

# デバッグ・インポート・高速化

2018/6/26(火) プログラミングIV 第十一回  
福井大学 工学研究科 情報・メディア工学専攻  
長谷川達人



# 演習問題の解説

- HP上にアップロードした.
- 点数に**不服がある人**は個別に申し立ててください.
- 不服申立によって第9回の再採点を行いました.
  - ほんの一部, 5点くらい上がっているかもしれません.
- 前回の解説を今から行います.

# まとめ

- ファイルの入出力を行うクラスの使い方を学習した.
  - 書き込み : FileWriter, BufferedWriter

```
FileWriter fw = new FileWriter( [出力先ファイルパス] );  
BufferedWriter bw = new BufferedWriter( [FileWriterインスタンス(この場合 fw)] );  
bw.write( [出力する文字列] );  
bw.newLine();    // 適切な改行コードを自動で生成  
bw.close();
```

- 読み込み : FileReader, BufferedReader

```
FileReader fr = new FileReader( [入力元ファイルパス] );  
BufferedReader br = new BufferedReader( [FileReaderインスタンス(この場合 fr)] );  
String str = br.readLine();    // 1行単位で読み込む  
System.out.println(str);  
br.close();
```

# まとめ

- try-with-resources文を用いてclose()を安全に省略する手法を学習した.

```
try(FileReader fr = new FileReader("file.txt")){  
    int ch;  
    while((ch = fr.read())!=-1){  
        System.out.print((char)ch);  
    }  
}catch(IOException e){  
    e.printStackTrace();  
}
```

1. try時にリソースを開くだけ

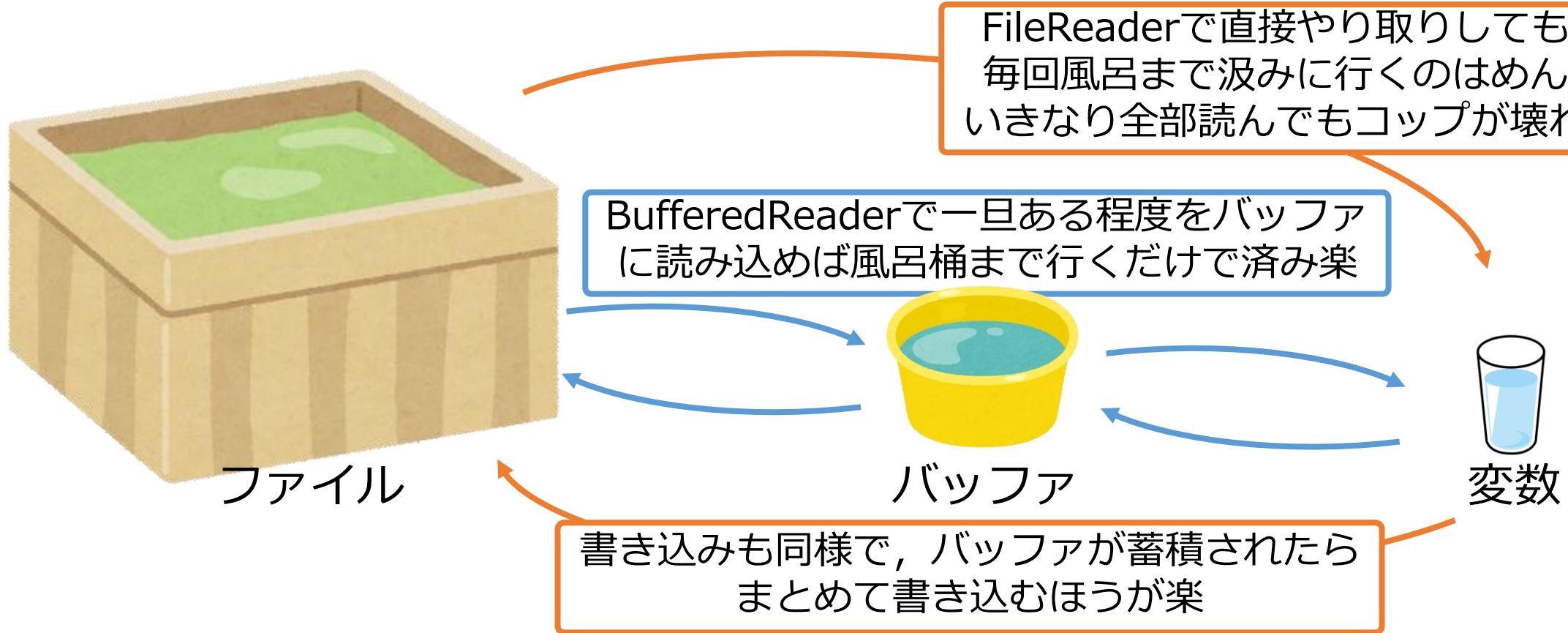
- オブジェクトをシリアライズしてファイルに保存しておく手法を学習した. (**implements Serializable**)

# ファイル入出力

## バッファ



**バッファ**とは、一時的に情報を保存する領域等の総称である。



# 質問への回答

- Buffered \* \* \* があるのに File \* \* \* って必要？
  - それぞれで役割分担を行っていて、使い方に応じてうまく使い分けられるようにしている様子である。実は他にも色々読み書き系のクラスはあって、今回は一例ということ。
- Buffered \* \* \* を使わないほうがいい場面はあるのか。
  - 基本的には Buffered \* \* \* を使ったほうが速いし良い（毎回ファイルにアクセスするのは非効率なのでバッファを挟む）。
  - 速度の差は以下。 1,000,000文字 \* 10セット 平均時間[ms]

FileWriter	BufferedWriter	FileReader	BufferedReader
95.3(±14.8)	69.1(±12.1)	233.2(±55.3)	46.0(±8.1)
3/4		1/5	

# 質問への回答

- バイナリ形式で保存すると人間にメリットはあるのか.
  - ファイルサイズが小さく, 機械が読み書きしやすい (高速).
  - 一方近年は容量が問題になりづらく, 逆に互換性を求めてテキストが望まれる機会もあるようだ.
- バッファはどこまでデータを貯められるのか.
  - メモリある限り.
- バッファの使いどころがわからない.
  - ダブルバッファリングのバッファも同じ意味である.
  - 意図して使うとか実装するという機会があるわけではなく, `BufferedReader`のように内部処理で使ってくれていることが多い. バッファを使うことでやはり高速なのが良い. 本日高速化でやる `StringBuilder`もバッファが使われている.

