

Java演習

第14回

2024/7/17

横山大作

講義前・後の質問

- 横山まで
- dyokoyama@meiji.ac.jp

提出課題12: 小田急(再掲)

- 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

- 今回の課題、難しかった人も多いようです。
 - ソートを「使う」こと（書くのではない）
 - 自分のデータの固まりを作って検索などを行うこと
 - 基本的なコレクションを使うこと

は、とにかくこれからの人生で頻出する考え方です。あやふやだったひとはぜひ復習してください。

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。

考え方

- まず、データ構造を考える
 - こんな感じの表でしたね

駅名	人	営業キロ
新宿	499919	0
南新宿	3782	0.8
参宮橋	15626	1.5
代々木八幡	20541	2.7
代々木上原	255378	3.5

- 縦に切り取ってString[], int[], double[]とすることもできます
 - でも、ソートが面倒じゃない？
- 横に切り取るのが便利そう
 - 駅の名前と情報を1つにまとめて処理する
 - オブジェクト指向の原点

データ構造の例

```
class Station {  
    String name;  
    int passengers;  
    double pos;  
    Station(String dat[]) {  
        name = dat[0];  
        passengers = Integer.parseInt(dat[1]);  
        pos = Double.parseDouble(dat[2]);  
    }  
}
```

コンストラクタは使いやすい引数で
作ればよい
今回は元データが文字列配列なので
そのまま使えるように設計した

もちろん

- フィールドは**private**にしておいた方が行儀はいい
 - `getter()`作ってアクセス
- この資料では割愛しておきます

要求と設計方針

- 駅の名前から駅情報を調べたい
 - 2つのデータの対応を覚える
 - Mapだ！
- どこかの駅からの距離で並べ替えてみたい
 - 順番が扱えるデータ構造じゃないとダメ
 - Listでよさそう！
- 両方で管理しておいたら便利そう
- Stationのインスタンスを作り、ListとMapの両方に登録する

データをどこに取っておくか

- 何回も駅リストを使うので、一度読みこんだデータを使いまわしたい
- **OdakyuSort**のインスタンス変数にするか、**static**変数にするかが良さそう
 - メソッド内の自動変数だと何度も読み込みしないといけない

データを格納する例

```
public class OdakyuSort {  
    List<Station> sts;  
    Map<String, Station> st_names;  
    static OdakyuData od = new OdakyuData();  
    OdakyuSort() {  
        sts = new ArrayList<Station>();  
        st_names = new HashMap<String, Station>();  
        for (int i = 0; i < od.numStations(); i++) {  
            String[] stationstr = od.station(i);  
            Station st = new Station(stationstr);  
            sts.add(st);  
            st_names.put(st.name, st);  
        }  
    }  
}
```

- データはインスタンス変数にしてみた
 - コンストラクタで準備してしまう

さて、大きな構造

- 名前から駅の座標を取り出す
 - Mapで準備しているので一発
- ある座標からの位置に近い順に表示
 - ソートして出す
- ソートをうまく「使う」ようにしましょう
 - 正直、ソートはもはやどの言語にも備わった機能という感覚
 - 活用しよう

駅の名前から座標を取り出す

- **Map**の対応を使って、該当する駅の**pos**を見つければ一撃
 - もちろん、エラー処理とかできるとより良いですね

```
double getpos(String name) {  
    Station st = st_names.get(name);  
    return st.pos;  
}
```

距離でのソートの例(無名クラス)

```
void posOrder(double o) {  
    Collections.sort(stations, new Comparator<Station>() {  
        public int compare(Station l, Station r) {  
            double lp = Math.abs(l.pos - o);  
            double rp = Math.abs(r.pos - o);  
            if (lp < rp) { return -1; }  
            else if (lp > rp) { return 1; }  
            else return 0;  
        }  
    });  
    for (int i = 1; i < 6; i++) {  
        System.out.println(stations.get(i));  
    }  
}
```

普通のクラスでComparator

```
void posOrder(double o) {  
    class Cmp implements Comparator<Station> {  
        double cmp_o;  
        Cmp(double arg_o) { cmp_o = arg_o; }  
        public int compare(Station l, Station r) {  
            double lp = Math.abs(l.pos - cmp_o);  
            double rp = Math.abs(r.pos - cmp_o);  
            if (lp < rp) { return -1; }  
            else if (lp > rp) { return 1; }  
            else return 0;  
        }  
    }  
    Cmp c = new Cmp(o);  
    Collections.sort(stations, c);  
}
```

