

Java演習

第11回

2024/6/26

横山大作

講義前・後の質問

- 横山まで
- dyokoyama@meiji.ac.jp

提出課題9: 配列のソート（再掲）

- 数字を表した文字列の配列がある
 - “19”, “3”, “0000470” みたいに
- これを「辞書順」と「整数だと思った時の数字の小さい順」の2通りでソートして表示しよう
- ソートアルゴリズムを「使う」練習です

まず、文字列としてソート

- **String.compareTo()**を明示的に呼び出しても良いし、デフォルトのソートがこれを使うのでそのままソートしても良い

```
public class ArrayTest {  
    public static void main(String[] args) {  
        String[] dat = ArrayDat.numstrings;  
        Arrays.sort(dat);  
        for (int i = 0; i < dat.length; i++) {  
            System.out.println(dat[i]);  
        }  
    }  
}
```

数字としてソート

- 「Stringの値を変換したintの配列を作ってそっちをソートする」という方法？
 - 間違いと言いたい
 - すごくデータサイズが大きい要素でも変換しちゃうの？
 - 出力するとintの数字が表示される
 - ので、0042とかの文字列は出てこない（42とだけ表示される）
 - ソートしたいのは「文字列」であって、数字ではない
- 複雑なデータはそのまま書き換えず、比較器を変える方法を身に着けよう
 - これからの人生でめちゃくちゃ頻出します

java.util.Comparator

- インタフェース
 - 「比較器」

```
int compare(T o1, T o2)
```

順序付けのために2つの引数を比較します。最初の引数が2番目の引数より小さい場合は負の整数、両方が等しい場合は0、最初の引数が2番目の引数より大きい場合は正の整数を返します。

- o1とo2がそのままの順序で良ければ負、入れ替えたければ正、と考えても（昇順ソートの場合）

実装例

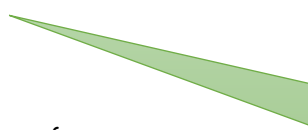
```
public class ArrayTest {  
    public static void main(String[] args) {  
  
        // . . . 文字列としてソートするのに続いて . . .  
  
        class CmpAsInt implements Comparator {  
            public int compare(Object lo, Object ro) {  
                int l = Integer.parseInt((String)lo);  
                int r = Integer.parseInt((String)ro);  
                return l - r;  
            }  
        };  
  
        Arrays.sort(dat, new CmpAsInt());  
  
        for (String s : dat) {  
            System.out.println(s);  
        }  
    }  
}
```

メソッド内のみで使えるローカルクラスもちろん外に出して普通のクラスにしても良い

拡張for文も使える

実装例（無名クラス）

```
public class ArrayTest {  
    public static void main(String[] args) {  
  
        // . . . 文字列としてソートするのに続いて . . .  
  
        Arrays.sort(dat, new Comparator() {  
            public int compare(Object lo, Object ro) {  
                int l = Integer.parseInt((String)lo);  
                int r = Integer.parseInt((String)ro);  
                return l - r;  
            }  
        });  
        for (String s : dat) {  
            System.out.println(s);  
        }  
    }  
}
```



どこまでがクラスの中身
なのかよく味わおう

ジェネリクス使うなら（後日説明します）

```
public class ArrayTest {  
    public static void main(String[] args) {  
  
        // . . . 文字列としてソートするのに続いて . . .  
  
        Arrays.sort(dat, new Comparator<String>() {  
            public int compare(String lo, String ro) {  
                int l = Integer.parseInt(lo);  
                int r = Integer.parseInt(ro);  
                return l - r;  
            }  
        });  
        for (String s : dat) {  
            System.out.println(s);  
        }  
    }  
}
```

キャストがい
らなくなる

今日のテーマ

- オブジェクトが備えることが多い機能
- インタフェースに関する理解を深める
- メモリ上のふるまいに関する理解を深める
 - オブジェクトの複製

インスタンスの5大基本操作 (実践編p.24)

