Java演習
第10回

2023/6/21

横山大作

講義前・後の質問

- 横山まで
- dyokoyama@meiji.ac.jp

提出課題8: エラー入りの文字列（再掲）

- 数字をたくさん文字列で読み込んだ
  - ファイルに書いてあったと思ってください
- PersonData.persons に数字の文字列が配列で入っている
  - persons = {“357”, “25849”,… }; みたいに
- この数字の平均を求めたい
- Integer.parseIntを使ってそれぞれの文字列をintに直し、平均を計算しようと思う
- ところが！
- どこかの文字列に変な文字がくっついていて、parseIntがエラーを吐いている
  - 試してみてください

やること

- PersonData.personsの文字列配列をそれぞれintに変換して、平均を求めて表示する
- ただし、Integer.parseIntで変換できない文字列は無視して平均を計算する
  - エラーの文字列があると、平均を計算するデータの個数も減るように扱う。
- 変換できない文字列があったとき、”Error in: 3”のように、何個目のデータにエラーがあったかを表示する
  - 配列の最初を「0個目」として数えることにする
- 全部の文字列を調べ終わったら、平均値を”Avg: 58.32”みたいに出力する
  - 出力フォーマットはお任せします
  - 小数点以下一位程度は出してください

基本構造

```
public class ExceptionTest {
    public static void main(String[] args) {
        String[] ps = PersonData.persons;
        int sump = 0;
        for (int i = 0; i < ps.length; i++) {
            int p = Integer.parseInt(ps[i]);
            sump += p;
        }
        System.out.println(String.format("Avg: %.2f",
            (double)sump / ps.length));
    }
}
```

- ここまでは素直に書けますね

例外処理

- エラーが出た行はエラー処理をする、ということは、行ごとにtry-catchを行えばよい

try-catch

```
public class ExceptionTest {
    public static void main(String[] args) {
        String[] ps = PersonData.persons;
        int sump = 0;
        for (int i = 0; i < ps.length; i++) {
            try {
                int p = Integer.parseInt(ps[i]);
                sump += p;
            } catch (NumberFormatException e) {
                System.out.println("Error in:"+i);
            }
        }
        System.out.println(String.format("Avg: %.2f",
            (double)sump / ps.length));
    }
}
```

- 1行のデータ処理をtry-catchで囲めばOK

このままだと

- 平均がおかしいですね
  - エラーの個数を数えて除外すればOK

解答例

```
public class ExceptionTest {
    public static void main(String[] args) {
        String[] ps = PersonData.persons;
        int sump = 0;
        int num_e = 0;
        for (int i = 0; i < ps.length; i++) {
            try {
                int p = Integer.parseInt(ps[i]);
                sump += p;
            } catch (NumberFormatException e) {
                System.out.println("Error in:"+i);
                num_e++;
            }
        }
        System.out.println(String.format("Avg: %.2f",
            (double)sump/(ps.length-num_e)));
    }
}
```

- もちろん、正しく読めた個数を数えても可
- どこでカウントするかは気を付けて！
  - 例外が発生したところからcatchへ飛ぶ

```
int sump = 0;
int num_read = 0;
for (int i = 0; i < ps.length; i++) {
    try {
        num_read++;
        int p = Integer.parseInt(ps[i]);
        sump += p;
    } catch (NumberFormatException e) {
        System.out.println("Error in:"+i);
    }
}
System.out.println((double)sump/(num_read));
```

まだ読めてないはず

これだと誤答

誤り例1

- for文全体をtry節で囲ってしまう間違いがよく見られました。これだと、for文の中で1回でも例外が起きたら、それ以降のデータは読まれなくなります。
- 別のパターンとして、例外が出た以降の、残ったtry節の中の文は実行されないことを見逃したミスもありました。

誤り例2: 浮動小数点と整数値が混ざった演算に注意

- double avg; int sum; int num; という状態で、
  avg = sum / num;
  を計算すると、
  - 最初にsum / numがintで計算される
  - それをdoubleへ変換しながらavgに入れる（よって、小数点以下は.0になる）

  という動きになります。
- もし、浮動小数点で計算したいのであれば
  avg = ((double)sum) / num;
  - // 慣れてきたら avg = (double)sum / num; でも良い

  にしましょう。こちらだと
  - 最初にsumがdouble型に変換された数字ができる
  - それをnumで割るので、numもdouble型に変換され、doubleで計算が行われる

