

Java演習

第9回

2024/6/12

横山大作

講義前・後の質問

- 横山まで
- dyokoyama@meiji.ac.jp

提出課題7: 商品価格（再掲）

- 商品を表すクラス**Food, Book**がある
 - いずれも名前と価格を保持
- 本と食品をセット売りすることがある
 - 価格は合計から**100円**引き
- 色々な商品やセットをリストにして表示したい

商品を表すクラス

- 3つ
 - Food
 - Book
 - GoodsSet
- それらすべてをまとめて扱えるような interface PriceTagがある
 - String tagstr() // 表示のための商品名を返す
 - int price() // 表示のための価格を返す

実装例

満たすべき条件をinterfaceとして名前を付ける

```
interface PriceTag {  
    String tagstr();  
    int price();  
}
```

名付けたinterfaceを実装する

```
class Food implements PriceTag {  
    String name;  
    int price;  
    Food(String name, int price) {  
        this.name = name; this.price = price;  
    }  
    public String tagstr() {  
        return name;  
    }  
    public int price() { return price; }  
}  
// Bookも同様
```

もともとあった処理がそのまま使える。**public**にする必要アリ。

実装例

```
class GoodsSet implements PriceTag {  
    Food food;  
    Book book;  
    GoodsSet(Food food, Book book) {  
        this.food = food; this.book = book;  
    }  
    public String tagstr() {  
        return food.tagstr() + " と "  
            + book.tagstr() + " のセット";  
    }  
    public int price() {  
        return food.price() + book.price() - 100;  
    }  
}
```

それぞれのメソッドを呼び出す形
にしておくのが良い（フィールド
をじかに触らない）

- もちろん、**Food**と**Book**の共通項に気付いてそこで親クラスを作っても良いですね。
 - 例えば**Goods**とか
- その時、**GoodsSet**を「**Goods**2つ覚えるもの」に変更することもできそうです。
 - そうすると、セットとして可能な組み合わせはどう変わるでしょう？
 - 意図通りの表現になるでしょうか？
 - 問題文からは、どちらが正しいとかはちょっと読み取れないですね。自分で想像してみましょう。
- **GoodsSet**を継承の中に入れることも可能そうです
 - そうやって整理していく方法は、後期のオブジェクト指向で練習します。お楽しみに。

今回の課題の狙い

- **Food**と**Book**や、何か「組」のように、直接は継承関係にないクラスたちでも、インタフェースを使えば統一的に扱えるようになる
- 現実の場面で非常によく出てくる考え方です

この資料の内容

- 継承を「使っている」ものの例として「例外」を知る
 - 「例外」という仕組みについて知る
 - 継承のありがたみを知る
- 提出課題8

例外

例外 (p. 614)

- プログラムにおける普通の処理とは異なる「異常事態」を処理しようとするもの
 - 処理できるようにした機構のことを「例外」と呼ぶ

例: 0での割り算

```
class Sample {  
    public static void main(String[] args) {  
        int e = divtwo(5, 0);  
        System.out.println(e);  
    }  
    static int divtwo(int d0, int d1) {  
        return d0 / d1;  
    }  
}
```

Exception in thread "main" java.lang.ArithmeticException: / by zero
at javalec6.Sample.main(Sample.java:6)

- 試しに適当な数字を0で割ってみよう
- ここで起きているのが「例外処理」

何が起きた？

受け取る人がいるところまで
どんどん飛んでいく

```
class Sample {  
  
    public static void main(String[] args) {  
        int e = divtwo(5, 0);  
        System.out.println(e);  
    }  
  
    static int divtwo(int d0, int d1) {  
        return d0 / d1;  
    }  
}
```