# 質問への回答

- 以下のプログラムの意味がわからなかった.

```
try(FileReader fw = new FileReader("fr.txt")){  
    int ch;  
    while((ch=fw.read()) != -1){  
        System.out.println((char)ch);  
    }  
}catch(Exception e){}
```

```
try(FileReader fw = new FileReader("fr.txt");  
    BufferedReader bw = new BufferedReader(fw)){  
    String str;  
    while((str=bw.readLine())!=null){  
        System.out.println(str);  
    }  
}catch(Exception e){}
```

どちらも,   を実行後, その結果に対して   を実行する.

```
while((ch=fw.read()) != -1){}
```



```
1. ch=fw.read();
```



```
2. ch != -1
```



**fw**から1文字ずつ**ch**に格納し,  
その結果が-1出ない間繰り返す.  
(-1はファイル終端を意味する.)



# 質問への回答

- finallyは今後もう使わない？
  - ファイルのようにリソースを開いて閉じる処理の例外対応としてのfinallyはtry-with-resourcesで対応可能.
  - しかし, その他のケースでtry-catchの一番最後にならずやりたい処理をfinallyに書くことは大いに有り得る.
- try-catchがめんどくさい. 必要性がわからない (6件).
  - try-catchが面倒なのは, 今君たちが異常系の処理を気にしないでよい環境で開発しているからである. 一方, 就職後を考えてほしい. 開発したソフトが例外で強制終了してデータが消えた場合, 顧客 (私の場合は医師だった) にぼろくそ言われることは間違いない. そんな環境でもこのコメントが言えるだろうか.

# 質問への回答

- try-with-resources文はなぜ自動でclose()するのか.
- なるべくtry-with-resources文を使ったほうが良いのか.
  - 従来のclose()をfinallyに記述する面倒を簡略化するために開発されたのがtry-with-resources文なので簡略化のため.
  - こちらを使ったほうが, 可読性が上がり, close()忘れも減るので良いこと尽くしだと思われる.
- ゲームのセーブファイルも今回の応用でできるのか.
  - まさにその通り. セーブや設定ファイル等々に使える.
- コピペプログラマーについてどう考えるか.
  - 作りたいものが作れば, 全然問題ない. コピペした後に内容を理解している場合に限る. ただし, 講義内での他者からのコピペは断罪に値する.

# 質問への回答

- ファイルが存在する場合の対応，先生はどっち派？
  - 上書きか，別名かという意味ととらえると，状況によりけり．
  - 過去のデータに意味がなければ容量削減と可読性から上書き，過去のデータに意味があったり，以前の状態に戻す可能性があれば別名保存．別名は現在時刻を名前に加えると重複しない．
- 一般的に使われているJavaのバージョン
  - 意外と出回っているシステムでJavaのバージョンが更新されていないケースがある（コスト等の都合）．個人的印象としてはJava7か8が強め．（Java7はサポート終了済み，8も1月まで）
- 基本的にFile \* \* \* よりBuffered \* \* \* を使えばよい？
  - とりあえずそれでよい．

# 質問への回答

- 参考書5冊の名前を教えて
  - スッキリわかるJava入門 : わかりやすい. おすすめ.
  - 新わかりやすいJava入門編 : 一般的な入門教科書. 上ので十分.
  - Java実践編アプリケーション作りの基本 : 上の+aを求める時に.
  - Java本格入門モダンスタイルによる... : マニアな知識をお求めの方.
  - JavaSE8 Silver問題集 : Oracleの資格に興味があれば.
- ファイルの事前チェックが多くて忘れそう.
  - 確実な例外対応には, 事前チェックを行うほうが良い. しかし  
いざ対応するときに使えれば良いので暗記までしなくて良い.
- 例外処理はすべてtry-with-resources文でいいか.
  - close()が必要となるリソース系例外はtry-with-resourceでよいが, その他は普通のtry-catchを使う.

# 質問への回答

## 自由課題について

- 音が出るアプリはありますか？
  - 一応，演習室PCは音が鳴ったと思う．ただ，マイクを近づけないと音を拾いづらいかもしれない．
- 自由課題の6割の基準は？
  - よっぽどひどくなければ6割はあると思う．
  - 初の試みなので基準を伝えづらいが，各回の演習課題レベルのものであれば6割はある．
- 自由課題の発表で1分を超えたら減点か．
  - 減点というか，1分超えた地点で発表を打ち切る．

# 本講義の概要

前半		後半	
第1回	基本文法の復習	第9回	標準ライブラリ
第2回	クラス～カプセル化の復習	第10回	ファイル入出力
第3回	抽象クラス, インタフェース	第11回	<b>デバッグ, インポート, 高速化</b>
第4回	ポリモーフィズム	第12回	オブジェクト指向
第5回	GUI 1	第13回	自由開発演習 1
第6回	GUI 2	第14回	自由開発演習 2
第7回	スレッド, 例外処理	第15回	自由開発演習発表会
第8回	ジェネリクス, コレクション	第16回	期末試験

