Java演習
第6回

2024/5/22

横山大作

講義前・後の質問

- 横山まで
- dyokoyama@meiji.ac.jp

提出課題4（再掲）

- Posクラスのオブジェクトのリストがある
- リスト内のすべての座標の要素をa倍するようなメソッドを作りたい
  - (3,6)という座標を3倍したら(9, 18)
- ただし、2通り
  - もとのリストはそのままで新たなリストを作って返す
    - dup_mult()
  - もとのリストの要素をa倍に書き換える
    - modi_mult()

解答例

- dup_mult(): 中身のインスタンスも新たに作る

```
public class MultTest {
    static ArrayList<Pos> dup_mult(ArrayList<Pos> l, int m) {
        ArrayList<Pos> ret = new ArrayList<Pos>();
        for (int i = 0; i < l.size(); i++) {
            Pos org = l.get(i);
            Pos p = new Pos(org.x * m, org.y * m);
            ret.add(p);
        }
        return ret;
    }
    ...
}
```

解答例

- dup_mult(): 中身のインスタンスも新たに作る

```
public class MultTest {
    static ArrayList<Pos> dup_mult(ArrayList<Pos> l, int m) {
        ArrayList<Pos> ret = new ArrayList<Pos>();
        for (int i = 0; i < l.size(); i++) {
            Pos org = l.get(i);
            Pos p = new Pos(org.x * m, org.y * m);
            ret.add(p);
        }
        return ret;
    }
    ...
}
```

配列そのもののnew

要素のnew

解答例

- modi_mult(): 元のインスタンスを書き換える

```
public class MultTest {
    ...
    static void modi_mult(ArrayList<Pos> l, int m) {
        for (int i = 0; i < l.size(); i++) {
            Pos org = l.get(i);
            org.x *= m;
            org.y *= m;
        }
    }
    ...
}
```

解答例

- modi_mult(): 元のインスタンスを書き換える

```
public class MultTest {
    ...
    static void modi_mult(ArrayList<Pos> l, int m) {
        for (int i = 0; i < l.size(); i++) {
            Pos org = l.get(i);
            org.x *= m;
            org.y *= m;
        }
    }
    ...
}
```

もらってきたorgは参照、
つまり元のインスタンス

書き換えれば元のインス
タンスが変わる

今回の内容

- Javaの言語に備わった機能の残り
  - Java API (標準ライブラリ)の使い方
  - 4,5,6章
  - オブジェクト指向に入る手前まで
- 参照型に関係する注意点
  - 等価性と同一性

- 提出課題5

配列（第4章）

- newの仕方を思い出しておきましょう
- 多次元配列(p.166)も使えます
  - 実体は配列の配列
  - newは一度に多次元まとめてできる

余談

- p.151「ループで配列の要素を順に使っていくことを「配列を回す」ともいうんだ」
- …本当？

- 「配列をなめる」「forループを回す」「for文を回す」ではなかろうか

メソッド（第5章）

- メソッド名の表記について
  - 引数があってもhello()みたいに書くことがある。引数なしという意味ではないことに注意
- メソッドをわかりやすい名前にするように心がけよう
- p.202「引数に配列を用いる」
  - 参照型でデータが渡される
  - オブジェクトが渡されるときも同じ、参照型
- P.204 「戻り値に配列を用いる」
  - 良くある書き方
    - オブジェクトを返すこともしばしば
  - 内部でnewしていることに注意しよう

オーバーロード (p.196)

- 違うメソッドを同じ名前にすることは原則できない
  - どれを呼んでよいかわからないから
- 引数の型や数が違うものは同じ名前にしてよい
  - 使うときに、自動的に選ぶことができるから
- 返り値の型では区別できないので注意

