の人は意識したくない
  - データが責任もって変えられれば良い

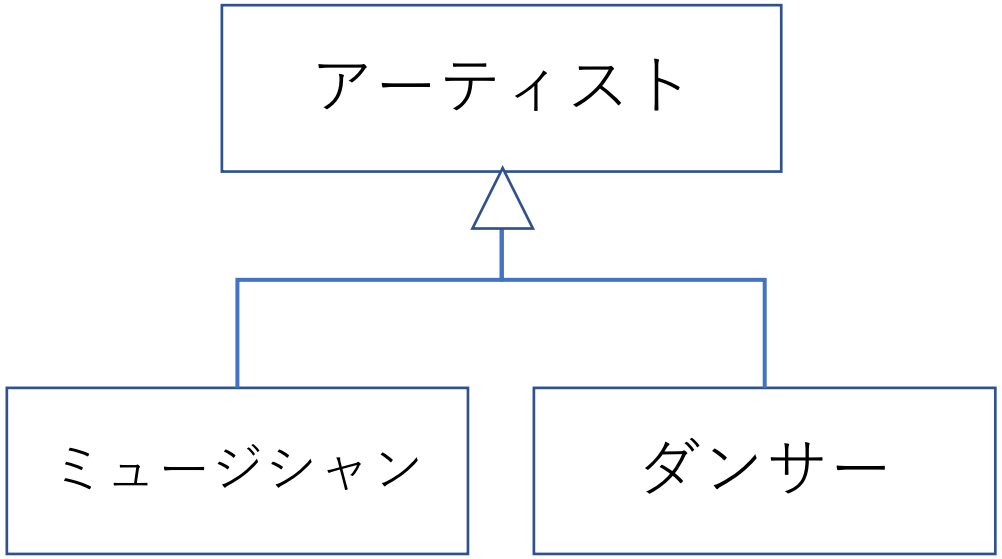
```
for (a : alist) {
    aにplayしてもらう
}
```

クラスのまとめ方：継承

- クラス分けにはまとまりがある
- 共通部分に名前を付ける

クラスのまとめ方：継承

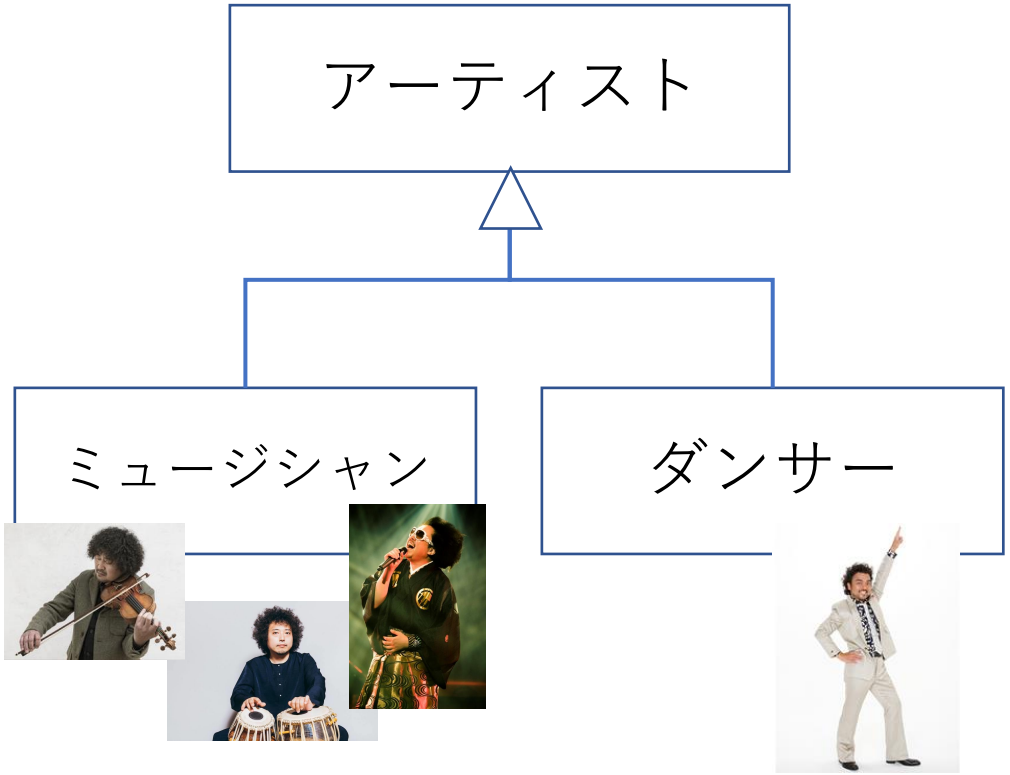
- クラスで共通する部分を「親クラス」
- 違いがあるほうの分け方を「子クラス」
- **「子は親の一種」という関係が成立**
  - ざっくり言うと親、細かく分けると子
- 親を**「継承」**して子クラスを定義する
  - 基本的には親と同じ
  - 違うところだけを書くよ



