Java演習
第13回

2024/7/10

横山大作

講義前・後の質問

- 横山まで
- dyokoyama@meiji.ac.jp

提出課題11: 点の重複排除（再掲）

- 座標を表すPointクラスがある
- Pointクラスには、テスト用の点を出力するPoint.testPoint(int i)というメソッドがある
- testPoint(1)からtestPoint(10)までで得られる10点について、相異なる点はいくつあるだろうか？重複を考慮し、2回以上現れる点は1回のみ表示するようにして、全部の点を表示しよう。
  - 「点が等しい」とは、座標の値が等しいこと
  - Point.toString()が用意されているので、異なる点を1行に1つずつprintln()する

まずやるべきこと

- 10個の点が与えられる＝10個のインスタンスが与えられる
- 異なるインスタンスでも「中身が同じ」ものは同一と判断したい

- equals()を定義すればよい

```
class Point {
    // . . .

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Point other = (Point) obj;
        return (x == other.x && y == other.y);
    }
}
```

- Eclipse使うとひな形が作られるので、それを使うとこんな感じ
  - 最後のところはこの例は手で書いたけれど、Eclipseが自動で作ってくれるはず

重複排除の方針

- コレクションにあるSetを使えば一発
  - 勝手に重複排除してくれる

- 実装がHashSet -> hashCode()が必要
- 実装がTreeSet -> 要素を比較する方法が必要
  - 比較する方法は2通り
    - 要素にcompareTo()を実装（Comparableインタフェースを実装）
    - TreeSetのコンストラクタにComparatorのインスタンスを渡す

HashSetの場合の例

```
class Point {
    // . . .

    @Override
    public int hashCode() {
        return Objects.hash(x, y);
    }
}
```

- Pointの方にhashCode()が必要
  - コレクションの「袋」「中身」の違いに注意

HashSetでの重複排除

```
public class PointTest {
    public static void main(String[] args) {
        Set<Point> s = new HashSet<Point>();
        for (int i = 1; i <= 10; i++) {
            s.add(Point.testPoint(i));
        }
        System.out.println(s.size());
        for (Point p : s) {
            System.out.println(p);
        }
    }
}
```

- Setに入れれば勝手に重複排除
- あとはサイズ見て、要素を取り出して表示

List.containsを使う

- Listを用意する
- Pointを順番に取り出す
- もし、まだListに入っていないならばListに追加する
  - ここでcontainsを使ってチェックする
- 最後にList全体を表示する

- という方法でもOKです。大量の点になると効率は悪くなるけど（考えてみましょう）

ときどきあった間違い

- hashCodeが一致したら同じ点だと思う
  - equals()の中身が

    return hashCode() == other.hashCode();

    になってるとか。
- これは間違い。hashは「だいたいこんな値」を返すものなので、厳密に同じかどうかは中身をきちんと比較しないとダメ
  - Hashの意味をもう一度復習しておこう

この資料の内容

- コレクションの続き
  - Mapインタフェース
- Collectionsクラスの使い方
  - 関数型インタフェース
  - ラムダ式

- 提出課題12

袋を大きく分類（再掲）

- 順序を保つ必要がある
  - List
- 順序には(あまり)興味がない
  - 要素だけで良い
    - Set
  - 対応関係が必要
    - Map

Map (p. 601)

- Key, Value のペアを保持するもの
  - Keyは重複なし
- Object put(K key, V value)
  - もともとkeyが入っていたときはMapの中は上書き、返り値で元の値を返す
- Object get(K key)
  - ジェネリクスを使うとObjectではなくV型が返る

```
Map<String, String> m = new HashMap<String, String>();

m.put(“red”, “赤”);
m.put(“blue”, “青”);

String value = m.get(“blue”);
System.out.println(value);
```

Mapの中身をなめる

- keySet(): keyの一覧 (Setで返ってくる)
- values(): valueの一覧 (Collectionで返ってくる)
- entrySet(): keyとvalueのペアの一覧 (Entry<K, V>型を要素とするSetで返ってくる)