何を開発するか, 少しずつ考えておくこと

# 目次

## めずらしく目次

- デバッグ

- Keyword: デバッグ, デバッガ, ブレークポイント

- リファクタリング

- Keyword: リファクタリング

- 外部ライブラリ

- Keyword: ビルド・パス

- 高速化

- Keyword: 分岐をさせない, HashMap, StringBuilder

# 本日の目標

## 概要

デバッグ・インポート・高速化等，これまで授業では触れてこなかったが，プログラミングにおいて重要な技術について学ぶ．

## 目標

デバッグ・インポートが使える．  
基本的な高速化処理を理解している．



なるほど



# 本日の提出課題

## 講義パート

課題を意識しながら  
講義を聞くと良い。

### 課題 1

本日の授業を聞いて、  
**よくわかった**と思う内容を2点簡潔に述べよ。

### 課題 2

本日の授業を聞いて、  
**質問事項**または**気になった点**を1点以上簡潔に述べよ。

### 課題 3

感想（あれば）

# 目次

## めずらしく目次

- **デバッグ**

- Keyword: デバッグ, デバッガ, ブレークポイント

- **リファクタリング**

- Keyword: リファクタリング

- **外部ライブラリ**

- Keyword: ビルド・パス

- **高速化**

- Keyword: 分岐をさせない, HashMap, StringBuilder

# デバッグ

デバッグとは

## デバッグ：バグを修正する作業



コードの形式的な誤りで、コンパイル時にエラーになる



コンパイルエラーに従ってデバッグする



実行中に、想定外の事態が発生し継続できず強制終了する



想定外の事態への例外処理を実装する

以前はココの話



実行はできるが想定していたものと異なる結果が返ってくる



原因を究明し、ロジックを改善する

本日はココの話

# デバッグ

デバッグとは

**デバッグ**：バグを修正する作業

*syntax*

文法  
エラー

*error*

コンパイルエラーを  
読めば、修正箇所が  
わかる。

*runtime*

実行時  
エラー

*error*

StackTraceを読めば  
例外発生箇所が特定可能  
稀に、根本原因を追いつ  
らい時がある。

*logic*

論理  
エラー

*error*

どこで発生したのかすら  
わからない時がある。

以前はココの話

本日はココの話

# デバッグ

## デバッグ手順

### 1. バグの再現条件を明確にする.

- 毎回必ず発生するのか？引数や変数に条件があるのか？

### 2. バグの発生箇所を明確にする.

- 例外落ちする場合, 例外の種類とファイル行数から推測する.
- しらない場合, ロジックが狂い始めた場所から徐々に遡る.

### 3. バグの発生原因を究明する.

- 発生箇所とロジックを熟考し原因究明を行う.

### 4. バグの修正を行う

- 周辺への影響を考慮しつつ, 原因への対策を実施する.

平均値を出力するプログラムavgを意気揚々と開発し他部署に納品したあなた。下記の通り、しっかりテストも実施しました。完璧だ！！

```
public class Main {  
    public static void main(String[] args){  
        int[] array = {1, 2, 3, 4, 5};  
        System.out.println(avg(array));  
    }  
  
    public static double avg(int[] array){  
        int sum = 0;  
        for(int i=0;i<array.length;i++){  
            sum += array[i];  
        }  
        return sum/array.length;  
    }  
}
```

[出力] 3.0



翌日他部署から、ちゃんと単体テストしたのか馬鹿野郎！！！！と罵声とともに例外落ちの動作例が送られてきました（優しい）。

```
1 public class Main {
2     public static void main(String[] args){
3         int[] array = new int[0];
4         System.out.println(avg(array));
5     }
6
7     public static double avg(int[] array){
8         int sum = 0;
9         for(int i=0;i<array.length;i++){
10             sum += array[i];
11         }
12         return sum/array.length;
13     }
14 }
```

他のケースは正常終了していたので、(1)バグの再現条件は本例の通りである。

(2)バグの発生箇所は  
Main:4行->Main:12行！

(3)バグの発原因は  
[java.lang.ArithmeticException:](#)  
/ by zero  
より、0除算の可能性！

[出力] Exception in thread "main" [java.lang.ArithmeticException: / by zero](#)  
at Main.avg(Main.java:12)  
at Main.main(Main.java:4)

# デバッグ

## よくある例外落ち

Arithmetic  
Exception

**算術エラー** ArithmeticException  
不正な算術処理が発生している（大体0で除算している）

NullPointerException

**ぬるぽ** NullPointerException  
Nullにアクセスした（大体インスタンスの初期化忘れ）

IndexOutOf  
BoundsException

**アウトオブバウンズ** IndexOutOfBoundsException  
インデックス範囲外にアクセスした（大体配列外参照）

FileNotFoundException

**ファイルノットファウンド** FileNotFoundException  
ファイルが見つからない



# デバッグ

## 練習問題 1 : デバッグせよ

```
1 import java.util.ArrayList;
2 public class Main {
3     private static ArrayList<String> array;
4     public static void main(String[] args){
5         add("TEST1");
6         add("TEST2");
7         add("TEST3");
8         add("TEST4");
9     }
10    public static void add(String str){
11        array.add(str);
12    }
13 }
```

質問対応していて  
非常によくあるケース  
(本来はもう少し複雑)

**[出力]** Exception in thread "main" [java.lang.NullPointerException](#)  
at Main.add([Main.java:11](#))  
at Main.main([Main.java:5](#))

翌日他部署から、ちゃんと単体テストしたのか馬鹿野郎！！！！と罵声とともに例外落ちの動作例が送られてきました（その2）。

```
1 public class Main {
2     public static void main(String[] args){
3         int[] array = getIntArray();
4         System.out.printf("%.10f¥n", avg(array));
5     }
6
7     public static double avg(int[] array){
8         int sum = 0;
9         for(int i=0; i<array.length; i++){
10             sum += array[i];
11         }
12         return sum/array.length;
13     }
14 }
```

getIntArray()は正の整数配列しか返さないことが保証されているらしい。

仕方ないからこの辺でsumをprintfしてみるか...