矢印の向きが逆に見える？
実はこう書くのが決まりになっています(p. 354)

クラスとインスタンス

- インスタンスはどこか1つのクラスに属する
  - newされたときに決まっている
- プログラム中で「ここでは～と扱うよ」という見方が様々に変わる
  - ここではアーティストとしてまとめて扱うよ、と言われたらインスタンスはそのようにふるまう



継承の書式(p.374)

- extends

```
class Artist {
    String name;
    int guarantee;
}

class Musician extends Artist {
    String instrument;
}

class Dancer extends Artist {
    String genre;
}
```

- 同じ部分は親に1度書くだけ
- 子供は違う部分のみを書く

まとめたいもの

- ここまではデータについてを中心に考えていた
  - フィールド
- メソッドだってまとめたい
  - 名前を表示するメソッド、ギャラを出力するメソッドは全部のクラスで共通

継承

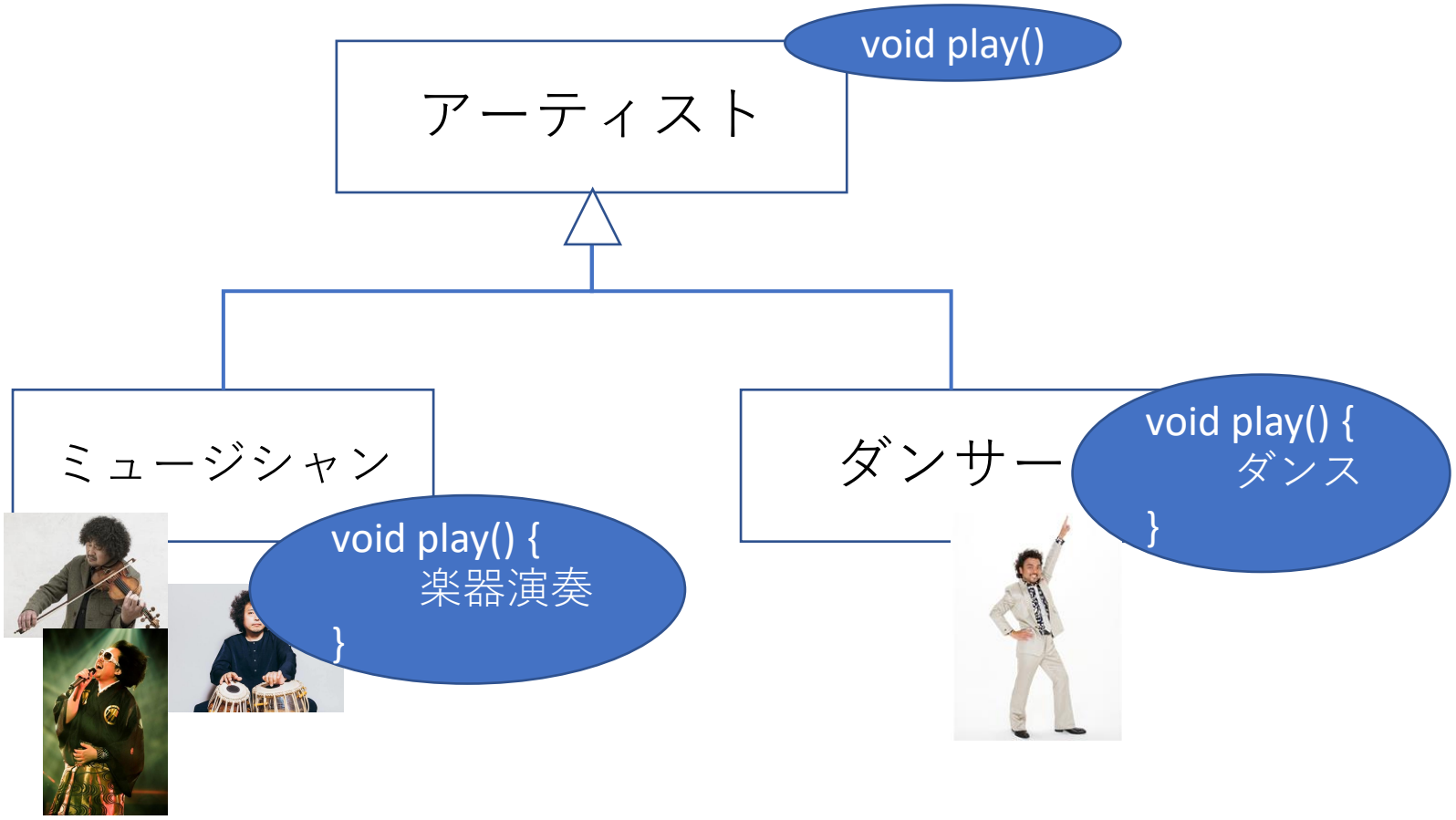
- 親クラスを継承した子クラスは、親クラスの
  - フィールド
  - メソッド