インスタンスに対してしばしば要求される機能

- `toString()`: 表示
- `equals()`: 等価判定
- `hashCode()`: ハッシュ値、要約
 - 素早く比較するために用いられる
- `compareTo()`: 大小(順序)関係の定義
- `clone()`: 複製
 - 参照をコピーするのではなく、インスタンスをまるっと作りたいときに必要
- それぞれ「こういうメソッドを呼びましょう」と決めてある
- データを表すクラスなら、真ん中3つは必要そう

インスタンスの5大基本操作 (実践編p.24)

インスタンスに対してしばしば要求される機能

- toString(): 表示
 - equals(): 等価判定
 - hashCode(): ハッシュ値、要約
 - compareTo(): 大小(順序)関係の定義
 - clone(): 複製
-
- 上3つはjava.lang.Objectが持つ
 - デフォルトで使える、必要に応じてオーバーライド
 - 下2つは必要に応じてインタフェースを実装

インスタンスの5大基本操作 (実践編p.24)

インスタンスに対してしばしば要求される機能

- `toString()`: 表示
- `equals()`: 等価判定
- `hashCode()`: ハッシュ値、要約
 - 素早く比較するために用いられる
- `compareTo()`: 大小(順序)関係の定義
- `clone()`: 複製
 - 参照をコピーするのではなく、インスタンスをまるっと作りたいときに必要
- この3つはそれぞれ関連している
 - 一貫性を持っていないとおかしなことが起きる
 - `equal`なのに`hash`値が違うとか、`compare`したら`==0`でないとかは混乱のもと

例: String

- equals(), hashCode(), compareTo()は定義されている

```
String s = new String("abc");  
System.out.println(s.equals("xyz"));  
System.out.println(s.hashCode());  
System.out.println(s.compareTo("xyz"));
```

```
false  
96354  
-23
```

例: java.time.LocalDateTime

- equals(), hashCode(), compareTo()は定義されている

```
LocalDateTime ld = LocalDateTime.now();  
LocalDateTime ld2 = LocalDateTime.of(2023, Month.MAY, 25, 10,  
30, 45);  
System.out.println(ld.equals(ld2));  
System.out.println(ld.hashCode());  
System.out.println(ld.compareTo(ld2));
```

```
false  
2105515596  
1
```

自作クラスでも必要？

- 便利に使いたければ必要
- 文字列として表示することが多ければ`toString()`を定義
 - デバッグの時にも有効なので、作っておこう
- コレクション（便利な袋）で便利に使うなら`equals()`や`hashCode()`を定義
- ソートする場面があったら`compareTo()`
 - データにもともと順序関係が考えられる場合

Arrays.sortの例

```
String[] ss = {"xyz", "Abc", "abc"};  
Arrays.sort(ss);  
for (String s : ss) {  
    System.out.println(s);  
}
```

Abc
abc
xyz

- 文字列には自然な順序がある
 - 辞書順
- **Arrays.sort()**は「自然な順序があるならそれを使う」と決めている
 - どうやって「ある」ことがわかる？

java.lang.Comparable

- インタフェース
- 「比較できる」「順序がある」という性質
- これをimplementsしているクラスは比較できる
 - int compareTo()がある

java.lang

インタフェース Comparable<T>

型パラメータ:

T - このオブジェクトが比較されるオブジェクトの型

```
int compareTo(T o)
```

このオブジェクトと指定されたオブジェクトの順序を比較します。
このオブジェクトが指定されたオブジェクトより小さい場合は負の整数、等しい場合はゼロ、大きい場合は正の整数を返します。

String

概要 パッケージ **クラス** 使用 階層ツリー 非推奨 索引 ヘルプ

前のクラス 次のクラス フレーム フレームなし すべてのクラス

サマリー: ネスト | フィールド | コンストラクタ | メソッド 詳細: フィールド | コンストラクタ | メソッド

compact1、compact2、compact3

java.lang

クラスString

java.lang.Object クラスString

 java.lang.String

すべての実装されたインタフェース:

Serializable, CharSequence, Comparable<String>

- Comparableを実装している

自作クラスをソート

```
class A {  
    int x;  
    A(int x) { this.x = x; }  
    public int compareTo(Object o) {  
        return x - ((A)o).x;  
    }  
}
```

比較用の
compareTo()作った

```
void func() {  
    A[] a = {new A(5), new A(3), new A(2) };  
    Arrays.sort(a);  
    System.out.println(a);  
}
```

compareTo()でソート
してくれるかな？

Exception in thread "main" java.lang.ClassCastException:
class A cannot be cast to class java.lang.Comparable ...