などがデータをなめるときには使える

```
Map<String, String> m = new HashMap<String, String>();
// . . .

for (String k : m.keyset()) {
    String value = m.get(k);
    System.out.println(k + “:” + value);
}
```

中身をなめる（続き）

- Iteratorを使った走査も可能 (p.586)
  - Iteratorは「Listの途中を指す」ようなもの
  - Listのどこまで見たか、を表すオブジェクト
- キーがあるので「このキーの要素」は取れる
  - 「3番目の要素」とかが取れないのはSetと同じ

```
Map<String, String> m = new HashMap<String, String>();
// . . .

Iterator<String> it = m.values().iterator();
while (it.hasNext()) {
    String value = it.next();
    System.out.println(value);
}
```

Mapの実装

- HashMap
  - hashCodeを使っている
- TreeMap
  - 木構造を使ったMap
- LinkedHashMap
  - HashMapに入れられた要素の順番を保持するリストの機能が付いたもの
  - リストとMapの混ざった感じ
- など

TreeMap

- キーを二分探索木で保持する
- ソートされたキーが扱える
- 「x以上y以下のキー」「x以上である最小のキー」などが取れる
- キーはComparableであるか、Comparatorを作成時に指定することが必要
- 要素の検索、追加、削除はHashよりは遅くなる
- Mapを拡張したSortedMapインタフェースの実装