呼び出し元へ
例外伝搬

おかしいことが起きると
例外発生

例外

- 例外は「投げる(throw)」もの
- 普通のプログラムの流れはそこでストップ、異なる制御が始まる
 - 呼び出し元へ、呼び出し元へと投げられ続ける
 - どこかで「受ける(catch)」人がいるまで
- mainの外側には「受ける(catch)」人がいる
 - 例外が起きたところの「スタックトレース」を表示する
 - どのような関数が呼ばれたところで、どんな例外が発生したか、をコンソールに表示

例外の種類

```
Exception in thread "main" java.lang.ArithmeticException: / by zero  
at javalec6.Ex_test.main(Ex_test.java:6)
```

例外の場所 (スタックの中身)

スタックトレースの見方

- 例外の種類はJava標準APIリファレンスを見ればわかる
 - `ArrayIndexOutOfBoundsException`
 - `NullPointerException`
 - などなど
- 例外が発生したところからmainまでの呼び出し関係が全部見える
- 例外はライブラリ内部から出るかもしれない
 - 自分が書いたコードでないところから発生
 - p.692を見て、読み方を確認しておこう
 - 書いてある「ここが怪しそう」という感覚は非常に重要！

try-catch文 (p. 632)

- 例外を扱うための制御構造

```
class Sample {  
    public static void main(String[] args) {  
        try {  
            int e = divtwo(5, 0);  
            System.out.println(e);  
        } catch (ArithmeticException e) {  
            System.out.println("例外キャッチしました");  
        }  
    }  
    static int divtwo(int d0, int d1) {  
        return d0 / d1;  
    }  
}
```

catchは複数書ける(p.633)

- catchに書いてあるのは「型の条件」
- 飛んできた例外のクラスにマッチする条件があればcatchできる
 - 上から順にチェックされる
 - | でつないでor条件も書ける

```
void func() {  
    try {  
        int e = divtwo(5, 0);  
        System.out.println(e);  
    } catch (ArithmeticException e) {  
        System.out.println("計算おかしい");  
    } catch (ArrayIndexOutOfBoundsException e) {  
        System.out.println("範囲おかしい");  
    }  
}
```

例外は「専用の制御構造」を作った

- 普通の処理とエラー処理はちょっと需要が違う
- エラー処理は、一気に移動したいとか、すべてのエラー処理はここにまとめたいとか、がち

例: 例外でないエラー処理

```
int func() {  
    ...  
    if () return NOT_VALID_STR_ERROR;  
    if () return NOT_VALID_DATE_ERROR;  
    ...  
    if () return 3;  
    return 0;  
}
```

エラーと普通の処理の見た目が同じ

普通の処理？エラー？

```
void g() {  
    ...  
    int e = func();  
    if (e > 10) {  
        switch (e) {  
            case NOT_VALID_STR_ERROR:  
                ...  
            case NOT_VALID_DATE_ERROR:  
                ...  
        }  
    }  
    // ここでやっと本来の処理...  
}
```

なぜ>10だとエラー？

もうeって何だか忘れちゃった

例: 例外にすると

```
int func() {  
    ...  
    if () throw new NotValidStr();  
    if () throw new NotValidDate();  
    ...  
    if () return 3;  
    return 0;  
}
```

明確にエラー

明確に普通の処理

```
void g() {  
    ...  
    int e = func();  
    // すぐに本来の処理...  
}
```

普段、エラーのことは意識して
ない

```
void h() {  
    try {  
        g();  
    }  
    catch (Exception e) {  
        ...  
    }  
}
```

いろんなエラーはまとめて処理

例外はオブジェクト指向を利用

- エラーはよく似ているがちょっと違うパターンが多い
 - 文字列がうまく日付に直せなかった、日付があり得ないものだった...
 - エラーを全部細かく番号で管理してたりすると大変
- ちょっとずつ違う処理をうまく扱いたい
- バリエーションがたくさんあるときにうまくまとめて処理書きたい

例: オブジェクトでないエラー処理

```
void g() {  
    ...  
    int e = func();  
    if (e > 10) {  
        switch (e) {  
            case NOT_VALID_STR_ERROR:  
                System.out.println("Not Valid str");  
                ...  
            case NOT_VALID_DATE_ERROR:  
                System.out.println("Not Valid date");  
                ...  
        }  
    }  
}
```

結局文字列表示するだけなんだけど...