実行すると例外

インタフェースの役割

```
Exception in thread "main" java.lang.ClassCastException:  
class A cannot be cast to class java.lang.Comparable ...
```

- AというクラスをComparableだと扱おうとした
 - けどComparableじゃないので例外発生
- 「比較できる」かどうかは、Comparableインタフェースを実装しているかどうかで判断
 - implements Comparable
 - compareTo()があるかどうか、ではない
 - (参考) pythonなど、「compareTo()があればOK」という考え方の言語もある

こうすればOK

```
class A implements Comparable {  
    int x;  
    A(int x) { this.x = x; }  
    public int compareTo(Object o) {  
        return x - ((A)o).x;  
    }  
}
```

- インタフェースをimplementsすると、
compareTo()を実装しないとコンパイルエラー
 - バグ除けになる
 - 引数を間違えてるとかのエラー除けになる
- Sort側も安心して使える

ソートを変えたい？

- 逆順にしたかったらどうする？

```
class A implements Comparable {  
    int x;  
    A(int x) { this.x = x; }  
    public int compareTo(Object o) {  
        return -(x - ((A)o).x);  
    }  
}
```

- `compareTo()`の符号を反転させれば確かに逆順にはなるけど...
- あるソートのためだけにデータ定義を変更したい？

Stringの例

- compareTo()
- compareToIgnoreCase() : 大文字小文字を無視して比較

```
String s = new String("abc");  
  
System.out.println(s.compareTo("xyz"));  
System.out.println(s.compareToIgnoreCase("Abc"));
```

-23
0

自然な比較方法がいくつも考えられる

Sortに「比較器」を渡す

- `sort`の引数に「比較のために使う比較器」のインスタンスを渡す

```
String[] a = new String[] {  
    "aaa", "bbb", "aab", "a", "aax", "x"  
};  
  
class Cmp implements Comparator {  
    public int compare(Object l, Object r) {  
        return ((String)l).length() - ((String)r).length();  
    }  
};  
  
Cmp cmp = new Cmp();  
Arrays.sort(a, cmp);
```

さっきの例なら

```
class Cmp implements Comparator {  
    public int compare(Object l, Object r) {  
        return ((A)l).compareTo(r);  
    }  
};  
Cmp cmp = new Cmp();  
Arrays.sort(a, cmp);
```

- **A.compareTo()**使いたいなら、**compareTo()**を使う比較器を作って**sort**に渡せばよい

(参考) 逆順

- **Comparator**のインタフェースには**reversed()**メソッドがある
 - インスタンスから、「逆順にしたインスタンス」を作る
 - デフォルト実装が入っていて、インタフェースだけで自動的に使える

```
class Cmp implements Comparator {  
    public int compare(Object l, Object r) {  
        return ((A)l).compareTo(r);  
    }  
};  
Cmp cmp = new Cmp();  
Arrays.sort(a, cmp.reversed());
```

比較のまとめ

- 自然な順序関係があるなら、Comparableにする
 - compareTo()を実装する
- 比較したいときだけなら、比較器を作って使う
 - Comparatorインタフェース
 - 必ずComparableにする必要があるわけではない

オブジェクトの複製

- オブジェクトを複製したいとき
 - =では参照しかコピーされないことを思い出そう

```
A a = new A(5);  
A b = a;  
b.X = 3;
```

- これは複製できていない (a.xも書き換わってしまう)
 - メモリ上でどうなっているか、イメージを書けるように！

外の人が複製しようとする

- オブジェクトを複製したいとき：愚直には
 - newして
 - フィールドをコピー

```
A a = new A(5);  
A b = new A();  
b.X = a.x;
```