- ちなみに、前回あまり説明しなかったTreeSetも同様

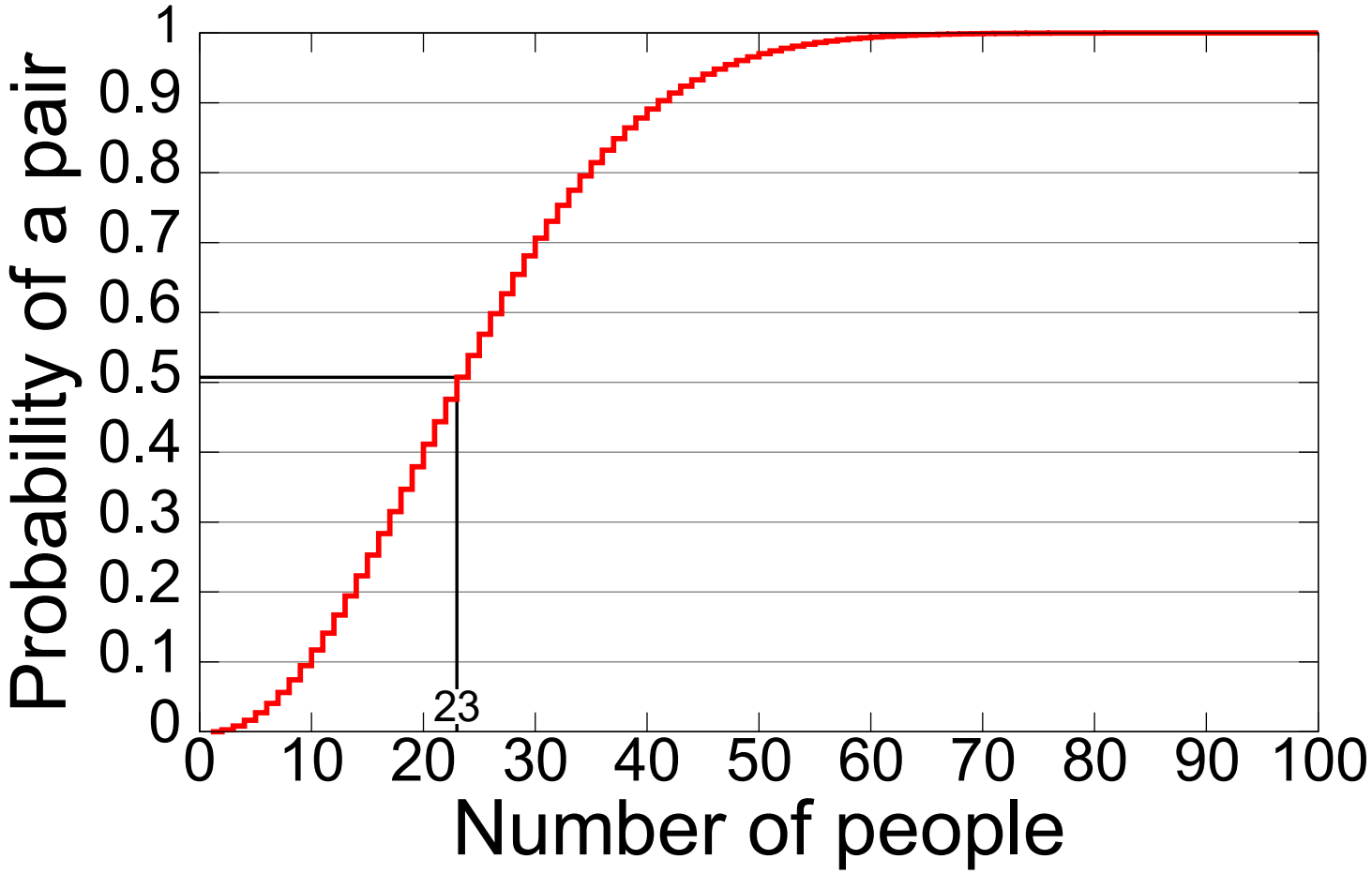
余談: ハッシュがぶつかる確率

- 誕生日(月日)をハッシュで保存するときを考える
- 100人のデータを入れるとどれくらいの確率でぶつかるか？
  - 100/365 ではないよ
  - 全部の人がぶつからない確率から考えてみよう

- 誕生日問題として有名

- 知っておくべきこと：ハッシュがぶつかったからと言ってMapの性能には大した影響はない

集団に同一誕生日の人がいる確率



Wikipedia: 誕生日問題　より

（余談）SetはMapで作れる

- MapのvalueのないものがSet
  - valueに適当な値を入れておけばよい

- 実際、そのような実装にしてあることが多い

Vector, Hashtable, Enumeration

- コレクションが出てくる前に使われていたコレクションっぽいデータ構造
- 古いので省略
- 過去のコードに出てきたらググろう

配列とListの変換

```
String[] array = new String[]{“東京”, “ロンドン”, “パリ”};

List<String> list = Arrays.asList(array);
List<String> list2 = Arrays.asList(“大阪”, “名古屋”, “札幌”);
// . . .

String[] array2 = list.toArray(new String[0]);
// . . .
```

- List.toArray()にはちょっと不思議な引数が付いているので注意
  - よければこの領域を使ってね、という指示
    - 大きさが十分ならそこを使う、足りなければ新たに配列が確保される
  - この例ではサイズ0の配列を渡している