Comparatorの
インスタンス変数に
原点情報を入れておく

呼ぶ側では

```
void printNeighbor(String stname) {  
    double p = getpos(stname);  
    posOrder(p);  
}
```

- 例えばこんな感じで組み合わせるとか。

Comparator.comparing使うと

```
public class OdakyuSort {  
    void posOrder(double o) {  
        Collections.sort(stations,  
            Comparator.comparing(x -> x.distance(o)));  
        for (int i = 1; i < 6; i++) {  
            System.out.println(stations.get(i));  
        }  
    }  
}
```

- どこかからのdistance()を計算するメソッドをソートのキーにする
- ずいぶん簡単になる

ちなみに

- こう書いたらバグが入りました

```
void posOrder(double o) {  
    Collections.sort(stations,  
        (x, y) ->  
            (int)(Math.abs(x.pos - o) * 10  
                - Math.abs(y.pos - o) * 10));  
    for (int i = 1; i < 6; i++) {  
        System.out.println(stations.get(i));  
    }  
}
```

何故か？

- 登戸、百合ヶ丘問題
- 距離としてはそれぞれ2.7と2.6だが...
 - だから10倍すればintにキャストして問題ないかな？と思った
- Javaのdoubleで計算するとそれぞれ
2.69999999999999993
2.60000000000000014
になる
- 浮動小数点数の表現に起因

登戸	162422	15.2	2.7
向ヶ丘遊園	66684	15.8	2.1
生田	45735	17.9	
読売ランド 前	35412	19.2	1.3
百合ヶ丘	21293	20.5	2.6

そのほか細かいこと

- 絶対値は`Math.abs()`というメソッドが用意されている
- **「知らない」のは問題ない、「この手のメソッドは絶対あるはず」と思わなかったことが問題**
 - ルートはあるはず。三角関数はあるはず。
 - ...絶対値くらいはあるはず！
- 調べようと思う感覚を養っておこう
 - プログラミングに限った話ではない。

データの管理

- 基準駅を変えて`printNeighbor()`呼ぶとリストの並び方が変わってしまう
 - 気持ちが悪い、と思う人はその感覚を大事にしましょう。その通りだと思います。
- 気持ちが悪い場合は、例えば`printNeighbor`で`Station`リストをコピーしてからソートすると良いでしょう
 - リストに入っているのはオブジェクトへの参照なので、必要なメモリは参照*駅数 分になる

よくあったミスや疑問な実装

- **Station**の**pos**を、指定された駅からの距離に書き換えてからソートする
 - そして、指定された駅の**pos**も書き換えてしまうので書き換えが途中からおかしくなる人も
 - データには「意味」があり、**pos**は原点からの「座標」である、という意味を守り続けることをお勧めします
 - たまたま型が**double**で同じだから使いまわす、というような態度はバグの温床です

よくあったミスや疑問な実装

- データを{駅名 -> [乗降客数、座標]}というMapで管理しようとする
 - Map<String, List<Object> >とか
 - Map<String, List<String> >とか
 - Key = “新宿”, Value = [“521160”, “0.0”] という感じ
 - 型がないスクリプト言語だと気持ちはわからないでもない
 - Pythonとか
 - その場限りにさっと書いてパッと使う感じ
 - Javaではこういう書き方はあまり向かない
 - リストやタプルをそんなに気軽につかえるようにしてない
 - 普通にクラス作ったほうが言語のサポート得られる
 - Pythonでもリストではなくタプルにしたい（気になる人はググって）

よくあったミスや疑問な実装

- `List<Object>`みたいな指定
 - せっかく型情報を付けられるのだから、`List<Station>`みたいにしよう
- {駅名->座標}, {座標->ある駅からの距離}, {座標->駅名}, みたいにたくさんの**Map**や**List**を作って、順番に求めていく
 - 目の前のことしか見えてない感じ。もう少し問題全体を見て、どんなデータだったら何ができるかを考えてから書こう
 - 同じ距離になる駅が複数あって破綻している人もいますね。**Map**のキーに何を使うかをよく考えましょう