- p.199 「シグネチャ」
  - メソッド名+引数の型と数
  - メソッドを区別するための情報

第6章: 複数ファイルに分ける

- 1つのクラスにメソッドを増やしていくとわかりにくい
- そもそも1つのクラスは1つのデータの塊だった
  - プログラムはいくつものデータの種類でできている

- 複数のクラスを使うときは？
  - すでに提出課題で使っていますが

クラスとソースファイル

- 1つのソースファイルには1つのpublic classしか入れられない
  - public: 外から見える
- ソースファイル名はpublicなクラスと同じ名前

- 別クラスを作るときは別ファイル
  - eclipseとかつかっていれば自動的にそうなってます
- 同じディレクトリに入っていれば特に問題なく使える

別クラスのメソッドやフィールド

- p.215～216

- クラス名.メソッド名();
- クラス名.フィールド名
  - staticな場合

- 変数.メソッド名();
- 変数.フィールド名
  - staticでない場合（インスタンスにくっついている場合）

パッケージ(p.221)

- クラスが増えたらパッケージを使って分割管理できる
- Javaの機能
- クラスを包む（グループ化する）名前
  - 同じ名前のクラスでも、別パッケージなら共存できる
- ソースの先頭に　package XXXX;
  - もう課題で使ってきた
- パッケージを指定しないと「無名パッケージ」「デフォルトパッケージ」扱い
- パッケージに階層構造はない
  - ただし、名前の付け方でxxx.yyy.zzz みたいな付け方をする
  - java.lang.System　という感じ
  - ソースファイルやクラスファイルのディレクトリ構造と一致していることを要求するので注意
  - クラスパスの話(p.231～)はとりあえず説明しませんが、少し大きなものを作りたい人はぜひ読んでおいてください

パッケージ

- 別パッケージのクラスはフルに名前を書けば呼び出せる(p.224)
- 長い名前のパッケージを楽に使うためにimportができる(p.227)
  - import java.util.Date; // 1つのクラスをインポート
  - import java.util.*; // java.util以下のクラスを全部インポート
  - 現在のパッケージ内のクラスは何もしなくても利用できることにも注意
  - パッケージはディレクトリ構造と一致していることも思い出して。つまり、別のディレクトリにあるソースは何もしないと使えない。
- java.lang.*に関してはよく使うので、自動的にimportされている

Java API(p.243)

- Javaは多くのライブラリが標準的に備わっている
  - 文字列、時間
  - 入出力
  - GUI（ウィンドウシステム）
  - など
- 利用法、機能の調べ方について学ぶ

色々なパッケージ、クラス (p.246)

- java.lang.String
  - 実は使っていた(p.346)
  - java.langは自動的にインポートされるのも思い出して
- java.lang.System
  - System.out.println()はこれだった
- java.lang.Math
  - 平方根とかlogとか
  - java.mathは別のものが入っているので注意
- java.util.Date
- java.util.Calendar
  - DateやCalendarはjava.time.*に新しい良いものが作られていることに注意

APIリファレンス (p.247)

- Java api reference くらいで検索すれば出てくる
- バージョンによって微妙に異なるので自分が使っているJavaのバージョンを知ることが重要
  - 多分新しめのものならばそれほど違いはないでしょうけれど
  - 参考: Javaのバージョン名称は
  - 1.0
  - 1.1,…1.4
  - 5
  - 6,…
  - 最新は22とか(2024/3)
- 非常に大きいので、全部を見る必要はない
  - 必要になったらその都度探せばよい

APIリファレンスの例

- フィールドや作り方、メソッドなどが書いてある

参照型

- オブジェクトへのポインタのようなもの
- これを使っていると、「等しい」関係に関して、2種類の考え方があることに気づく

- 同じオブジェクトを指しているか？
  - **== で比較**
- オブジェクトの中身が同じか？
  - 同じ内容のコピーか？と考えても良い
  - **.equals()で比較**

- 違う概念なのだから、違う書き方をしないとダメ

等値と等価 (p.505)

- 等値：同じ存在を指していること (equality)
  - ポインターが等しい
  - ==

- 等価：同じ内容であること (equivalent)
  - 中身を見比べたら同じことが書いてあること
  - ある視点から見て、同じと取り扱ってよいこと
  - .equals()