[出力] -94215685.0000000000

# デバッグ

デバッガとは

println()デバッグなんて時代遅れ！デバッガを使おう！！！！

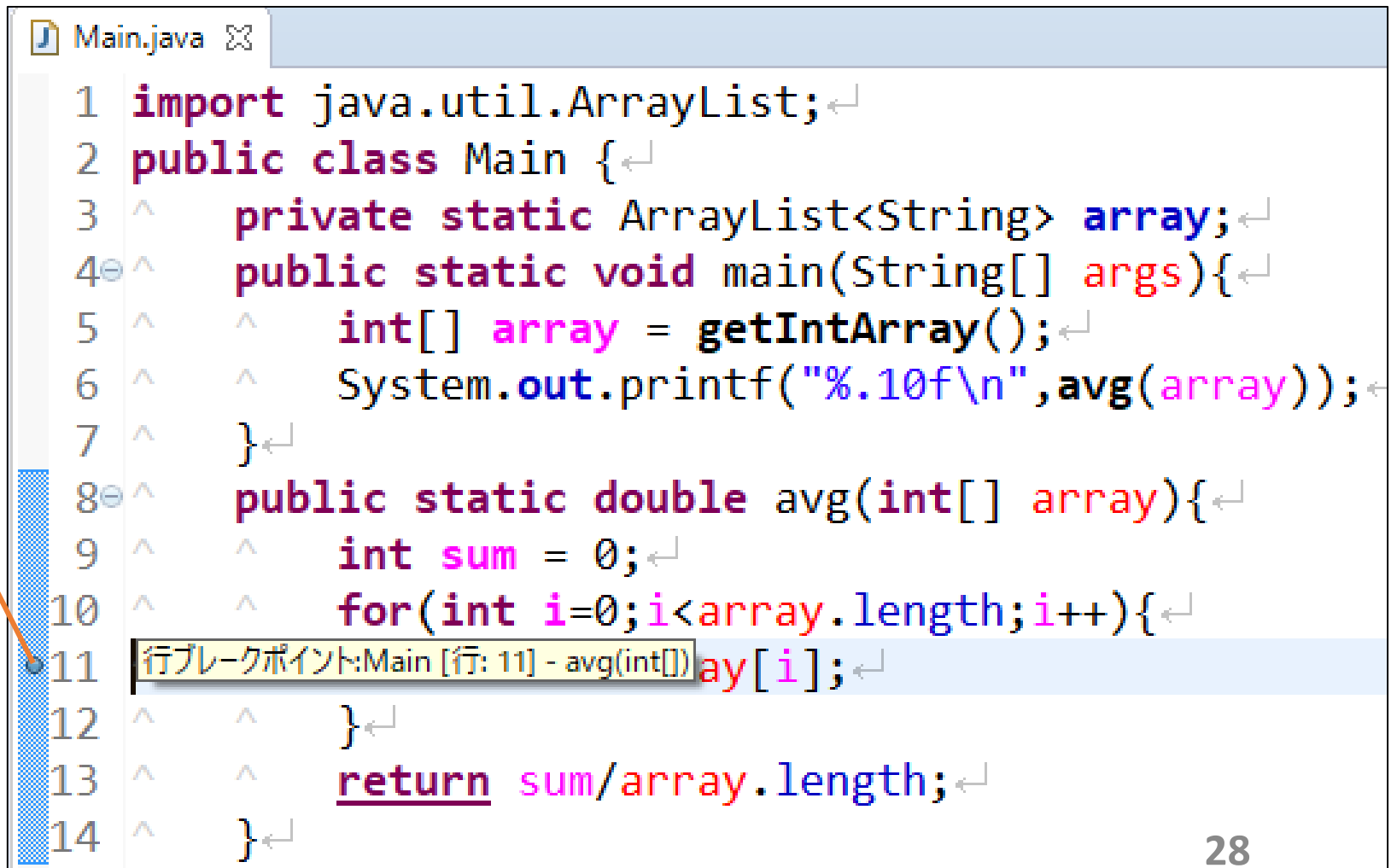
**デバッガ**：デバッグを支援するプログラム  
eclipseには標準でデバッガが搭載されている。

デバッガの使い方をこれから説明するが、VisualStudio等、他のIDE（統合開発環境）でも同じようにデバッガが搭載されている。若干使用感の違いはあるが、デバッガが使えることはデバッグ効率を大きく変える。

# デバッグ

デバッガとは

検査したい行をダブルクリックし  
ブレークポイントを設置する。



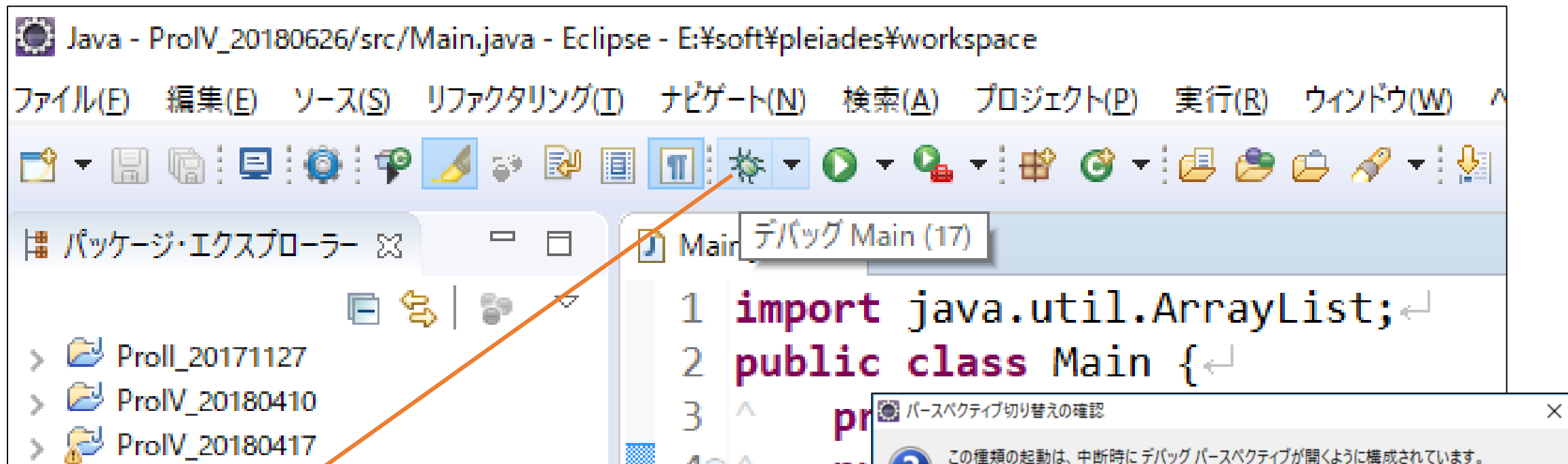
The screenshot shows a Java IDE window titled "Main.java". The code is as follows:

```
1 import java.util.ArrayList;
2 public class Main {
3     private static ArrayList<String> array;
4     public static void main(String[] args){
5         int[] array = getIntArray();
6         System.out.printf("%.10f\n", avg(array));
7     }
8     public static double avg(int[] array){
9         int sum = 0;
10        for(int i=0; i<array.length; i++){
11            行ブレークポイント: Main [行: 11] - avg(int[]) array[i];
12        }
13        return sum/array.length;
14    }
```

A blue vertical bar on the left side of the editor indicates the current line. A small circle on line 11 represents the breakpoint. A tooltip is visible over the breakpoint, displaying the text "行ブレークポイント: Main [行: 11] - avg(int[]) array[i];".