- 面倒
- **A**の中がどうなっているかを外の人が知っている必要がある
 - 知らないうちにフィールドが増えていたらどうする？

複製の方法

- 複製するメソッドを用意してもらう
 - 中身がどうなっているかはオブジェクトが知っているはず
 - オブジェクトが責任をもって自分のフィールドを全部複製する
- 方法は 2 通り
 - コピーコンストラクタ
 - Cloneable / clone()
- Cloneableは色々変で、避けた方が良さそうだ...
 - 実践編の教科書はCloneableを説明してるけど

コピーコンストラクタ

- コンストラクタの引数が自分と同じ型のオブジェクト
 - 仲間のオブジェクトを見せてもらって、新たに自分を作る
- 普通はこれが良い方法のようだ

```
class A {  
    int x;  
    A(A org) {  
        x = org.x;  
    }  
}  
  
class B extends A {  
    int y;  
    B(B org) {  
        super(org);  
        y = org.y;  
    }  
}  
  
...  
    A m = new A();  
    A n = new A(m);
```

clone()の使用

2つやらないとだめ

```
class A implements Cloneable {  
    int x;  
    public A clone() {  
        A ret = new A();  
        ret.x = x;  
        return ret;  
    }  
}
```

- java.lang.Cloneable インタフェースを実装する
- clone() を public でオーバーライドする
 - clone() の中身を適切に作る
 - public にする
 - 実は java.lang.Object.clone() は protected で宣言されている
 - 外の人からは呼び出せない
 - 使ってもらうために public で定義しなおす
 - 復習: アクセス修飾子は「緩くする」ことはOK、「使えなくする」ことはNG

clone()の実装

- 親クラスがCloneableの場合
- 親のコピーは親のclone()に任せる
 - newでなく、super.clone()を呼ぶ

```
class B extends A implements Cloneable {  
    int y;  
    public B clone() {  
        B ret = (B) super.clone();  
        ret.y = y;  
        return ret;  
    }  
}
```

豆知識

- **Cloneable**は実は関数を定義していない
 - `clone()`は`java.lang.Object`が既に持っているので、インタフェースで定義しなくても存在する
 - 単に「しるし」としてインタフェースが使われている
 - マーカーインタフェース と呼ぶ
- 実は`java.lang.Object`は`clone()`を呼び出せない
 - **Cloneable**を実装していないため
 - 実行時に例外がスローされる

どこまで複製する？

- 変数にオブジェクトを覚えさせるとき、「参照」が入っていたことを思い出そう
- m のコピーである n は、どのようなオブジェクトになっていることが期待されるだろうか？
 - 絵に描けるだろうか？

```
class A {  
    int x;  
}
```

```
class C {  
    A a;  
}
```

...

```
C m = new C();  
C n = //何らかの方法でmをコピー
```

1つの例

- 例えばこうコピーコンストラクタが書かれているとき、mとnはどんな状態？
 - 絵に描けるだろうか？
- `m.a.x = 100;` と代入すると、nはどうなるだろうか？

```
class A {  
    int x;  
    A() { x = 0; }  
}  
  
class C {  
    A a;  
    C() { a = new A(); }  
    C(C org) {  
        a = org.a;  
    }  
}  
  
...  
C m = new C();  
C n = new C(m);
```

Shallow Copy（浅いコピー）

- オブジェクトのフィールドの値をそのままコピーするものをShallow Copyと呼ぶ。
 - 基本データ型はそのまま
 - 参照型も「その参照のまま」コピー
 - 同じ参照先を指している状態になる

```
class A {  
    int x;  
    A() { x = 0; }  
}  
  
class C {  
    A a;  
    C() { a = new A(); }  
    C(C org) {  
        a = org.a;  
    }  
}  
  
...  
C m = new C();  
C n = new C(m);
```

Deep Copy（深いコピー）

- フィールドが参照だった時、その参照の先のオブジェクトを複製する（別のオブジェクトに作り直す）ものを**Deep Copy**と呼ぶ。
 - 参照をたどって行って、オブジェクトを全部複製するもの
 - Cのコピーコンストラクタの中身が変わったことに注意しよう

```
class A {  
    int x;  
    A() { x = 0; }  
    A(A org) { x = org.x; }  
}  
  
class C {  
    A a;  
    C() { a = new A(); }  
    C(C org) {  
        a = new A(org.a);  
    }  
}  
  
...  
C m = new C();  
C n = new C(m);
```