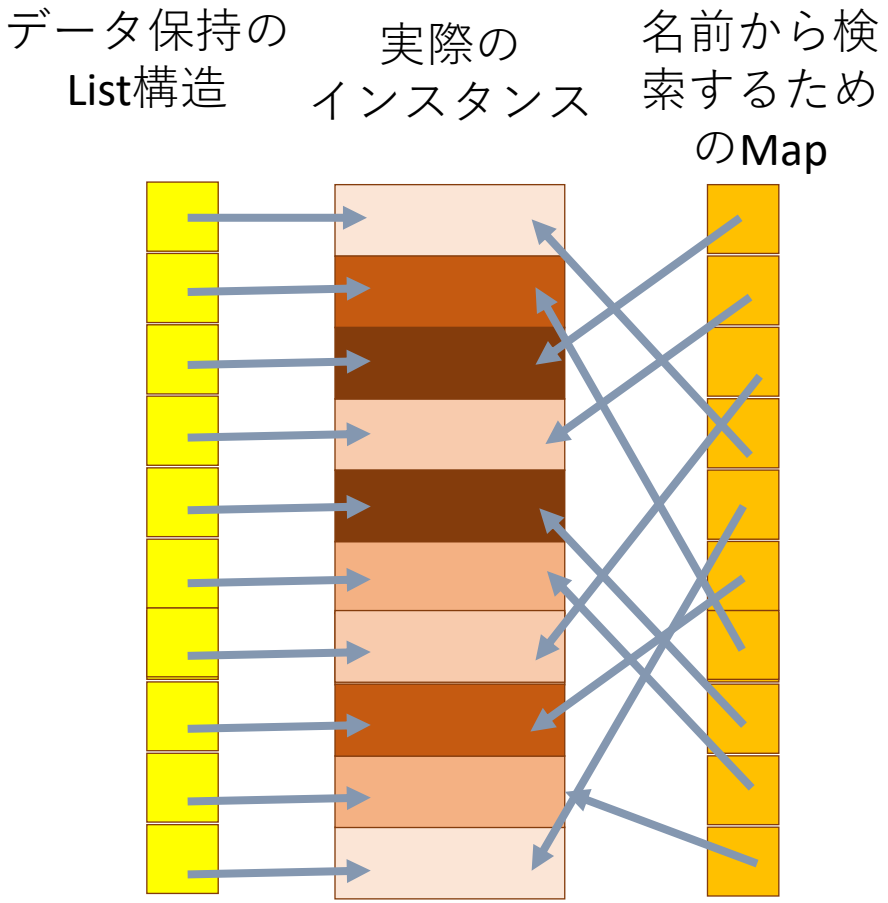
複数のコレクションに入れた要素

- 全学生のデータはListに入っている
- ある講義の受講者だけ、名前を使って頻繁に検索したい
- 必要なものだけ　名前->データ　という関係をMapで覚える

- この時、インスタンスはどのようにつながっているか絵で描けますか？
  - コレクションの要素には実際は「参照」が入っている
    - 中身がオブジェクトだから

インデックスを張る

- 検索用に外部に構造を作ることを「インデックスを張る」と呼ぶことがある
  - DBの用語
- よくある状況
- 複数のインデックスを張ることもよくある
- データを書き換えたときにインデックスを直さないといけないときがあることに注意
  - あるいは1要素を消したとき、どうなるか説明できますか？



コレクションの組み合わせ (p.606)

- コレクションは組み合わせて使える
- 例：1つのキーに複数の要素が対応するMap
  - 1人の学籍番号に複数のメールアドレスを覚えさせたい
  - Map<String, String> では1つの学籍番号に1つのメールアドレスのみ
  - Map<String, List<String> > で良いよね
  - 組み合わせて複雑なデータ構造が作れる
    - 参考: これはMultiMap と呼ばれる構造。よく使われるのでGoogleなどが作ったライブラリもある。
- Map.keySet() とかMap.values() はSetが返ってくる

Map<String, List<String>> の例

```
Map<String, List<String>> m
        = new HashMap<String, List<String>>();

m.put("abc", new ArrayList<String>());

List<String> abc_l = m.get("abc");
abc_l.add("val1");
abc_l.add("val2");
```

参照による格納(p.607)

- コレクションに格納されているのは参照
  - 複数の人がコレクションの中身への参照を持ちがち
  - 誤って要素を変更しがち
- 積極的に共有構造を使う場合もある
  - 前述のインデックスを張るときなど

- メモリ上のイメージ、データの構造をよく意識して使おう
- 参照はコストが低いこともよく理解しよう
  - 「情報工学のあらゆる問題は1つ間接参照を挟めば解決する」みたいな言われ方もある

Collections

- コレクションを扱うときの様々なヘルパー
  - Collection　ではないことに注意
  - List、Map等のCollectionに属するデータ構造に対応
- ソートとかある
  - 他に、binarysearch、copy、disjointなどいろいろ
- Arraysクラスと同じ立場
  - Arrays.sort()使ってみましたね
  - こちらは配列専門