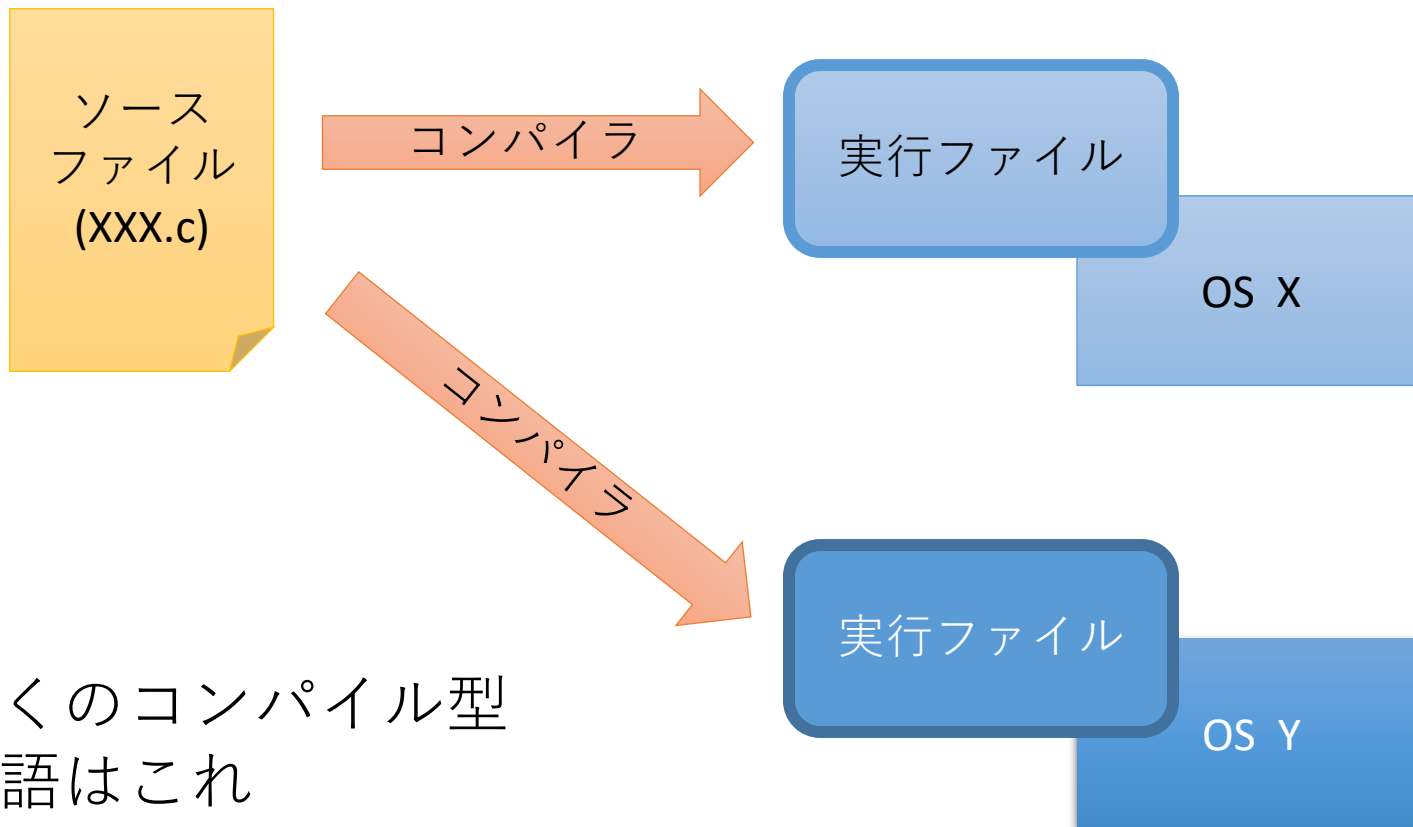
この資料の内容

- 細かいトピックを落ち穂拾い
 - Java VM
 - 様々な用途
 - ストリーム処理
 - スレッド

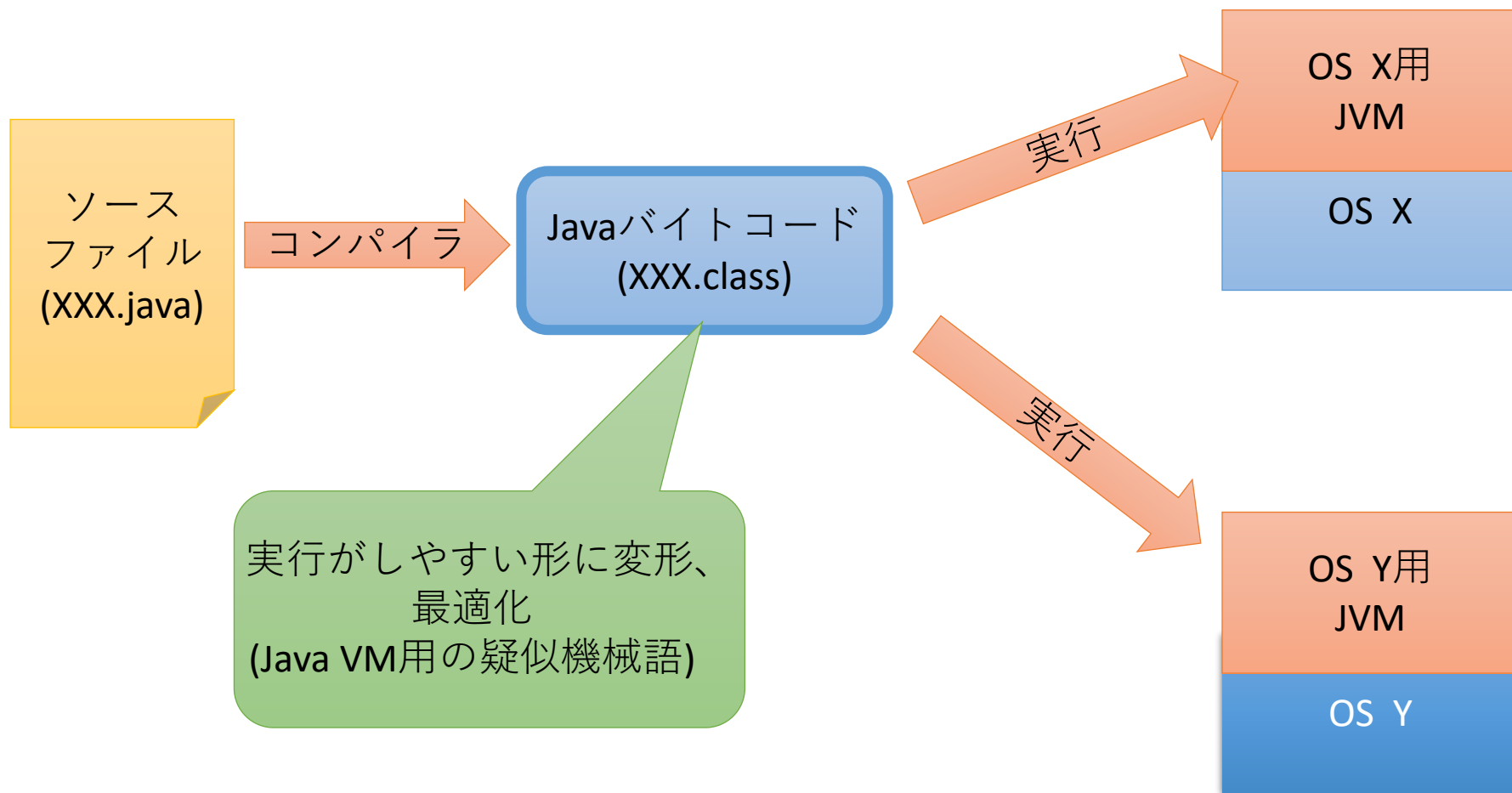
Java仮想マシンについて(p. 34)

- コンパイルの時に何が起きているか
 - 実行の時に何が起きているか
- それぞれおさらい

C言語では



Javaでは



コマンドラインだと違いは明快

- javac : コンパイル
- java : JVM上で実行
- Eclipse上だと気付かないかも

Java VM

- Javaのバイトコードを実行するための仮想マシン
- 高速化のために様々な最適化がなされる
 - JIT: Just In Time Compiler
 - 動いている最中にコンパイルしてだんだん速くなっていく
- JVMは出来が良いので、JVM用のバイトコードを吐く別の言語も存在 (p.668)
 - Scala, Kotlin, ...
- バイトコードをいじるのは研究にも適している

Javaの様々な用途

- 18章: まだまだ広がるJavaの世界
- 様々な用途に使われている
 - クライアントアプリケーションとして
 - GUI
 - ケータイアプリ
 - サーバサイドで
 - Webバックエンドで
 - 計算部分
 - ストレージ部分
 - JVMを活かして別言語に見えていることも多い
 - AndroidのケータイアプリのKotlin
 - データ処理のScala とか