  という動きになります。

誤り例3

- 整数で平均を計算しておいて、表示の時点でString.format("%.2f", avg)のように表示だけ変える、というのは論理的におかしいですね。最後に小数第2位まで必要なら途中計算もその精度が出るようにする、というのが誠実な姿勢です。

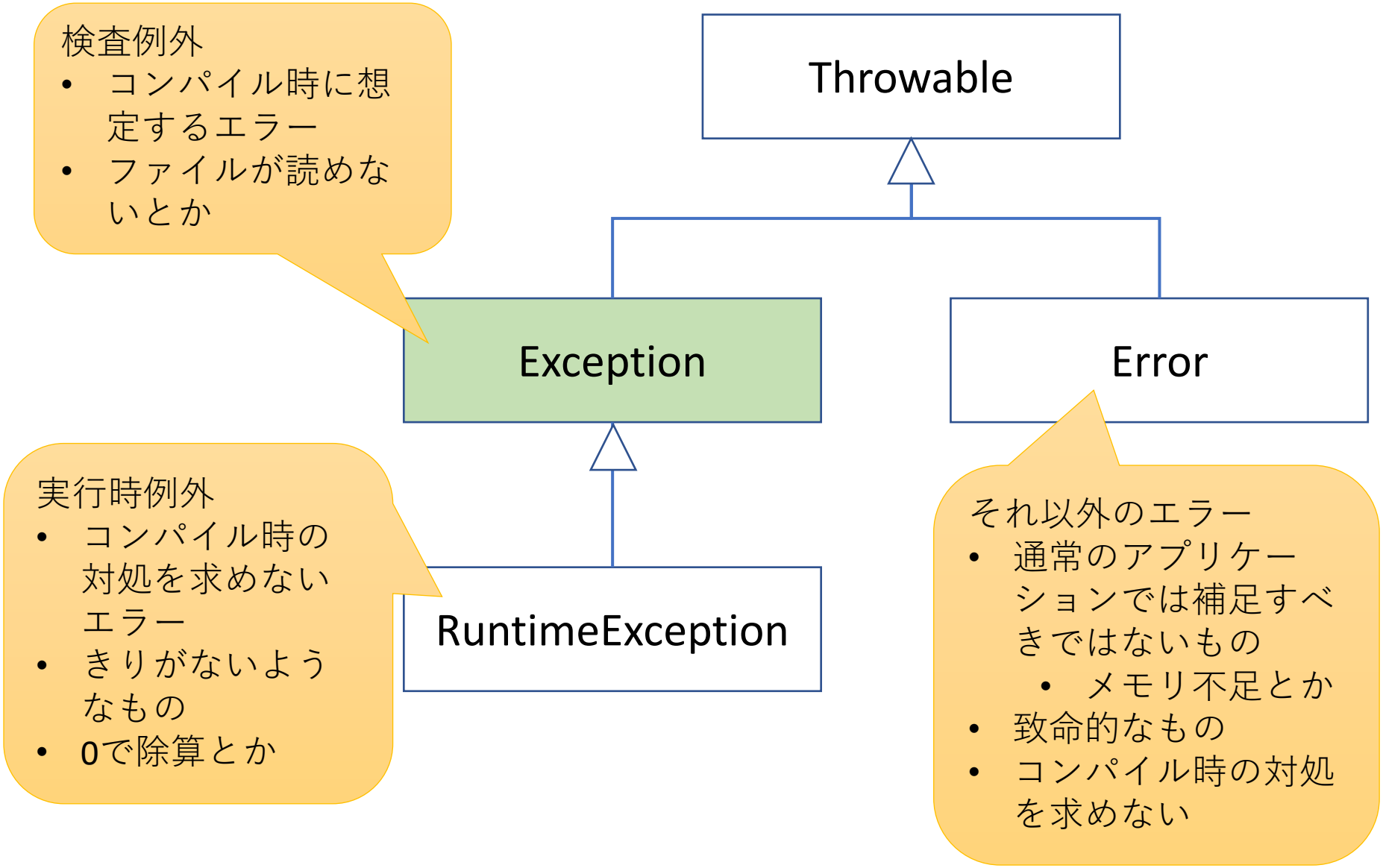
この資料の内容

- 例外とアサーション
- クラス色々
  - 関数オブジェクト
  - ソートの例

例外 (p.614)