String.equals() (p.112)

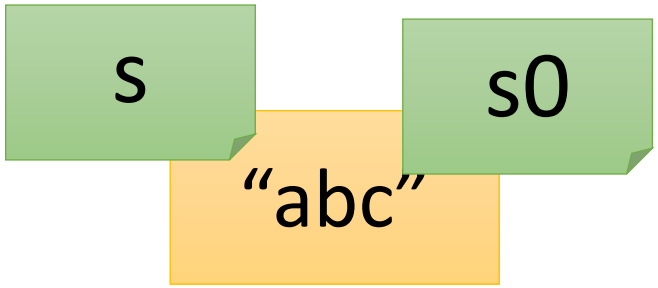
- 文字列の比較で「同じ文字の並びになっているか？」を比較するときはequals()を使う

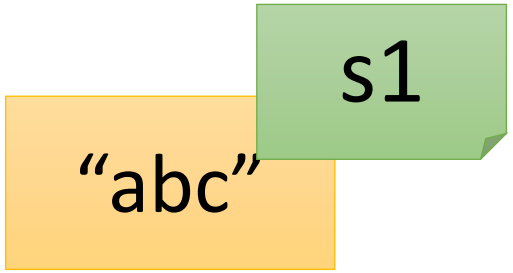
- Stringに限らず、オブジェクトが「中身を見ると等しい」かどうかのチェックにはequals()を使う

- 「等価」かどうかをチェックしたいときの方法
  - “==“ は「同一」かどうかのチェックになる

```
String s = “abc”;

if (s == s0) { ... }
if (s == s1) { ... }
if (s.equals(s1)) { ... }
```





「等価」の考え方

- ある基準で見て「等しいと扱ってよい」もの
  - インスタンスとしては異なるけれど、中身を見れば同じならば「等価」
  - 同一インスタンスかどうかを見るのは「等値」

- プログラマの都合を表現していると思ってもよい

同じ？違う？

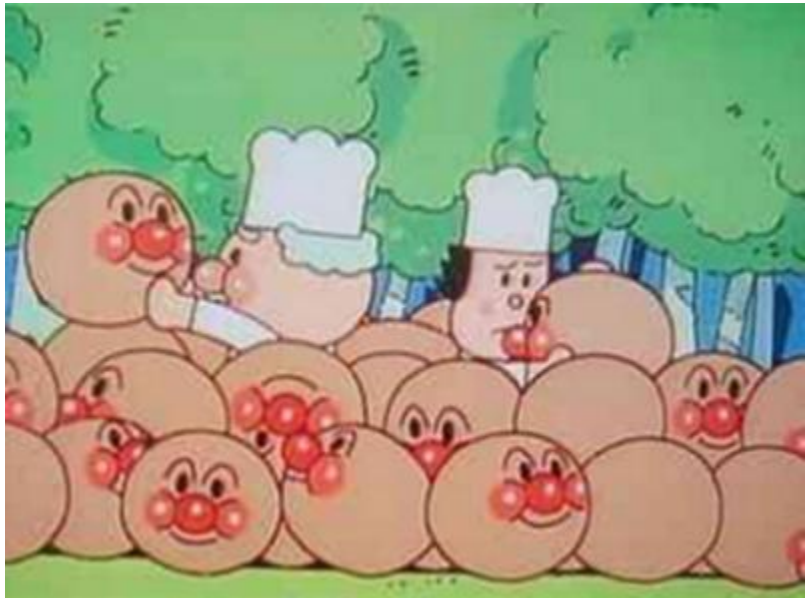
どちらも1万円として使えるんだから同じ！

現実に違うものなのだから違う！（ナンバーとか違うし）

- 結局、立場による
- 支払いがしたいだけなら「同じ」立場
- 銀行強盗に盗まれたお金を追跡したいのなら「違う」立場

哲学的

- 何をもって「同じ」とみなすかは意図しだい
- ==はインスタンスとしての同一性
- equalsはある「意図」「目的」を持っている



オブジェクト指向の難しさ

- プログラムはメモリ上にある「オブジェクト」が単位になって動くよ、という考え方
- 同じオブジェクトが複数できることがしばしばある
  - ネットワークやDBから持ってきたデータをオブジェクトにしたら、これはすでにメモリにあるオブジェクトと同じかも？
  - 分散システムだと勝手にコピーができているかも？
- 実体は2つに分かれたのに、この2つは「同じもの？」
- 何が「同じ」を決める基準なのか、自動的には判断できない
- 決めるのはプログラマしかいない