Shallow vs Deep Copy

- どちらの複製の立場もあり得る
- どちらの動作をさせたいのか、よく考える
- 配列を使うと問題にぶつかりがち
 - 配列の中身は参照型なので
- 言語で用意されているコピーはたいていShallowであることが多い
 - Deepはよく理解せずに使うとすごくたくさんのオブジェクトをコピーするかも
 - 「本当にDeep Copyは必要か？」をよく考える

ちなみに文字列だと

- class Aの代わりにStringだった場合
 - これも参照型
- m.aを書き換えるとn.aはどうなる？
 - m.a = "xyz";

```
class C {  
    String a;  
    C(C org) {  
        a = org.a;  
    }  
}  
  
...  
C m = new C();  
C n = new C(m);
```

ちなみに文字列だと

- もし文字列が書き換えられるならば、例えば
`m.a.change("xyz")`というメソッドが実際に文字列を書き換えるならば、影響はnに及ぶ
- しかし、そういう書き換えをするメソッドはない。
 - 書き換えるならば
`m.a = m.a.change("xyz")`
みたいな書き方しかできない
 - `m.a`が新たなオブジェクトを指し直すだけ

```
class C {  
    String a;  
    C(C org) {  
        a = org.a;  
    }  
}  
  
...  
C m = new C();  
C n = new C(m);
```

共有の影響

- 複製を作ったとき、意図せずに別のオブジェクトに書き換えの影響が出てしまうのは、あるオブジェクトの「中身」を書き換えられるとき
 - オブジェクトへの参照が同じままで、中身が書き換わるとき
- **String**で影響が出なかったのは、**String**は中身を書き換えられないから
 - **Immutable**と呼ぶ
- **Immutable**なオブジェクトを使っているなら、共有しても影響が起きにくい

ガベージコレクション (p.168)

- データ（のメモリ領域）を自動的に回収してくれる仕組み
- モダンな言語では提供されていることが多い
- 変数(の先のオブジェクト)が使われなくなったらそのオブジェクトを消す
- いつ動くかは決まっていない
 - メモリが足りなくなったら動く、ということがほとんど
 - ひょっとしたら最後まで動かないことも

GCを助ける？

- 使わなくなった参照にnullを代入すると明示的に参照を消せる
 - そのうちオブジェクトが破棄されると期待できる
- 多用しなくてよい
 - GCの良さが台無し
- メモリ管理を自分で書いているときはnull代入をせざるを得ない
 - オブジェクトの配列を自分で管理してるとか
- 普通はnull代入するような事態にはならない

提出課題10: 複製

- 箱詰めのチョコレートを表現しようとしたクラスが作られている
 - `class Chocolate`: 1粒のチョコレート、味が`int`で表現されている
 - `class Box`: チョコが何粒か入った箱。`Chocolate`インスタンスがいくつか配列で覚えてある
- ソースコードのひな型を参照

CloneTest.main()でやること

- 3個詰めセットのサンプルとなるsampleインスタンスが用意されていた
 - 味は[0, 1, 2]だった
- これを使って、1個だけ味を変更したboxインスタンスを作りたい
 - sampleをコピーしてboxを作り、boxの0番目の味を9に変える(set_flavor)
 - 味が[9, 1, 2]になるはず
- このような処理が問題なく（sampleに影響を与えないこと）行えるようにプログラムを完成してほしい

提出物

- 提出物はCloneTest.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 javalec10 とする
 - CloneTest.main()を呼び出したら、boxとsampleの中身を表示するので、期待された結果が表示されるようにする
- ✂切は7/2(火) 17:00

ヒント

- 複製の作り方はコピーコンストラクタでも `clone()` でもどちらでも良い
- ひな型の **XXX** の部分と、他にも書き直すべき場所を考えて直す
- 結果が正しくなるようによく確認
 - `box` は味が変わるように
 - `sample` は味が変わらないように