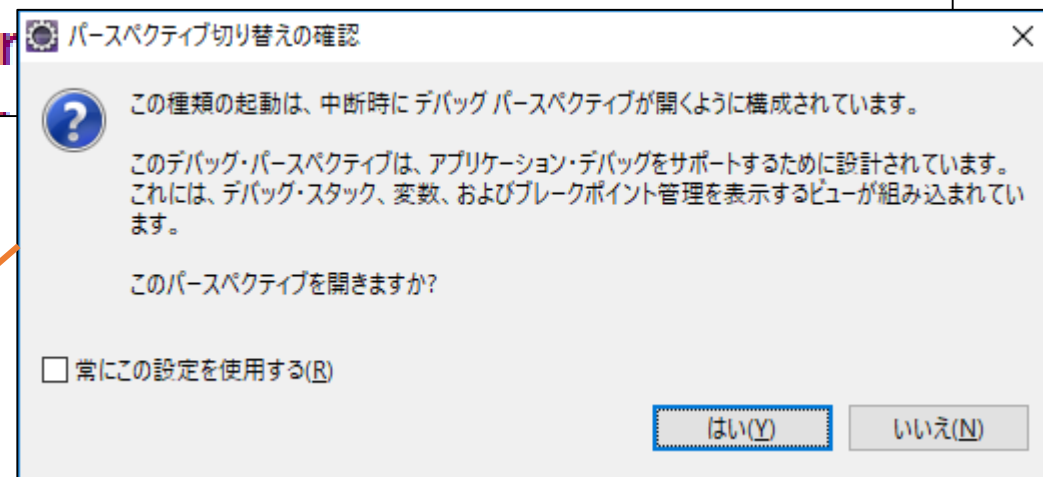
# デバッグ

デバッガとは



デバッグを実施する  
(普通の実行ボタンではない)

これが出たら. 「はい」 でよい



デバッグモードになる

現在のプログラム実行箇所

変数とその値

コンソール

デバッグ - ProIV\_20180626/src/Main.java - Eclipse - E:\soft\pleiades\workspace

ファイル(F) 編集(E) ソース(S) リファクタリング(R) ナビゲート(N) 検索(A) プロジェクト(P) 実行(R) ウィンドウ(W) ヘルプ(H)

クイック・アクセス | Java | デバッグ

デバッグ | サーバー

▼ Main (17) [Java アプリケーション]  
▼ ローカル・ホスト上のメイン:50372  
▼ フレームワーク (main) (中絶中 (メインの 11 行にブレークポイント))

(x)= 変数 | ブレークポイント

名前	値
> array	(id=16)
sum	0
i	0

Main.java

```
7 ^  
8 ^ public static double avg(int[] array){  
9 ^     int sum = 0;  
10 ^     for(int i=0; i<array.length; i++){  
11 ^         sum += array[i];  
12 ^     }  
13 ^     return sum/array.length;
```

コンソール | タスク

Main (17) [Java アプリケーション] E:\soft\pleiades\java\bin\javaw.exe (2018/06/20 22:18:26)

# デバッグ

デバッグとは

**ブレークポイント**：デバッグ時に、その地点までのプログラムを実行し、その地点で一度処理を中断する。

**デバッガの機能**：ブレークポイントで中断したときに、その地点における全ての変数とその値を確認することができる（神）。

**ステップ・イン (F5)**：次の行までプログラムの処理を進める（メソッドの場合、メソッドの中へカーソルを移動）

**ステップ・オーバー (F6)**：次の行までプログラムの処理を進める（メソッドの場合、呼び出し元の次の行へカーソルを移動）

**ステップ・リターン (F7)**：メソッドの中から戻る（メソッドの終了までカーソルを進める）。

それぞれ、メニューの「実行」の中にある。

# デバッグ

## デバッグの終了

1. デバッグの停止

2. Javaモードに戻す

さっきの例についてデバッガを用いてデバッグしてみる。



# デバッグ

## 練習問題 2 : デバッガを使ってみよう

WebからRenshu2.javaをDLし、**デバッガを用いてデバッグ**せよ。このプログラムは第9回の演習問題1の解答例である。なお、デバッグ手順がわからない場合は次を参照すると良い。