別の関数にも同じこと書くのか...

えっ、エラー追加ですか？

例: オブジェクトによるエラー処理

```
void h() {  
    ...  
    catch (Exception e) {  
        System.out.println(e.getMessage());  
        ...  
    }  
}
```

e自身に文字列表示はおまかせ

継承を使えばまとめて処理も簡単

エラー追加? エラー側で新しい文字列準備してね

エラー処理と継承

- 例外は階層構造を持っている
 - 大まかな分け方：Exception
 - 細かい分け方：ArrayIndexOutOfBoundsException, NumberFormatException, ...
 - 中間の分け方があることも
- エラー処理の時、まとめてよければ「親クラス」でcatchする
 - 極端な例がExceptionクラスでのcatch (p.633)
 - 実は例外はほぼすべてExceptionクラスの子供だから
- 細かい違いに興味があれば子クラスでcatchする

```
void func() {  
    try {  
        int e = divtwo(5, 0);  
        System.out.println(e);  
    } catch (ArithmeticException e) {  
        System.out.println("計算おかしい");  
    } catch (ArrayIndexOutOfBoundsException e) {  
        System.out.println("範囲おかしい");  
    }  
}  
}
```

```
void func() {  
    try {  
        int e = divtwo(5, 0);  
        System.out.println(e);  
    } catch (Exception e) {  
        System.out.println("何かおかしい");  
    }  
}
```

参考

- `catch`は上から順番にチェックされていく
- 先に大まかに`catch`する節を書くとその下には来ない
 - コンパイルエラー

```
void func() {  
    try {  
        int e = divtwo(5, 0);  
        System.out.println(e);  
    } catch (Exception e) {  
        System.out.println("何かおかしい");  
    } catch (ArrayIndexOutOfBoundsException e) {  
        System.out.println("範囲おかしい");  
    }  
}
```

Unreachable catch block for
`ArrayIndexOutOfBoundsException`. It is already
handled by the catch block for `Exception`

例外でのオブジェクト指向の 使われかた

- 例外はオブジェクト
 - 中に様々な情報をくるんでいる
- お任せできるメソッドが用意されている
 - ポリモーフィズム
 - `getMessage()`, `printStackTrace()`など
- 継承の階層を生かして分類できる
 - 処理を大きく分けたり、細かく分けたりが自由

チェック例外 (p. 625)

- コンパイル時に「きちんと処置書いて！」と言われる例外と、言われない例外がある
- 対処すべき例外はコンパイル時にチェックするぞ、というJava設計者の意志
 - 対処しないとコンパイルできない
- 対処の仕方は2通り
 - そのメソッド内で対処する
 - try-catch
 - そのメソッドから上へ投げる、と宣言時に明示する
 - 上の人誰かが対処しなければならない、と指示
 - `void func() throws InterruptedException { ... }`