  をそのまま使える
- 子クラスにフィールドやメソッドを書くと、それが追加される
  - 子クラスだけで使えるフィールドやメソッド

```
class Artist {
    String name;
    int guarantee;
    void showname() { 名前を表示する }
}

class Musician extends Artist {
    String instrument;
    void musicplay() { 演奏する }
}

class Dancer extends Artist {
    String genre;
    void dance() { ダンスする  }
}
```

継承されるもの

- 子クラスで新しい名前のフィールドやメソッドは「追加」される
- 子クラスで「同じ」名前のメソッドを定義すると、「上書き」することになる
  - オーバーライド
  - フィールドは上書きではなく、あまりうれしくない動きになるので使わない（詳細は後日）
- メソッドを呼ぶと、「今のインスタンスに適切なもの」が自動的に選ばれることになる
  - メソッドはインスタンスにくっついているから
- 詳細は後日



- インスタンスが「本当は何者なのか」によってplay()は楽器演奏したりダンスしたりする

オーバーライド

- 違う部分を切り替えて**同じように**使いたいとき
  - play()と言われたら違うことがしたい
- 同じ名前のメソッドをそれぞれのクラスで定義する

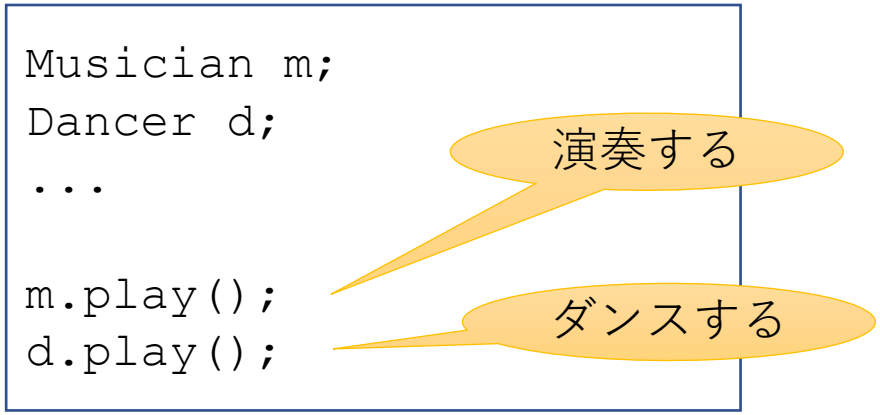
```
class Artist {
    String name;
    int guarantee;
    void play() { (何もしない) }
}

class Musician extends Artist {
    String instrument;
    void play() { 演奏する }
}

class Dancer extends Artist {
    String genre;
    void play() { ダンスする  }
}
```