1. `initGlossary();`でちゃんとglossaryが作られてるか？

- ブレークポイントを`initGlossary()`直後に貼ってglossaryの中身を確認する。

2. `text`はちゃんと編集されてるか？

- 適切なところにブレークポイントを貼って、ステップ・オーバーしながら`text`の様子を確認する。

# 目次

## めずらしく目次

- デバッグ

- Keyword: デバッグ, デバッガ, ブレークポイント

- リファクタリング

- Keyword: リファクタリング

- 外部ライブラリ

- Keyword: ビルド・パス

- 高速化

- Keyword: 分岐をさせない, HashMap, StringBuilder

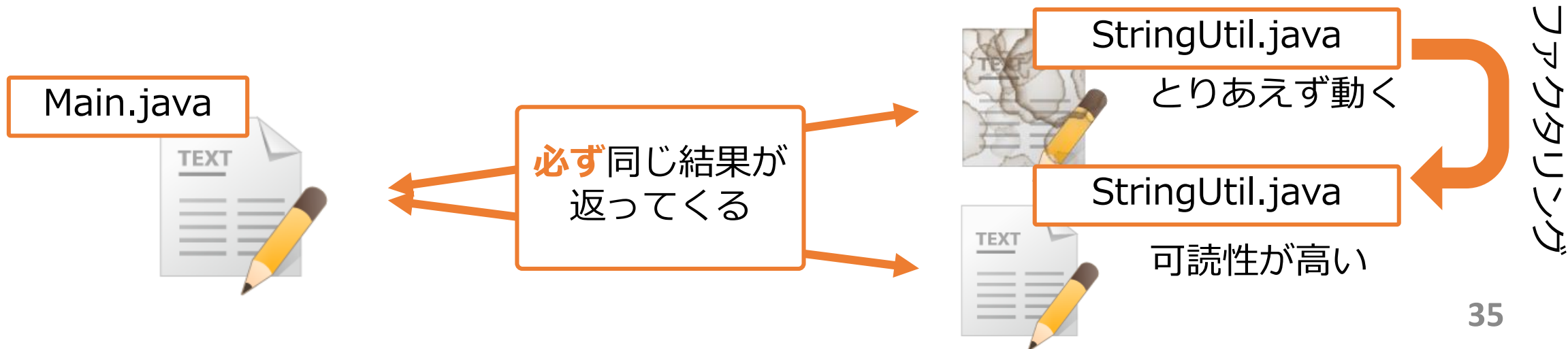
# リファクタリング

リファクタリングとは

**リファクタリング**：動作を変えずにコードを改善すること。

**目的**：なんとか動くコード → 保守・運用が容易なコード

- ・ 重複を削除して無駄のないコードに
- ・ 命名を改善して読みやすいコードに
- ・ 影響範囲を限定しバグが発見しやすいコードに



# リファクタリング

リファクタリングとは

**リファクタリング**：動作を変えずにコードを改善すること。

**手法：**

1. テストパターンを事前に準備しテストパターンがクリアできれば、動作が変わっていないと言える準備をする。
2. 動作が変わらないよう気を使いながら、コードを改善する。

**きれいなコードのために：**

- ・ 同じ処理はまとめる（繰り返さない）。
- ・ ネストは浅くする。
- ・ ネーミングをシンプルかつひと目で意図がわかるようにする。
- ・ 正確で簡潔なコメントを付与する。

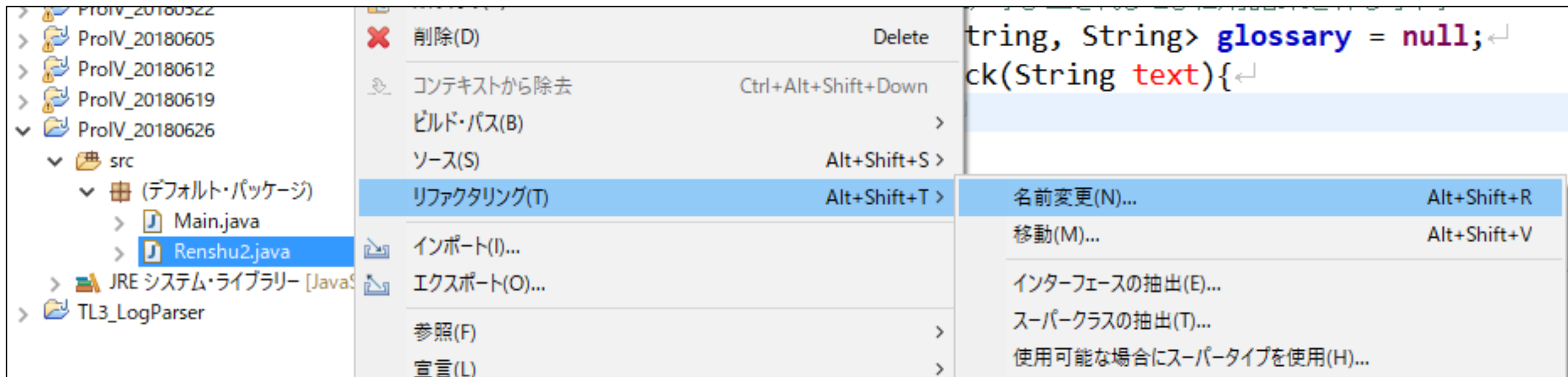
※ただし、コードを綺麗にする＝コスト(時間)がかかる点には注意

# リファクタリング

## eclipseにおけるリファクタリング

eclipseでは簡単かつ安全にリファクタリングの一部をサポートしてくれる機能が存在する.

- ・プロジェクトやファイル, 変数名等で右クリック
- ・リファクタリング→名前変更・移動



# リファクタリング

リファクタリングとは

**名前の変更**：安全にリネームする（クラス名，変数名，メソッド名）

- Javaでは基本的にファイル名＝クラス名なのでファイル名を変更すると各所に影響が出る.
- 影響範囲もろともまとめてリネームしてくれる便利機能である.

**移動**：安全に移動する（クラス，変数，メソッド）

- 移動することで呼び出し元で影響が出る.
- 呼び出し元での呼び出し方等も含めて移動してくれる便利機能である.
- クラスはパッケージを移動，変数やメソッドはクラスを移動.

**前で実践してみる.**

# リファクタリング

## 練習問題 3 : リファクタリングしてみる

練習問題 2 で使用したRenshu2.javaについて以下の通りリファクタリングを試してみよ.

1. メソッド名check()をproofreading()にリネームせよ.
  - main()から呼び出す側も変更されている点を目視で確認せよ.
2. 新規クラスUtil.javaを作成し, Renshu2.javaのglossaryフィールドをUtilに移動せよ.
  - Renshu2からglossaryへアクセスする際には「Util.」が接頭辞になっていることを確認せよ.
  - また, Utilに移動したglossaryの修飾子「private」が削除されていることを確認せよ.