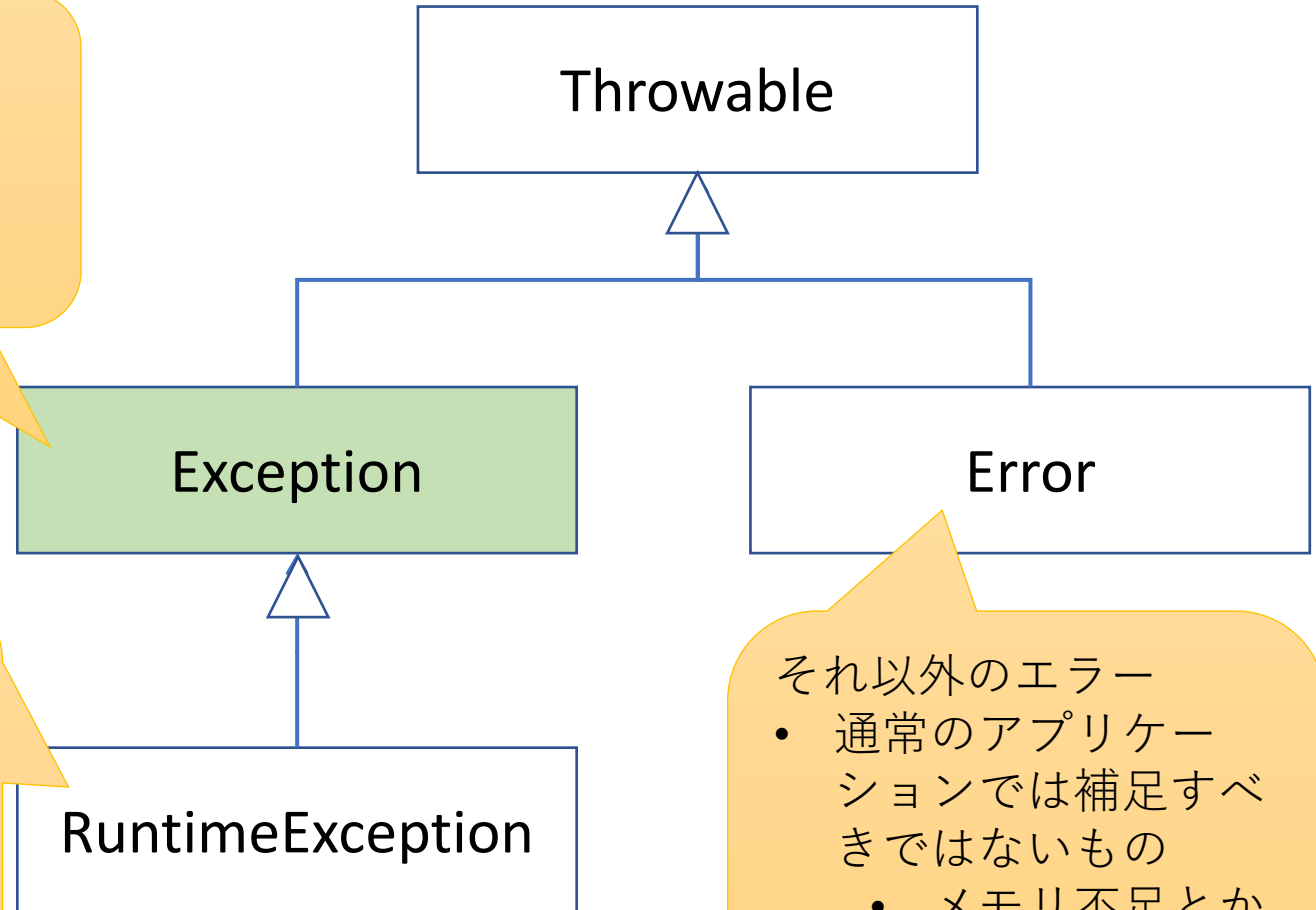
エラーの階層構造

検査例外

- コンパイル時に想定するエラー
- ファイルが読めないとか

実行時例外

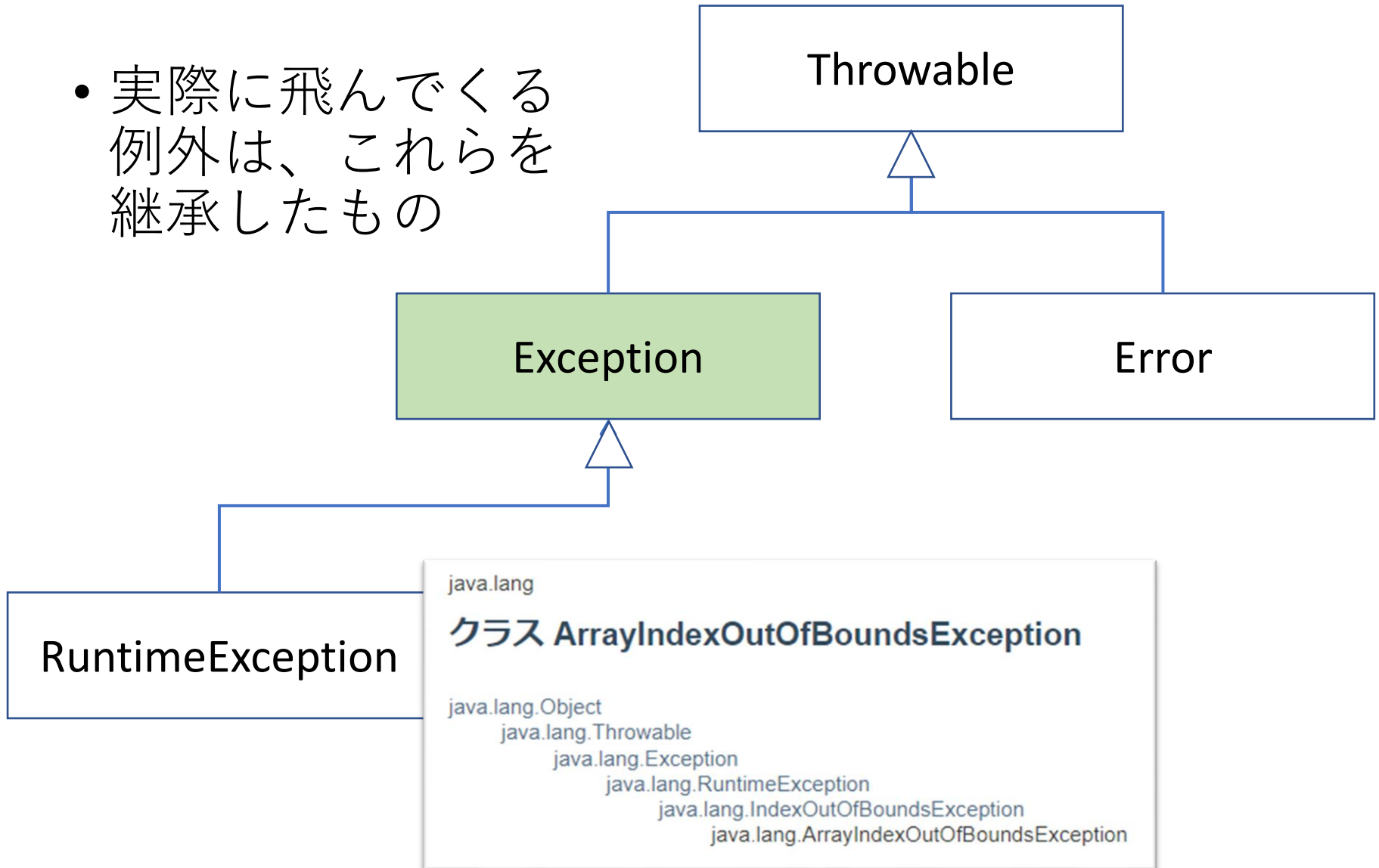
- コンパイル時の対処を求めないエラー
- きりがいいようなもの
- 0で除算とか



それ以外のエラー

- 通常のアプリケーションでは補足すべきではないもの
 - メモリ不足とか
- 致命的なもの
- コンパイル時の対処を求めない

- 実際に飛んでくる例外は、これらを継承したもの



例外クラスのメソッド (p.630)

- **Exception** クラスを継承していれば（＝普通に飛んでくる例外オブジェクトなら）この辺のメソッドが使える
 - コンストラクタ（引数なし）
 - コンストラクタ（メッセージ文字列の引数あり）
 - `void printStackTrace()`
 - `String getMessage()`
- 例外を受けて行いたい処理に使いやすい部品群

finally (p.634)