equalsの使い方

- 片方のオブジェクトに「君と同じ？」と判断してもらう
- どちらのオブジェクトに聞いても同じ（はず）
- まぁ見やすい方で良い
- 主従関係的に意図がある時はこだわろう

- どうしても対称に扱わないと気持ちが悪い人はObjects.equals()を使おう
  - どちらかがnullになる可能性がある時にも便利かも
  - import java.util.Objects;

```
String s1, s2;

if (s1.equals(s2)) { ... }
if (s2.equals(s1)) { ... }
if (s1.equals(“abc”)) { ... }
if (“abc”.equals(s1)) { ... } // ややバッドノウハウ的？

if (Objects.equals(s1, s2)) { ... }
```

String型の注意

- 一度作ると中身は変えられない
  - charの配列とは異なる
- 連結は「新しいStringを作ってコピー」している
  - 書き換えするように見えるメソッド(replaceなど)も同じ
- なぜ？そうするとうれしいことも多いから
  - バグ除けとかの面で

書き換えてなくて動くの？

- 次のコードではメモリ上にどんなインスタンスができている？説明してみよう。

```
String s = “はじめ”;
s = s + “なか”;
System.out.println(s + “おわり”);
```

- String自体を書き換えなくても、その参照を使っている限りは特に困らない

Immutable

- （一度できたら）書き換えができないデータ

- Immutableだと保証できると、色々良いことが起きる
  - いくらコピーしても良い（値は変わらない）
  - いつ読んでも同じ値：さっき読んだ値を使いまわしてよい

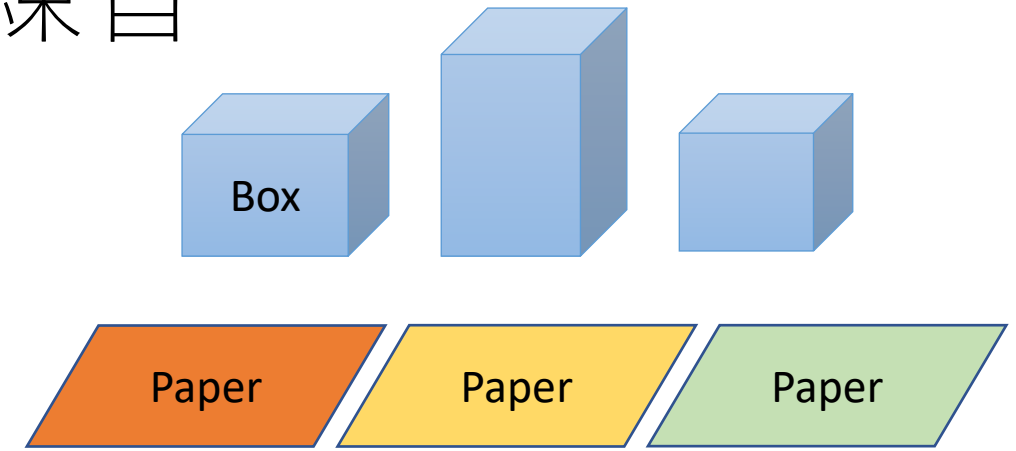
- 関数型は「世界のほぼすべてをImmutableにする」という感じ
- オブジェクト指向は「mutableなものはオブジェクトとして扱って、書き換えをオブジェクトに責任持たせたい」という感じ
  - Immutableなオブジェクトならそれはそれでうれしい
  - String、java.time.LocalDateTimeなど