- 例外の種類について復習

例外クラスの階層構造（再掲）



例外への対応

- チェック例外: 対応しないとコンパイルエラー
  - キャッチかスローが必要
    - 自分で処理するか、上に投げることをクラス宣言に書くか
  - Exceptionクラス（のサブクラス）
- 非チェック例外: 対応しなくてもコンパイルは通る
  - RuntimeExceptionまたはErrorのサブクラス
- p.628 APIが投げる例外はリファレンスに書いてある

```
void func() throws IOException { ... }
```

```
void func() {
  try { ...
  } catch (IOException e) {
    ...
  }
}
```

例外を作る (p. 653)

- 例外はクラス
- 自分で例外を作れる

- 通常、Exception か RuntimeException を継承
  - チェック例外か非チェック例外か
- 例外には普通あるもの、を理解しておく
  - メッセージ文字列：表示の時に使う
    - getMessage()
  - 例外の原因になった元の例外がある場合、覚えておくと良い
    - getCause()
  - printStackTrace()もよく使う

- 作り方のセオリーを学んでおこう
  - メッセージ文字列、原因例外を引数に持つコンストラクタを作る
    - コンストラクタは継承されないから

作り方のセオリー

例外にありそうなコンストラクタは作っておこう（継承されないので）

```
class MyException extends Exception {
  public MyException(String message) {
    super(message);
  }
  public MyException(Throwable cause) {
    super(cause);
  }
  public MyException(String message, Throwable cause) {
    super(message, cause);
  }
  // ...
}
```

例外はあくまで例外

- 普通の処理の制御と混ぜてはいけない
  - 例えば、2種類の型の値を返したいときに使うとかは推奨されない
    - returnで物足りないのであれば、そもそものメソッドの設計がおかしいかも？
  - 単に便利なジャンプ、ではない

catchの順序

- catchを書いた順番に「代入可能か？」をチェック
  - そのクラス、またはサブクラスなら代入可能
- 例外のクラス階層（継承関係）に注意
  - サブクラスは親クラスのcatchに含まれる

```
} catch (IOException e) {
  // . . .
} catch (FileNotFoundException e) {
  // . . .
}
```

「FileNotFoundExceptionはすでにキャッチされています」というコンパイルエラー

オーバーライド

```
void func() throws IOException { ... }
```

- オーバーライド時、親のthrows宣言と異なる対応が必要になる例外をthrows宣言するのはダメ
  - 同じクラスやサブクラス例外を投げると宣言するのはOK
  - 例外を投げないと宣言するのもOK
    - 使う側は新たな対応は求められないから
  - 非チェック例外はコンパイル時にチェックされないのでthrows書いてもOK

サブクラスだからOK

非チェックだからOK

```
void func() throws FileNotFoundException, IllegalArgumentException
```

```
void func()
```

投げない宣言もOK

アサーション

- デバッグ用の仕組み
- 動作時（実行時）にチェックする事柄をあらかじめプログラムに記述したもの
- 様々な言語でみられる、一般的な仕組み

- テストの時だけON
- 十分テストした本番ではOFF　　とかが可能
  - OFFにすると、パフォーマンスへの影響を抑えられる

assert

- assert 条件式;
- assert 条件式 : 引っかかった時の式;

- 動作時にチェックON/OFFできる
  - デフォルトではOFF
  - java -ea …というオプションを付けるとONに

- テストが通らないとAssertionErrorがthrowされる

```
void func(int num) {
  . . .
  assert num > 0 : “numが<=0になっています”;
  int k = 1000 / num;
  . . .
}
```

副作用 (アサーションでやってはいけないこと)

- プログラミング言語の専門用語
  - 特に関数型言語などで重要な概念
- その手続きが「本来やるべきこと」以外のこと

- テストが目的なのにフィールドを初期化するとか、そういうこと
  - Assertのon/offで動作が変わってしまう

クラス色々

ネストクラス

- クラスの中にクラスを作る
  - 今まで、意図せずにやっていた人いますか？
- あまり使われないが、理解しないでうっかり使うと困惑する
- 3通りがあるので以降紹介
  - メンバクラス
  - ローカルクラス
  - 無名クラス
  - 使うなら意図通りに

```
class StationTest {
  . . .
  class Station {
    . . .
  }
  . . .
}
```