アプリケーションの配布の仕方

- どこでも動く（のが建前）
- Javaソースを配布しても良いが...
- classファイルを配布するだけで良い
 - JAR形式とか

パッケージとクラスパス (p. 221)

- パッケージはJavaのクラスファイルの整理法
 - 第6回講義も参照のこと
- `package` で指定
 - 階層的に整理することができる
 - `foo.bar`パッケージのように.でつないで表現
- ソースファイルのディレクトリ構造がパッケージ構造と一致している必要あり
- 実行時にもディレクトリ構造と一致する必要がある
 - クラスファイルを探すときにパッケージのディレクトリ構造を使う
 - クラスファイルを探す場所を指定するのがクラスパス

JAR (p. 220,234)

- Javaのアプリケーションを配布するための形式
- クラスやリソース(画像、データなど)をまとめて配布し、簡単に実行できる
 - 設定できていればダブルクリックで起動
 - クラスの固まりとしてクラスパスに指定もできる
- 実は単なるzipファイル
 - 決まった名前(META-INF/MANIFEST.MF)のファイルに「起動すべきmainメソッドのあるクラス」が書いてある

ストリーム (p.658)

- ファイルや標準入出力の取り扱い
- 注意: Java8からStreamと呼ばれる別の機能ができた
 - Collectionのようなデータの固まりに対する処理をわかりやすく書くための機能
 - ググったり本を読んだりするときに誤解しないように気を付けて

入出力ストリーム

- Javaのプログラムにデータを出し入れする方法の割り切り方
 - UNIXなど、ほぼすべてのOSがこのような入出力を基本にしている
 - 参考：コンピュータにとって、入出力はたいてい次の2通り
 - ストリーム（順番に流れてくる）
 - ランダムアクセス（メモリのようなイメージ、アドレスを指定して読み書き）

プロセスと入出力

- プロセス: 隔離された空間
 - 風船の膜のようなもの
 - 内側は独立しているので、外側の影響は受けない
 - 内側で計算しても、外側には何も影響は出てこない
 - (データの) 出入りがあるのは入出力
- 入出力: 風船に刺さったストローみたいなもの
 - そこを通してデータが出入りする
- 何でもかんでも「入出力 (ファイル)」だ
と思う
 - ディスク上のデータ、画面出力、キーボード、ネットワーク...
 - 切り替えて使えるように抽象化している



標準入出力

- 色々なOSで基本となる考え方
- 1つの計算単位=プロセス
- プロセスにはもともと
 - 1本の入力（標準入力）
 - 2本の出力（標準出力、標準エラー出力） がある
 - 3本のストローが刺さっている
- 標準入力は普段はキーボードにつながってる
- 標準出力、エラーは画面につながってる
- 標準入出力を繋ぎ変えて連携させるのが（特にUNIX系の）思想

標準入出力・標準エラー出力

- 標準入出力の取り扱い
 - `System.in`: 標準入力
 - `System.out`: 標準出力
 - `System.err`: 標準エラー出力すべてStreamと呼ばれるもの
 - ストローで説明したような、データの流れるパイプ
- `System.out.println()` は皆様ご存知
- `System.err.println()` も知っておこう
 - ログからエラーだけ取り出したりするのが容易になる

ファイルでの実例で見てみよう (p. 660)

```
import java.io.*;
class WriterSample {
    public static void main(String[] args) throws IOException {
        Writer out = new FileWriter("hoge.txt");

        out.write("foo");
        out.write("bar");

        out.close();
    }
}
```

バッファリング

- 入出力は重い処理
- 1文字ずつ入出力していると負荷がかなり高い
- バッファリングすると負荷が減る
- バッファリング: 複数文字をまとめて入出力し、結果をメモリのどこかに置いておく
- 生のストリームに対し、「デコレート」してバッファリングの機能を追加して使う
 - ストリームをくるんでより高機能なストリームにしていく
 - `BufferedInputStream`など

closeが必要

- 教科書はcatchを省略していますが、「途中まで書けた」などの処理がしたければ、ちゃんとIOExceptionのcatchもしましょう
- 必ずclose()を最終的に呼ばなくてはならない、
というのは結構大変

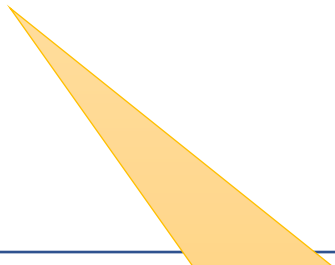
9回講義より再掲： エラー処理は変態になりがち

```
try {
    InputStream 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
        }
    }
}
```