Immutableと等価性

- Immutableなら、==とequalsは多少考え方が簡単になる
  - equalかどうかは未来永劫変わらない
    - ==かどうかはコンピュータの都合で変えたくなる
    - キャッシュメモリを考えてみよう。コンピュータの都合で勝手にデータのコピーを近くに持ってきている
  - equal使って判断しておけばよい部分が増える
    - 将来not equalになるかもしれないから、==かどうかをチェックしておこう、みたいなことをしなくても良い

意図を考える練習

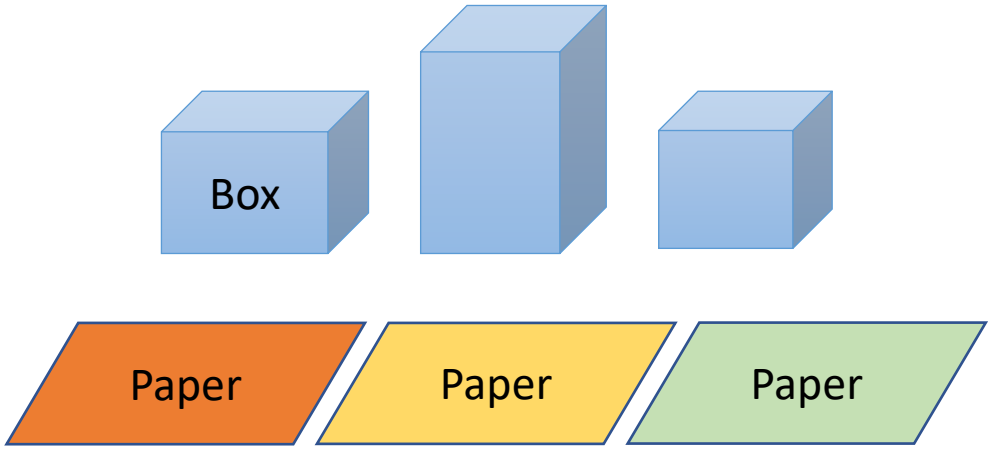
- 荷物: Box
- 荷物の包み紙: Paper
  - colorあり
  - 都合でcolorは日々変わるかも



- Boxはpaperというフィールドに包み紙を覚えている

```
class Box {
    Paper paper;
}

class Paper {
    int color;
}
```



- 2つのBoxについて、次のチェックをしたいときどう書く？

- 「2つのBoxが、今日は（たまたま）同じ色の包み紙を使っているか？」

- 「2つのBoxが、常に同じ色の包み紙を使う想定か？」

```
boolean check(Box a, Box b);
```

```
boolean check(Box a, Box b);
```

```
class Box {
    Paper paper;
}

class Paper {
    int color;
}
```

- 2つのBoxの包み紙
  - a.paper
  - b.paper
- この2つをどう比較するか？
  - a.paper == b.paper
  - a.paper.equals(b.paper)
    - Paperのequalsは定義されているとして
      - colorが同じかどうかで判定

```
boolean check(Box a, Box b);
```

```
class Box {
    Paper paper;
}

class Paper {
    int color;
}
```

- a.paper == b.paper
  - aとbの包み紙が「オブジェクトとして同じ」かどうかを比較
  - いつも「同じ紙」を意図しているはず
  - →「2つのBoxが、常に同じ色の包み紙を使う想定か？」

- a.paper.equals(b.paper)
  - aとbの包み紙の「データが同じ」かどうかを比較
    - 「色が同じか」を比較
  - →「2つのBoxが、今日は（たまたま）同じ色の包み紙を使っているか？」

equals()は自由に作れる

- Paperクラスのequals()を自分で定義できる
  - 後ほど、コレクションなどの時に説明します
  - （実践編　p.28）
- 基準を自由に決めても良い
  - 一部のフィールドだけ同じなら同じと扱うとか
  - ただし、よく考えてやらないと混乱したりする
    - 明日の自分は今日の自分とは別人