ソートとComparator

- 順序があるコレクション(List系)はソートできる
- ソート順序はコントロールできる

- デフォルトではcompareTo()でソート
  - Comparableインタフェースを備えたデータなら比較できる
- デフォルトと違う順序にしたいときは比較基準を明示的に指定する
  - Comparator

関数オブジェクト（復習）

- 変数は「データ」「オブジェクト」を入れるもの、とこれまで扱ってきた
- 「関数」だって変数に入れられる
  - データとしての実体がないものだけど、想像してみよう
  - 今までは「名詞」を変数に入れてきた
    - 座標、学生、記事
  - 「動詞」だって入れられる
    - 動かす、比べる、要約する
  - 変数に入れられると部品として使える
    - 引数に渡せる、変数で取っておける
    - 入れ替えて使える、何度も使える

名詞と動詞：例

- 鉛筆

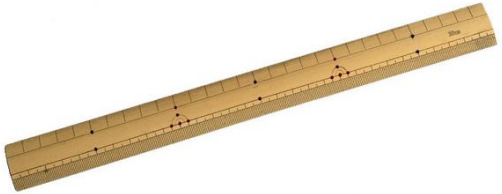
- 削る

動詞のままだとわかりにくいなら、「削るもの」と「もの」を付けるとわかりやすいかも

名詞と動詞：例2

- 鉛筆

- 比較する



鉛筆の並び順は「比較する装置」によって決まる

Comparator

- java.util.Comparator　というインタフェース
- 2つの要素を受け取って、大小関係を返す関数オブジェクト
- sortメソッドの第2引数に渡せばソート基準になる

- 普通のクラスとして定義してもOK
- 無名クラスにもできる(第10回講義資料)

自作クラスでソートの例

- 食料品店の商品管理を考える。
  - 商品には名前と価格がある
- 食べ物を値段が高い順にソートしてみよう
  - Foodにはint getPrice()があるとする

```
class Food {
    private String name;
    private int price;
    Food(String n, int p) {
        name = n; price = p;
    }
    int getPrice() { return price; }
}
```

方法1: Comparableにする

- Food自身が比較できるようにする

```
class Food implements Comparable<Food> {
    private String name;
    private int price;
    Food(String n, int p) {
        name = n; price = p;
    }
    int getPrice() { return price; }
    @Override
    public int compareTo(Food o) {
        return name.compareTo(o.name);
    }
}
```

Comparableという性質を持つと定義する

このメソッドが比較に使われる

この例は名前順に順序が付くと想定

Comparableのソート

- Collections.sort()に比較の基準を与えなければComparableのcompareToが使われる

```
List<Food> l = new ArrayList<>();

Collections.sort(l);
```

- もちろん、比較基準を明示的に指定しても良い（後述の方法で）

Comparableの与える順序

- データに1つだけ
- データにもともと備わる順序　　と思われる

- なるべくそのデータにとって自然なものであると想定される
  - 今回の例では、名前のみの順番
  - 価格を無視しているがそれでよいか？
  - equal()でないものはなるべく順序を付けたほうが良いでしょう

- 必要に応じて変える、というようなものではない

方法2: Comparatorを作る

- getPrice()の値を比較する

```
void sortPrice() {
    class Cmp implements Comparator<Food> {
        public int compare(Food l, Food r) {
           if (l.getPrice() > r.getPrice()) {
              return -1;
           } else if (l.getPrice() < r.getPrice()) {
              return 1;
           } else {
              return 0;
           }
        }
    }
    Cmp cmp = new Cmp();
    Collections.sort(foods, cmp);
    for (Food f : foods) {
        System.out.println(f);
    }
}
```