9回講義より再掲： try-with-resources文

- 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！

Files クラス

- ファイルそのものの操作に関するクラス
 - ファイルの中身ではなく、ファイル名とか、ファイルの場所とかに関する情報取得、操作

ちなみに...再帰処理

- ファイル操作では再帰処理で書きたくなる場所が多数あります
 - このディレクトリ以下のすべてのファイル进行处理する、とか必要になりがち
- 再帰処理のイメージがイマイチつかめない人は、ファイル操作を考えてみるとわかる可能性も

並行性の利用

- GUIなどで利用したくなる
 - 理解が必要
- Javaは最初から並行性を扱えるように設計されていた

スレッド

- 並行処理を行う仕組み
 - 複数の処理が「見た目上」同時に走る
- 例:
 - **GUI:** 何か裏で処理をしている間、表ではプログレスバーが進む
 - **サーバ:** 複数人が同時にアプリケーションにアクセスできる
- **1つの処理の流れのことをスレッドと呼ぶ**（一般名称）
- **複数のスレッドが同時に動くことをマルチスレッドと呼ぶ**（一般名称）

並行と並列

- 並行: 論理的に同時に実行されるもの
 - 実際には**1**つの**CPU**で、時分割で実行されているかもしれない
- 並列: 物理的に同時に実行されるもの
 - 本当に複数の**CPU**、複数の筐体などが存在して、同時に複数の計算が走っている状態

スレッドとThread

- 1つの処理の流れ（？）
- 何か「プログラムを読んで実行するもの」を想像してみてください
- それにオブジェクトとして実体を与えたものがThreadです
 - 関数オブジェクトと同じ。データではなく「実行するもの」に実体が生まれている
- 今までのプログラムは、暗黙のうちに1つだけThreadオブジェクトが作られて動作していました
 - エラーが起きると in thread “main” とか言われてた

スレッドの作成方法

- Threadクラスを継承
- runメソッドを作る
- startメソッドを呼ぶ
- Runnableインターフェイスを実装しても良い
 - やることはrunメソッドの実装
 - Threadを作る時にインスタンスを渡してあとは同じ
 - Runnableインタフェースは関数型インタフェースなのでラムダ式で書ける。ラムダ式をThreadのコンストラクタに渡せる。

ラムダ式

```
public class Threadtest {  
    public static void main(String[] args) {  
        Thread t = new Thread(  
            () -> {  
                for (int i = 0; i < 5; i++) {  
                    System.out.println("Threadの実行中" + i);  
                }  
            }  
        );  
        t.start();  
  
        for (int i = 0; i < 5; i++) {  
            System.out.println("mainの実行中" + i);  
        }  
    }  
}
```

スレッドの同期

- 並行性がある場合には同期が必要
 - 並列のときに限らない！
 - 実行が途中で止まったりするから
- 何かオブジェクトに目印をつけて、そのオブジェクトを握った状態でスレッド実行する
 - ロック と呼ばれる機構
- Javaではsynchronizedというキーワードで制御

マルチスレッドプログラムは 難しい

- 適切に同期をとらないといけない
- 同期をミスるとバグ
- もしくはデッドロック
- バグが取りにくい
 - 再現しない、時々しかバグが発生しない

スレッドをなぜ使うか

- 目的を理解して使うことが大事
- 処理速度を速くする
 - 本当に速くするのは大変。データ構造に立ち戻って熟考を。
 - 単純な問題（Embarrassingly Parallel）では、プロセス並列の方が簡単な場合もある
 - 高レベルなフレームワーク（後述）を使う方が良いことが多い
- 複数の処理を見かけ上並行に実行する必要がある
 - この場合はむしろスレッドを使った方がスッキリ書ける
 - 恐れずに使ってみるのが良い
 - 高レベルフレームワークも重要

Modern Javaでの並行性の取り扱い

- Threadは一番単純な部品
 - 何でも書けるが、扱いが面倒
- 安全に・便利に利用できる様々な機能が追加されている
 - Executer
 - Fork/Join
 - ParallelStream
 - Future

Javaを学んで

- 皆さんはすごいパワーを得ました
 - オブジェクト指向で問題を解ける
 - 定番のコレクションを使える
 - これらは**40年**は持つ能力
 - 例えコンピュータがなくても使えるはず
 - あとはおまけ
 - **Java**が使えるようになった
- これからの人生に役立ててください