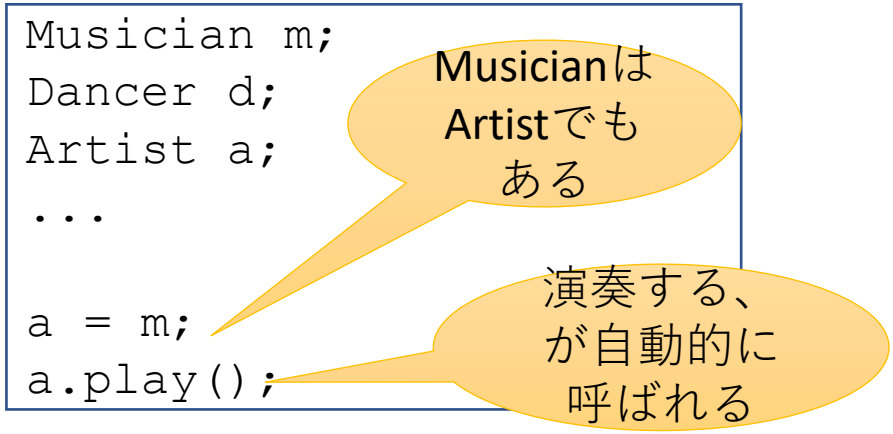
オーバーライド

オーバーライド

使う側

- 同じ名前のメソッドがオブジェクトごとに違ったふるまいをする
  - 上の使い方ではあまりうれしくない
- 同じ変数に代入されるものが色々ある、というシチュエーションだとありがたみが出てくる
  - ArtistはMusicianかDancerかはわからないが、勝手に正しいメソッドが呼ばれる





代入

- 自然に考えられる型の違いは何も対応しなくても代入できた
  - 「double に intの値を代入」は可能
- 継承の場合も同じ
  - 子は親の一種、なので親だと言っても嘘ではない
  - 「MusicianはArtistの一種」、つまりMusicianをArtistとして扱って構わない
  - ArtistにMusicianを代入できる

配列やリスト

```
Artist[] alist = new Artist[10];
alist[0] = new Musician();
alist[1] = new Dancer();
...

for (Artist a : alist) {
    a.play();
}
```

その人にとっての正しいplay()が呼ばれる

- MusicianやDancerが何人もいる
- 全員にplayしてもらいたい

- Artistの集団だとして扱えばよい
  - Artistの配列に入れてしまえば全員同じように扱える

継承のここまでのまとめ

- クラスとはデータ（インスタンス）のまとめ方
- そのまとめ方にさらにまとまりをつけたい
  - ほとんど同じ分け方だけどちょっとだけ違うとか

- クラスの間に親子関係を作ってまとまりを表現する
  - 子の分け方は親の分け方の一種
  - 子クラスをざっくり言うと親クラスとも言える
- フィールドやメソッドを継承できる
  - 共通部分と個別部分が無駄なく記述できる
- メソッドをうまくオーバーライドすると切り替えができる

提出課題6: 円と三角形

- 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
  - どっちのクラスも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

課題

- ひな型ソースファイルは統合しようと作業し始めたところ
- FigureTest.main()のみ完成している
  - CircleとTriangleが1個ずつあり、それを同じように扱ってprintFig()したい、という使い方まではできている
- Circleクラスと Triangleクラスはもともとあった状態のまま
  - 適切に直してほしい
- Figureクラスも新たに作る必要がありそう

注意

- 何も付けないclassは、1つのjavaファイルの中に何個も置けます
  - パッケージ内からはアクセスできる、という状態のクラス
  - 今回のCircle, Triangle, Figureはこの状態で扱おう
    - FigureTest.java内に置くことにしよう
- public class は1つのjavaファイルには1個だけ
  - 今回はclass FigureTestだけがpublicにできる

提出物

- 提出物はFigureTest.java
  - 先頭に「**組番号、名前**」と、 FigureTest.main()で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 javalec6　とする

- 〆切は6/4(火) 17:00

ヒント

- なるべく共通部分は親クラスに取り出そう
  - 共通するデータはないか？
  - 可能なら、処理の中でも共通な部分はないか？

- (発展課題)コンストラクタについても調べてみると良い
  - 共通するフィールドの初期化をうまく書く方法がある
  - コンストラクタについてはまた次回以降で説明します（ので安心して）

本日のまとめ

- オブジェクト指向
  - カプセル化が大切な機能の1つ
  - アクセス修飾子: public, privateなど
- 継承
  - 考え方
    - 共通する機能やデータの共通化
  - 基本的な書き方
    - extends, 代入、オーバーライド
- 提出課題6