他のクラスでもequals

- Integer クラス
- Double クラス
- Booleanクラス

- ラッパークラスと呼ばれる(p.578)
  - 元の値をwrapしている薄い皮、のイメージ

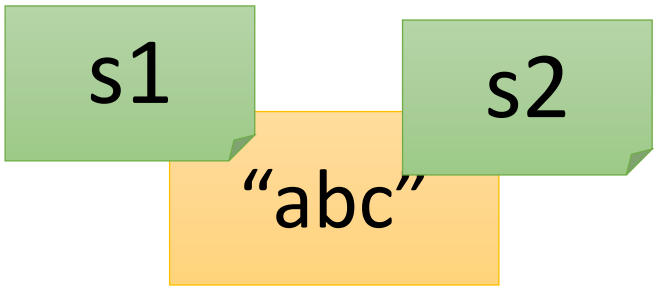
- 基本データ型をオブジェクトとして扱う、文字列から基本データ型に変換する、などの時に利用
- ラッパークラスもオブジェクトなので等価性と同一性の違いに注意
  - ==を使うと同一性チェックになってしまう

再利用されるインスタンス

- Stringの「リテラル」(値がコードに直接書いてあるもの)は、等価なStringはすべて同じインスタンスを指している
  - コンパイル時にまとめている
  - Stringが書き込み不可だからできること
- **一見、==でうまくいっているように見えてしまうので注意**

```
String s1 = “abc”;
String s2 = “abc”;

if (s1 == s2) { ... }
```



試してみよう

- コンパイル時に簡単な計算はしてしまう
  - 私の環境では、一番上の書き方ではs1 == s2になった

```
String s1 = “abc”;
String s2 = “a” + “bc”;

if (s1 == s2) { ... }
```

- （余談）腕に覚えがある人は、自由にコピーしたり、逆に共通化する方法を調べてみよう
  - String.intern()というメソッド
  - メモリをケチるときに使うかも

```
String s1 = “abc”;
String s2 = “a”;
s2 += “bc”;

if (s1 == s2) { ... }
```

```
String s1 = “abc”;
String s2 = new String(s);

if (s1 == s2) { ... }
```

再利用されるインスタンス

- Integerなども再利用されることがある
  - newで作らず、valueOf()で生成した場合
  - -128～127までの範囲の値は再利用
- Booleanは2つのインスタンスが存在
  - Boolean.TRUE ， Boolean.FALSE
- 再利用されていると==とequals()が同じ動きをするので注意

```
Integer i1 = Integer.valueOf(127);
Integer i2 = Integer.valueOf(127);

if (i1 == i2) { ... }
```

127が128になった瞬間に==の結果が変わる

提出課題5

- 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(Parts.NUM * Parts.NUM個)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

例

- さっきの文章で理解できるよう、少し考えてみよう

例

- Parts.item() が出してくるものが仮に以下だったとすると
  - {“ab”, “cde”, “abcd”, “e”}
- 組み合わせを全部作ると
  - {“abab”, “abcde”, “ababcd”, “abe”, “cdeab”, “cdecde”, …}
- 2回以上出るのは
  - {“abcde”}

- ひな型のParts.item()が返す文字列は違うことに注意！ひな型の方で正しく動くことを確認しよう

提出物

- 提出物はStringTest.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 javalec5　とする
  - StringTest.main()を呼び出したら課題の結果が表示されるようにする
  - 配布したParts.javaは提出する必要なし

- ✂切は5/28(火) 17:00

ヒント

- 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

簡単だと思った人

- コレクションクラスを使ってみよう
  - やりたいことにぴったりする袋はあるかな？
- 解ければどんな方法でも良しとします

今回のまとめ

- Javaの言語に備わった機能の残り
  - 配列
  - パッケージ
  - Java API (標準ライブラリ)の使い方
- 参照型に関係する注意点
  - 等価性と同一性
    - ==　は同じインスタンスかどうか
    - .equals()　は中身のデータが同じになっているかどうか