メンバクラス

- クラスのメンバとして宣言されたクラス
- staticかそうでないかで振る舞いが変わる

非staticメンバクラス

- 外側のオブジェクトのインスタンスにくっついているクラス
  - 外側のインスタンスのメンバへのアクセスが可能
  - 外側のインスタンスがないと作れない

```
class Outer  {
  class Inner {
    int i;
  }
  void f() {
    Inner in_o = new Inner();
  }
  static void sf() {
    Outer out_o = new Outer();
    Inner in_o = out_o.new Inner();
  }
}
```

メソッド内はthisにくっついてnewされる

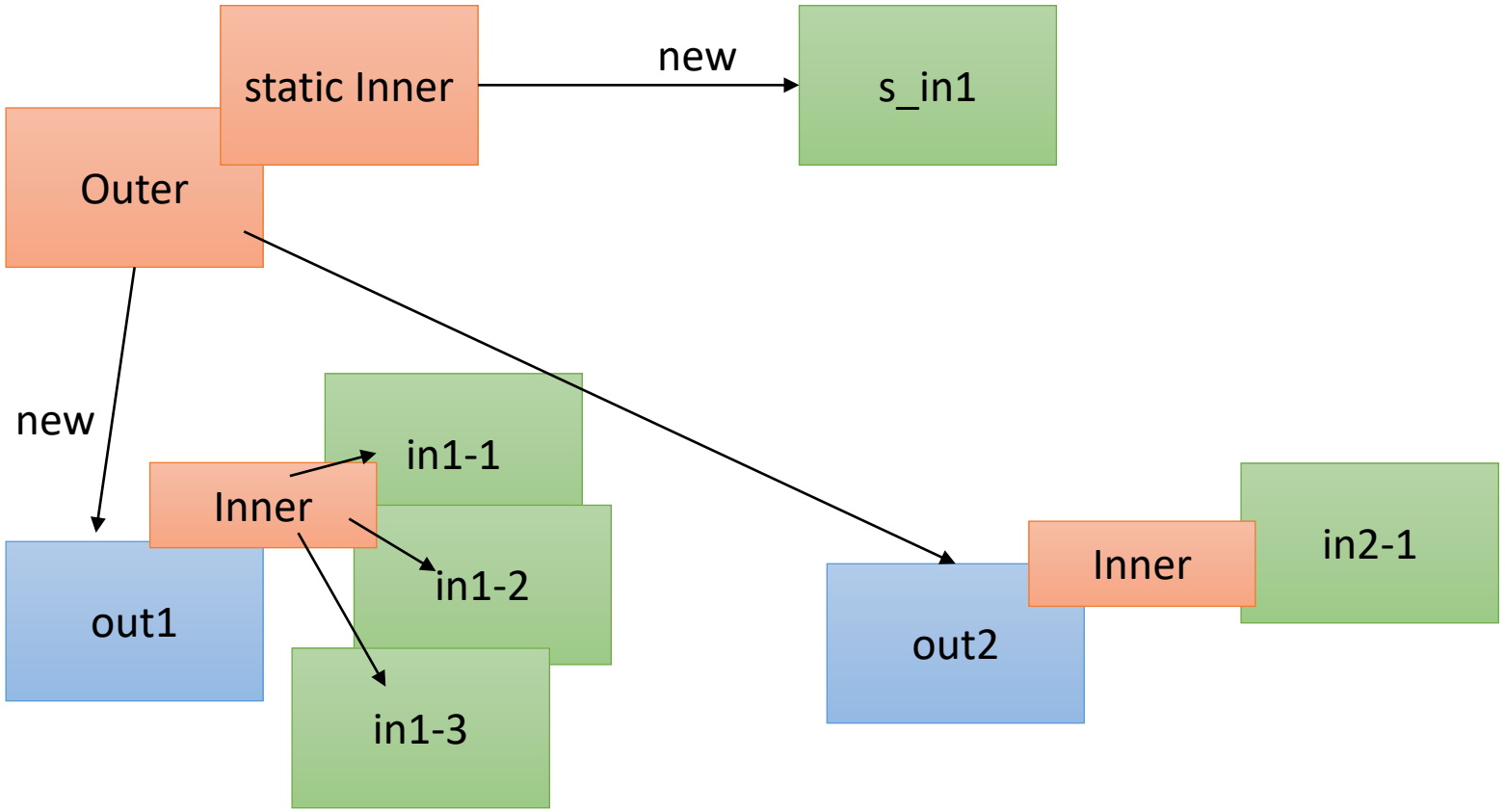
staticメソッド内はくっつくインスタンスが必要
不思議な見た目ですが

staticメンバクラス

- 外側クラスにくっついたクラス
  - 外側のstaticなメンバにはアクセスできる
  - いきなり(外側なしで)newできる
  - ほとんど独立したクラス

```
class Outer  {
  static class Inner {
  }
  static void sf() {
     Inner in_o = new Inner();
  }
}
class Ex {
  void f() {
    Outer.Inner in_o = new Outer.Inner();
  }
}
```