# 目次

## めずらしく目次

- デバッグ

- Keyword: デバッグ, デバッガ, ブレークポイント

- リファクタリング

- Keyword: リファクタリング

- **外部ライブラリ**

- Keyword: ビルド・パス

- 高速化

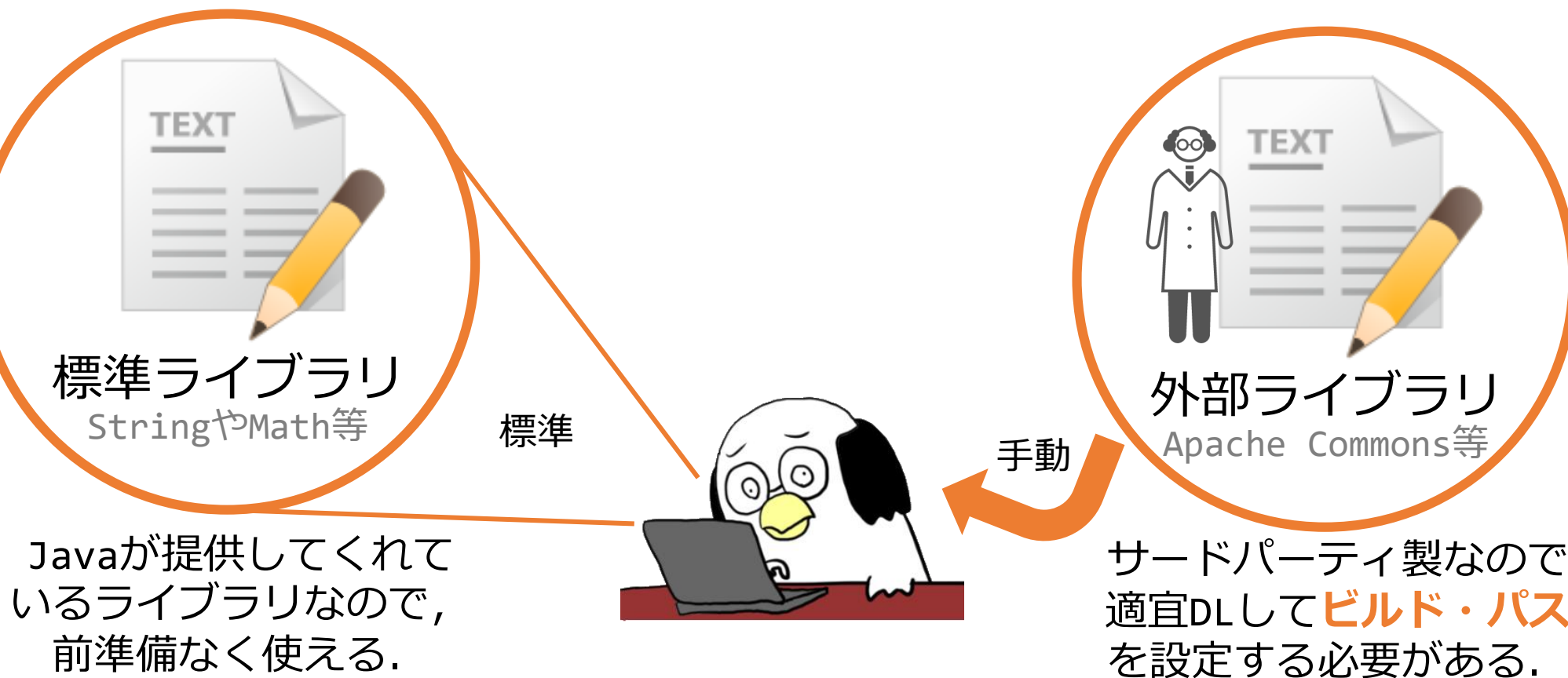
- Keyword: 分岐をさせない, HashMap, StringBuilder



# 外部ライブラリ

サードパーティな便利ライブラリを使う

Javaにはサードパーティ製ライブラリを提供する機能がある。



# 外部ライブラリ

サードパーティな便利ライブラリを使う

今回は、Apache Commons Lang (java.langの機能を拡張しているライブラリ) を使ってみた例を示す。

1. Apache Commonsのサイトから該当のjarファイルをDLする。  
<https://commons.apache.org/> ->Lang->Release Infomation->Download 3.7
2. 解凍するとcommons-lang3-3.7.jarがあるのでeclipseからビルド・パスを通す (次のページで説明) 。
3. これまで使えなかったクラスやメソッドが使用できるようになる。



.jarファイル

JARはJava ARchiveの略で、Javaに必要なクラスファイルをzip形式で圧縮したファイルである。これによりアプリやライブラリの配布が容易になる。

# 外部ライブラリ

サードパーティな便利ライブラリを使う

**前で実践してみる.**

将来的に、入った企業やプロジェクトによって様々なライブラリを使用することになるだろう。今は使いこなせなくてもよいが、このような機能があったなあ、程度に記憶しておくと、Java エンジニアになったときに間違いなく役に立つだろう。

# 目次

## めずらしく目次

- デバッグ
  - Keyword: デバッグ, デバッガ, ブレークポイント
- リファクタリング
  - Keyword: リファクタリング
- 外部ライブラリ
  - Keyword: ビルド・パス
- **高速化**
  - Keyword: 分岐をさせない, HashMap, StringBuilder

# 高速化

高速化とは

**高速化**：ロジックを変更することで処理を高速にすること

**必要となる場面**：

- ・ 組み込み等，計算資源が限られている場合
- ・ ゲーム等，リアルタイム処理が求められる場合
- ・ 機械学習等，膨大な計算量がかかる場合

**注意点**：

- ・ 現在はCPU・GPU性能が非常に高度化している。
  - ＞ 高速化しなくても良いシーンが多い。
- ・ 高速化と可読性はトレードオフな場合が多い。
  - ＞ 意味のない高速化のために可読性を下げると保守が辛い。
- ・ そもそもアルゴリズムを改善する方が速くなるケースが多い。

# 高速化

プログラムロジックを工夫して処理を高速に

高速化は非常に奥が深く、一朝一夕で理解できるものではない。  
＞最低限のことだけ伝えておく。

★★☆分岐は可能ならば計算か配列にする。

★☆☆検索する機会が多いならHashMapをうまく使う。

★★★多数の文字列結合は+ではなくStringBuilderを使う。

・同様にファイル書き込みはBufferedWriterを使う（前回参照）。

# 高速化

★★☆分岐は可能ならば計算か配列にする

以下のように「引数に応じて所定の値を返す処理」を考える.

ifバージョン

```
int fromif(int a){  
    if(a==0)return 10;  
    else if(a==1)return 20;  
    else if(a==2)return 30;  
    else if(a==3)return 40;  
    else if(a==4)return 50;  
    else return 60;  
}
```

switchバージョン

```
int fromswitch(int a){  
    switch(a){  
        case 0: return 10;  
        case 1: return 20;  
        case 2: return 30;  
        case 3: return 40;  
        case 4: return 50;  
        default: return 60;  
    }  
}
```

