Lambda お稽古 問題編

Lambda お稽古 問題編

1. はじめに

Java SE 8 の一番の注目は、なんといっても Project Lambda です。

Project Lambda は Java に関数型言語のアイデアを導入することで、さまざまな処理を簡潔に書けるようになったり、パラレル処理を行うことができるようになりました。

Project Lambda で提供されているのは主に以下の 2 つの点です。

- Lambda 式
- Stream API

Lambda 式は文法の変更で、Stream API はライブラリの追加です。

Lambda 式は関数型言語の関数をオブジェクト的に表す記法です。一方の Stream API はコレクションなどのループを今までの拡張 for 文から、内部イテレータという方法で記述するための API です。

どちらも、今までの Java の書き方、考え方とはちょっと違います。でも、恐れることはありません。

こういうのは習うより慣れろです。繰り返し書くことにより Lambda 式や Stream API の書き方も身についていくはずです。

本ハンズオンでは、プログラムの断片を提示していきます。それを Lambda 式や Stream で書き換えていってください。PC はなくても大丈夫。鉛筆で直接書き換えていってください。

もし、余裕があるのであれば、家に帰ってから、ぜひ PC で実行し直してみてください。使用した問題、解答は GitHub の LambdaOkeiko プロジェクトに置いておきます。

LambdaOkeiko プロジェクトのソースでは、ほとんどの問題が実行可能になっています。Lambda 式と Stram API に書きかえた後に、実行して結果が同一になるかどうか確かめてみましょう。

LambdaOkeiko https://github.com/skrb/LambdaOkeiko (https://github.com/skrb/LambdaOkeiko)

2. Lambda 式への書きかえ

Lambda 式は、実装すべきメソッドが 1 つのインタフェースを実装した匿名クラスを置き換えるものです。

実装すべきメソッドが 1 つのインタフェースを関数型インタフェースと呼びます。たとえば、java.lang.Runnable インタフェースや java.util.concurrent.Callable インタフェース、java.util.Comparator インタフェースなどが関数型インタフェースです。

Runnable インタフェースは run メソッド、Callable インタフェースは call メソッド、Comparator インタフェースは compare メソッドを実装します。

これ以外にも Java SE 8 から追加された java.util.function パッケージでいろいろな関数型インタフェースが提供されています。

なぜ、実装すべきメソッドが 1 つのインタフェースを関数型インタフェースと呼ぶのでしょう。たとえば、ある処理を別のスレッドで実行したい場合、ExecutorService インタフェースを使って次のように記述することがあります。

```
        // スレッドプールを生成
        ExecutorService service = Executors.newCachedThreadPool();

        // 別スレッドで行いたい処理を表すためのRunnableオブジェクト
        Runnable task = new Runnable() {
            @Override
            public void run() {
                // 別スレッドで行いたい処理
            }
        };

        // 別スレッドで処理を行う
        service.submit(task);
```

ExecutorService オブジェクトである service 変数はスレッドプールを表しています。そこに対して、別スレッドで行いたい処理を submit メソッドで指定します。

本来であれば、run メソッドの内部に書いてある処理だけを submit メソッドに渡せばいいのですが、Java ではそれはできません。そこで、Runnable オブジェクトで行いたい処理をくるんで、それを submit メソッドに渡していきます。

処理だけを表す書き方に関数があります。Java でいうところのメソッドのようなものです。関数型言語では、この関数を直接メソッドの引数に渡すことができます。Java では関数は渡せませんが、Runnable オブジェクトであれば渡せます。ということは、Runnable インタフェースはあたかも関数のように扱えるということです。

しかし、複数のメソッドを実装しなくてはいけないインタフェースだと、どのメソッドを行いたい処理なのか指定する必要があり、単純に関数の代わりに使うことはできません。

そこで、実装すべきメソッドが 1 つのインタフェースを関数型インタフェースとよぶことにしたのです。

そして、この関数型インタフェースを実装した匿名クラスの代わりになるのが、Lambda 式です。上のコードでも、本当に submit メソッドに渡したいのは run メソッドの内部だけなのに、匿名クラスでは定義する部分など冗長な部分が増えてしまいます。そこで、Lambda 式は匿名クラスの定義部分などは取っ払って、メソッド (もしくは関数) の部分だけ記述すればいいようにしたのです。

Lambda 式は (引数) -> { メソッド本体 } の形式で表します。引数は関数型インタフェースの実装すべきメソッドの引数を表し、処理はそのメソッドの実体を表しています。たとえば、上記の Runnable インタフェースの匿名クラスは次のように記述できます。

```
// 別スレッドで行いたい処理を表すためのLambda式
Runnable task = () -> {
        // 別スレッドで行いたい処理
};
```

ずいぶん、簡単になりました。Runnable インタフェースは引数がないので、丸カッコ () だけで、その後にメソッド本体を記述します。

では、皆さんの番です。以下の関数型インタフェースを実装した匿名クラスを Lambda 式を書き換えてみてましょう。

2-1. Lambda 式で書き換えてみましょう

Comparator インタフェースは 2 つの引数の比較をするためのインタフェースです。ここでは、整数を表す Integer クラスを引数にとる Comparator インタフェースの実装匿名クラスを Lambda 式で書きかえてみましょう。

```
Comparator<Integer> comparator1 = new Comparator<Integer>() {
    @Override
    public int compare(Integer x, Integer y) {
        return Integer.compare(x, y);
    }
};
```