これが「getPrice()の値で比較する比較器」の関数オブジェクト

Comparatorを作る

- こんな書き方もよくある
  - 単純なintの昇順、降順のときによく見る

```
void sortPrice() {
    class Cmp implements Comparator<Food> {
        public int compare(Food l, Food r) {
            return -(l.getPrice() - r.getPrice());
        }
    }
    Collections.sort(foods, new Cmp());
    for (Food f : foods) {
        System.out.println(f);
    }
}
```

もっと複雑なオブジェクト

- 例えば「閾値未満の商品はサービスで無料とする」場合
  - オブジェクトなので、内部に色々情報持てる

```
class Cmp implements Comparator<Food> {
    int threshold;
    Cmp(int th) { threshold = th; }
    public int compare(Food l, Food r) {
        int li = l.getPrice(); int ri = r.getPrice();
        if (li < threshold) { li = 0; }
        if (ri < threshold) { ri = 0; }
        return -(li – ri);
    }
}
Cmp cmp = new Cmp(100);

Cmp cmp2 = new Cmp(200);
```

「100円未満は0円と扱う」という専用の比較器

「200円未満は～」という比較器

さらに別解

- 無名クラスにすることもできる
- ちょっと考えてみましょう

- sortの引数でnew Cmp() しているところにクラス宣言が入れば良いだけ

関数型インタフェース

- 抽象メソッドを1つだけ持つインタフェース
  - 「このメソッドを作ることが目的」とわかっているインタフェース
- 様々なアルゴリズムの「ここを埋めろ」という部品に使われる
  - 例：Comparatorインタフェースは関数型インタフェース

- 関数型インタフェースの中身を簡単に書けるように「ラムダ式」という仕組みがある

ラムダ式

- 関数を（見た目的に）扱いやすくしたもの

- 参考
  - 名前の由来は「ラムダ計算」
  - 計算をどのようにモデル化するか、という学問から生まれてきた考え方
  - 関数型言語などで重要な基盤

ラムダ式の書式

- (引数) -> { 処理 }

文字列の長さソートの例は
ラムダ式ではこれ

```
(String o1, String o2) -> {
  return o1.length() – o2.length();
}

(int x, String s) -> { return s.length() * x; }

() -> { System.out.println(“called”); }
```

引数がない例

省略形

- 引数の**型**は省略できる
  - インタフェースの型情報がある

```
(o1, o2) -> {
    return o1.length() - o2.length();
}
```

- 引数が1個しかないときは()が省略できる
- 処理が1行のみで書けるならばreturn と{}と;が省略できる

```
(o1, o2) -> o1.length() - o2.length()
```

ラムダ式でのComparatorの例

```
void sortPrice() {
    Collections.sort(foods,
                     (l, r) -> -(l.getPrice() - r.getPrice())
                     );
    for (Food f : foods) {
        System.out.println(f);
    }
}
```

Comparatorが
ラムダ式で書ける

- ずいぶんすっきり！
- 一時的に使いたい関数オブジェクトが必要な時には重宝する

ラムダ式の外の変数

- 外側の変数をラムダ式内で見ることができる
  - ただし、final扱いに変化する
  - つまり、その時点での「定数」として見える

```
int threshold = 5;
filterd = l.filter(x -> x > threshold)
```

参照のみ可能

- 理由はおそらく、ラムダ式は遠く離れた場所でも実行できるから
  - 遠くから書き換えられたら気づきにくい
- 外側の変数を書き換えるとコンパイルエラーになるので注意

メソッド参照

- 関数を渡すための別の方法
- メソッドそのものの名前を関数型インタフェースが使える場所に使える
  - インタフェースの引数の数、型が一致していれば

```
int lcmp(String o1, String o2) {
  return o1.length() - o2.length();
}
void func() {
  Collections.sort(list, this::lcmp);
}
```

メソッド名を関数オブジェクトのように使える

ソートをまとめると

ソート基準は

- デフォルト: ComparableインタフェースのcompareTo()
  - データごとに1つだけ決まっている
  - データにとって自然な順序をComparableで指定する
- 変えたいとき：
  - Comparatorのオブジェクトを渡す
    - 実クラスを作って渡す
    - 無名クラスで渡す
    - ラムダ式で渡す
    - メソッド参照で渡す
  - Comparatorにはint compare()を実装