メンバクラス


static Inner
new
s_in1
Outer
new
in1-1
Inner
in1-2
in1-3
out1
Inner
in2-1
out2

メンバクラスの使いどころ

- このクラスからしか使わない処理をまとめる
  - privateにすると良さそう
- あまり非staticメンバクラスは推奨されない
  - 内側->外側への参照が残るから
  - 後でガベージコレクションの時にまた述べます

ローカルクラス

- メソッドの中で定義したクラス
- メソッド内の変数と同じような感覚
  - メソッドを出れば使えなくなる
- 局所的に使うクラスを定義するとき
  - 例えば、インタフェースを継承したクラスを一瞬使うときとか

```
void func() {
  . . .
  class TempClass {
    . . .
  }
  TempClass t = new TempClass();
}
```

無名クラス

- クラスを継承したり、インタフェースを実装して新しいクラス（のインスタンス）を作る時、1か所でしか使わないようなら名前を付けずに使える
  - ひな形がある場合に使える感じ
- よく使う
- 抽象クラスやインタフェースはこれを想定していることも多い

- コンストラクタは書けないが、初期化子は使える

```
Arrays.sort(a,
    new Comparator() {
        public int compare(Object l, Object r) {
            return ...;
        }
    }
);
```

ローカルクラスや無名クラスの典型的な使い方

- ソートに渡す比較用オブジェクト

関数オブジェクト

- 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

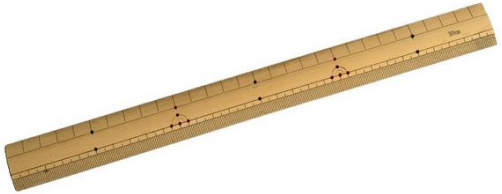
- 削る

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

Arrays.sort()の例

```
String[] a = new String[] {
    "aaa", "bbb", "aab", "a", "aax", "x"
};

Arrays.sort(a, null);
```

ここにComparatorインタフェースを備えた関数オブジェクトが入れられる

- import java.util.Arrays;

Comparatorインタフェース

```
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);
```

これはローカルクラス

- ここだけ別基準でソート、ということが簡単にできる

無名クラスを使うと

```
String[] a = new String[] {
    "aaa", "bbb", "aab", "a", "aax", "x"
};

Arrays.sort(a, new Comparator() {
    public int compare(Object l, Object r) {
        return ((String)l).length() - ((String)r).length();
    }
});
```

これは無名クラス

- 読み方ちょっと慣れが必要ですが、ずいぶん簡単に書けますね

- 関数オブジェクトは大事なので、また後日じっくりとやります。お楽しみに。

提出課題9: 配列のソート

- 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

要求

- ArrayDat.numstrings にStringの配列が入っている
- これを使って、配列をソートして1行1つずつデータを表示する。
- 2通りのソート結果を、どちらのソート結果なのか一行説明付けて、続けて表示する。
- 右のような感じ

<辞書順ソート>
XX
XX (数字文字列を1行1つずつ表示) . . .
<数としてソート>
XX
XX . . .

ヒント

- Arrays.sort()というメソッドを調べて使ってみよう
  - import java.util.Arrays; が必要なのを忘れずに
- sortにはComparatorを実装したクラスのインスタンスが渡せましたね
  - import java.util.Comparator; が必要なのを忘れずに
  - やり方は今回の講義で2通りくらいやりました。色々試してみるのが良いでしょう。
- Comparator.compare()はObject型を引数に取るので、比較するときにはキャストが必要になります
  - 「パラメータ化してください」みたいな警告についてはまた後日説明しますので、今回は無視で。
- 辞書順はStringのデフォルトのcompareTo()が返す順番です
- 文字列を数字として見るにはInteger.parseInt()とかが使えそうですね

提出物

- 提出物はArrayTest.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 javalec9　とする
  - ArrayTest.main()を呼び出したら課題の結果が表示されるようにする

- ㋈切は6/25(火) 17:00