# 高速化

★★☆分岐は可能ならば計算か配列にする

分岐せずとも、計算や配列でもできるのでは？

計算バージョン

```
int fromcalc(int a){  
    return (10 + a*10);  
}
```

配列バージョン

```
int[] adder = {10,20,30,40,50,60};  
int fromarray(int a){  
    return adder[a];  
}
```

100,000,000回の試行(10セット)にかかった平均時間[ms]

if	switch	計算	配列
183.8(±6.2)	183.8(±8.1)	55.5(±10.0)	41.2(±9.8)



# 高速化

★★☆分岐は可能ならば計算か配列にする

引数が文字の場合は連想配列（HashMap）で応用が可能

ifバージョン

```
int fromif(String s){  
    if(s.equals("a"))return 10;  
    else if(s.equals("b"))return 20;  
    else if(s.equals("c"))return 30;  
    else if(s.equals("d"))return 40;  
    else if(s.equals("e"))return 50;  
    else return 60;  
}
```

switchバージョン

```
int fromswitch(String s){  
    switch(s){  
        case "a": return 10;  
        case "b": return 20;  
        case "c": return 30;  
        case "d": return 40;  
        case "e": return 50;  
        default: return 60;  
    }  
}
```

# 高速化

★★☆分岐は可能ならば計算か配列にする

引数が文字の場合は連想配列（HashMap）で応用が可能

計算バージョン（実装不可ケースが多い）

```
int fromcalc(String s){  
    return (s.charAt(0) - 'a' + 1) * 10;  
}
```

getOrDefault(s, 60)はget()の拡張で、sに該当する値がなければデフォルト値60を返す。

連想配列バージョン

```
HashMap<String, Integer> map;  
// mapは適切に初期化されているとする  
int fromarray(String s){  
    return map.getOrDefault(s, 60);  
}
```

100,000,000回の試行(10セット)にかかった平均時間[ms]

if	switch	計算	配列
861.5(±5.2)	909.2(±22.9)	88.2(±7.1)	199.9(±20.7)

# 高速化

☆☆☆検索する機会が多いならHashMapをうまく使う

Student型インスタンスを多数扱い、  
IDでStudentを検索することが多い  
ケースを考える。



```
Student
String ID
String name
int score;
```

ArrayListバージョン

```
// 一人ずつIDが一致するかどうか
// 調べる必要がある
Student fromList(String ID){
    for(Student s:stuList){
        if(s.ID.equals(ID))return s;
    }
    return null;
}
```

HashMapバージョン

```
// IDをキーにしてMapに登録しておけば一発
Student fromMap(String ID){
    return stuMap.getOrDefault(ID, null);
}
```

1,000回の試行(10セット平均)

ArrayList[ms]	HashMap[ms]
341.1(±16.6)	1.0(±2.1)

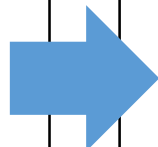
# 高速化

★★★多数の文字列結合は+ではなくStringBuilderを使う

1からnまでの数字をカンマ区切りで結合した文字列を生成するメソッドを考える。これまで+で文字列結合を行ってきたが長い文字列を生成する際には**StringBuilder**を使うほうが圧倒的に速い。

## +バージョン

```
String fromPlus(int n){  
    String res = "";  
    for(int i=1;i<=n;i++){  
        res += i + ",";  
    }  
    return res;  
}
```



## StringBuilderバージョン

```
static String fromBuilder(int n){  
    StringBuilder sb = new StringBuilder();  
    for(int i=1;i<=n;i++){  
        sb.append(i);  
        sb.append(",");  
    }  
    return sb.toString();  
}
```

# 高速化

★★★多数の文字列結合は+ではなくStringBuilderを使う

## StringBuilderの使い方

初期化 : `StringBuilder sb = new StringBuilder();`

文字列追加 : `sb.append( [ 追加する文字列 ] );`

最終文字列取得 : `sb.toString();`

50,000文字の結合(10セット)にかかった平均時間[ms]

＋バージョン	StringBuilder
2631.4(±117.7)	7.6(±5.1)

# 高速化

★★★多数の文字列結合は+ではなくStringBuilderを使う

## StringBuilderはなぜ速いのか

StringBuilderはバッファ領域に文字列をappendしていき最後にまとめてインスタンス化する。

一方+を使った場合、セミコロンで区切られた一行の処理毎にインスタンス化が行われるため、インスタンス化する分遅くなる。

したがって、右のような場合  
str1とstr2を生成する  
処理速度は変わらない。

```
String str1 = "a" + "i" + "u" + "e" + "o";
StringBuilder builder = new
StringBuilder();
builder.append("a");
builder.append("i");
builder.append("u");
builder.append("e");
builder.append("o");
String str2 = builder.toString();
```

# まとめ

- Javaの文法以外の面で、知っておいてほしい
  - デバッグ, リファクタリング, 外部ライブラリ, 高速化  
についての学習を行った.
- 本日の内容は慣れるまで中々使いこなせないかもしれないが、知識として知っておいてほしい.

# 次週予告

※次週以降も計算機室

## 前半

オブジェクト指向について復習する.

## 後半

オブジェクト指向の練習課題に取り組む.



# 本日の提出課題

## 講義パート

### 課題 1

本日の授業を聞いて、  
**よくわかった**と思う内容を  
2点簡潔に述べよ。

### 課題 2

本日の授業を聞いて、  
**質問事項**または**気になった点**  
を1点以上簡潔に述べよ。

### 課題 3

感想（あれば）

### 課題 4

なし

# 演習

- 昼休み, いつものWebページに演習問題をPDFで演習問題をアップロードする. 各自実施してプロII同様のWebページから提出すること.
- 質問は3人体制で受け付けるので遠慮なく申し出る. 質問の際は, どこまでわかっていて何がわからないのかを申し出ること.
- (ないとは思うが) コピペは発覚次第両成敗する.
  - ✓ {コピペ, カンニング} ∈不正行為
- つまらないミスも今回は問答無用で×とするので, 最終チェックを怠らないこと. (去年は目視で甘めに採点していたが, 自動採点を開発している意味がないので. . . )