- Comparable / Comparator 紛らわしいですがよく見て

参考: Comparatorクラス

- Comparatorを作る便利機能が色々ある
  - Comparator.comparing()
    - キーとなる値を取り出すメソッドを与えると、そのキーでの比較器を作ってくれる
    - Comparator.comparing(Food::getPrice); とか
  - 「キー取り出し器」をはめ込んで「キーによる比較器」を作り、ソート器にはめ込む

- import java.util.Comparator;

参考文献

- Joshua Bloch, Effective Java 第3版, 丸善出版
- 谷本心ら、Java本格入門、技術評論社

ジェネリクスなどが気になったら読んでみよう

提出課題12: 小田急

- 小田急小田原線の中で、指定された駅との距離が近い順に5つの駅について、駅名と乗降客数などを表示しよう

データが入っているクラス

- OdakyuDataクラス (OdakyuData.java)
- int numStations()
  - 駅の数を返す(Nとする)
- String[] station(int id)
  - [0, N)の範囲の駅番号を入れると、駅の情報の文字列が3個配列に入って出てくる
  - “駅名”, “乗降客数”, “新宿からの距離”
- 例: station(0)と呼ぶと{“新宿”,“355563”,“0.0”}が返ってくる

- OdakyuData.javaは今回の提出ファイルとは別ファイル
  - 同じディレクトリ内に入れておけばOK

乗降客数: https://www.odakyu.jp/company/railroad/users/
駅の場所: https://ja.wikipedia.org/wiki/小田急小田原線　より引用

使用イメージ

```
OdakyuData od = new OdakyuData();
for (int i = 0; i < od.numStations(); i++) {
  String[] sstr = od.station(i);
  String name = sstr[0];
  int passengers = Integer.parseInt(sstr[1]);
  double pos = Double.parseDouble(sstr[2]);
  // name, passengers, posがi番目の駅の情報
  // これをうまく保存して処理する
}
```

課題指示

- OdakyuDataを使い、OdakyuSortクラスに駅を表示するvoid printNeighbor(String)を作成する
  - 引数に駅の名前xを受け取る
  - 駅xからの距離が近い順に、5つの駅について、駅の情報を表示する
    - 駅xは表示に含まない(最も近いものと扱わない)
    - 駅の情報表示はStation.toString()の結果を表示する
    - 1行に1つずつ駅を表示
- OdakyuSort.main()には、printNeighbor()を使い、「生田」と「代々木上原」から近い駅を表示するコードが書かれている。
- printNeighbor()や、それに必要なデータ構造を適宜追加して、望む機能を完成させよ。

ヒント

- まず、データを使いやすい形式で覚える
  - Stationクラスはある程度作ってあるので、適宜利用する
    - 使い方によっては追加する部分があるかもしれない
  - Stationクラスを要素とするコレクションを作ると良さそう
- 大きく2つの機能が必要そう
  - 名前から該当する駅を見つける（座標も見つかる）
    - データをなめて探してもいいけど、簡単な方法もありそう
  - ある座標を入力とし、その座標からの距離順で表示する
    - ソートを使えばよい
- やりたいことがちょっと違うので、コレクションを2種類使うのも良さそう
  - ひな型ではListを使った場合を参考として書いてある。これだけで良いか、何か別のコレクションを使う方が良いか、組み合わせるか、など自由に変えて考えてよい。
- 距離は「座標の差の絶対値」で求まる
- compare()はintを返す必要があるので、距離をdoubleで計算しているときに返り値をどう計算するかに注意

提出物

- 提出物はOdakyuSort.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。
  - OdakyuSort.javaには、OdakyuSortクラスと、実行に必要なクラスを書いておく(OdakyuData以外)
    - Stationクラスは必要ですよね
  - package javalec12　とする

- ㌔切は7/16(火) 17:00