- エラーが起きても必ずやらなければならない処理がある
 - データベースの処理とか、メモリの処理とか
 - finallyというブロックで記述できる

エラー処理は変態になりがち

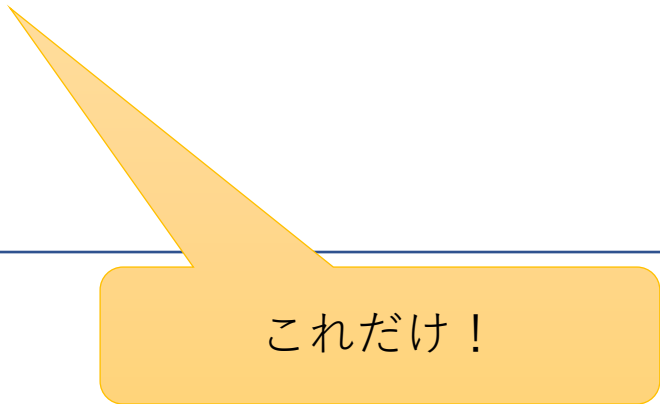
```
InputStream is = null;
Try {
    is = Files.newInputStream(path);
    is.read(buf);
} catch (IOException e) {
    ...
} finally {
    if (is != null) {
        try {
            is.close();
        } catch (IOException close_e) {
            // 何もすることない
        }
    }
}
```

- p.637から詳細なストーリーがあります

try-with-resources文(p.644)

- Java7から導入
- `java.lang.AutoCloseable`などが使えるクラスはtry-with-resources文で自動的に`close()`される

```
try (InputStream is = Files.newInputStream(path)) {  
    is.read(buf);  
} catch (IOException e) {  
    ...  
}
```



これだけ！

例外をどう処理する？

- 表示する
 - スタックトレースを出力
 - 簡単に表示（例外を直接printlnする）
- 例外を食べる
 - 何もしない、握りつぶす　ということ
 - `catch`に何も処理を書かない
 - エラーに気付けなくなるから、原則、やってはいけない
 - p.650

例外の伝搬(p.646)

- 例外の処理を、その場でやらない方法
- 上の人にお任せ、を宣言
 - `void func() throws InterruptedException { ... }`
- 結局、どこかではtry-catchなどで対応しないと
いけない

受けるだけじゃない(p.651)

- 例外を自分で投げることも可能
- **throw**キーワードを使う
- 例外の実体はインスタンス
- 自分で例外のインスタンスを**new**して**throw**する

```
void func() {  
    ...  
    if (something_wrong) {  
        IllegalArgumentException e;  
        e = new IllegalArgumentException("おかしい");  
        throw e;  
    }  
  
    ...  
}
```

例外まとめ

- エラーを明確に処理する「例外」機構がある
- 例外はオブジェクト
- 例外は**throw**する
- 例外は**try-catch**で受ける
- 例外クラスは階層構造になっている
 - 継承を使っている

提出課題8: エラー入りの文字列

- 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（そのまま）`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”のように出力する
 - 小数点以下2桁まで表示しましょう
 - `String.format()`を調べて使ってみましょう

提出物

- 提出物はExceptionTest.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。
 - 出力が貼ってない場合減点します。採点がしんどいのでご協力お願いします。
 - package javalec8 とする
 - ExceptionTest.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6/18(火) 17:00

ヒント

- 変な文字列を読んだ時、起きているエラーは「例外」です
- **try-catch**でうまくエラーを捕まえて、エラーの時に例外処理をしよう
- 正常に読めた文字列の個数（または、エラーが発生した数）を覚えておかないと、平均を計算するためのデータ数がわからないことに注意
 - 正しいデータの個数は提出前に目で見て確認しよう。意外と間違いが入ります。
- 細かいところでは、割り算を整数で行うと平均値がいい加減になりすぎるので注意