2-2. Lambda 式で書き換えてみましょう

Callable インタフェースは、別スレッドで処理を行うためのインタフェースです。時間がかかる処理を別スレッドで行う場合などに使用します。たとえば、別スレッドでファイルを読み込む処理を Lambda 式で書いてみましょう。

ここで使用している Path インタフェースと Files クラスは Java SE 7 から使えるようになった、ファイルを扱うためのクラスです。

```
Callable<List<String>> task = new Callable<List<String>>() {
    @Override
    public List<String> call() throws Exception {
        Path path = Paths.get("README.md");
        return Files.readAllLines(path);
    }
};
```

2-3. Lambda 式で書き換えてみましょう

2-2 は Callable インタフェースを使用してファイルの内容を戻り値として返していました。Runnable インタフェースは戻り値がないので、別スレッドで表示まで行ってみました。これもラムダ式で記述できます。

```
        Runnable task = new Runnable() {
            @Override
            public void run() {
                try {
                    Path path = Paths.get("README.md");
                    List<String> contents =  Files.readAllLines(path);
                    System.out.println(contents);
                } catch (IOException ex) {
                    System.err.println("Fail Read.");
                }
            }
        };
```

2-4. Lambda 式で書き換えてみましょう

Function インタフェースは Stream API と一緒に導入された関数型インタフェースで、引数が 1 つ、戻り値がある処理を表すために使用します。ここでは、文字列を数値に変換する処理を表しています。

これも Lambda 式で記述してみましょう。

```
        Function<String, Integer> function = new Function<String, Integer>() {
            @Override
            public Integer apply(String x) {
                return Integer.valueOf(x);
            }
        };
```

2-5. Lambda 式で処理を汎用化してみましょう

次の filterList メソッドは、数値のリストから 10 以上の数値のリストを作るためのメソッドです。

```
    private List<Integer> filterList(List<Integer> src) {
        List<Integer> dest = new ArrayList<>();

        for (Integer x: src) {
            if (x > 10) {
                dest.add(x);
            }
        }

        return dest;
    }
```

この書き方だと、if 文の条件が固定されてしまいます。ここでは、10 以上かもしれませんが、あるときは 10 以下の数値リストを抽出する必要があるかもしれません。しかし、この書き方では条件に応じた同じようなメソッドをまた書かなくてはなりません。

そこで、条件の部分だけ抽出して、条件部分を変更可能にすることができます。ここでは条件を記述するのに、Predicate インタフェースを使用します。

```
    private List<Integer> filterList(List<Integer> src, Predicate<Integer> predicate) {
        List<Integer> dest = new ArrayList<>();

        for (Integer x : src) {
            if (predicate.test(x)) {
                dest.add(x);
            }
        }

        return dest;
    }
```

このように処理を変更可能にしているメソッドには、java.util.Collections クラスの sort メソッドがあります。sort メソッドはリストをソートさせるために使用しますが、ソートの順番を決めるために第 2 引数の型が Comparator インタフェースになっています。昇順で並べる Comparator オブジェクトを引数に使用すれば、sort メソッドの結果としてえられるリストも昇順になります。一方で逆順で並べる Comparator オブジェクトを引数に使用すれば、結果も逆順のリストになります。

さて、filterList メソッドを使ってみましょう。ここでは、数値のリストである numbers 変数があったとします。このリストから 10 以上のものだけをフィルタリングしたい場合、次のように記述します。

```
        List<Integer> numbers = ...;

        List<Integer> bigNumbers
                = filterList(numbers,
                         new Predicate<Integer>() {
                             @Override
                             public boolean test(Integer t) {
                                return t > 10;
                             }
                         }
                );
```

匿名クラスだと冗長ですが、Lmabda 式にすることで簡潔になります。そこで、これも Lambda 式で書いてみましょう。

3. for 文の書きかえ (forEach)

Lambda 文を書くことに慣れたら、for 文もしくは拡張 for 文を Stream API で書き直してみましょう。

ストリームは様々な処理を行うことができるイテレータです。Iterable インタフェースのように、自身ではオブジェクトを保持せずに、元となるコレクションなどの要素をイテレートしていきます。

ストリームにはオブジェクトを対象とした Stream インタフェース、プリミティブ型を対象にした IntStream インタフェース、LongStream インタフェース、DoubleStream インタフェースの 4 種類があります。

基本的なメソッドは同じですが、sum メソッド、max メソッドなどプリミティブ型が対象のストリームにしかないメソッドもあります。

ストリームの生成には複数の方法があります。コレクションから生成するには、Collection#stream メソッドを使用します。また、IntStream オブジェクトを生成するには、前述した range メソッドを使用するのが手軽です。

Stream API のメソッドには中間処理と終端処理に分類されます。そして、Stream API は複数の中間処理を連ね、最後に終端処理を行うスタイルでコードを記述していきます。これを Stream パイプラインと呼びます。

中間操作の戻り値は必ずストリームになります。終端操作の戻り値がストリームパイプラインの処理結果になるわけです。

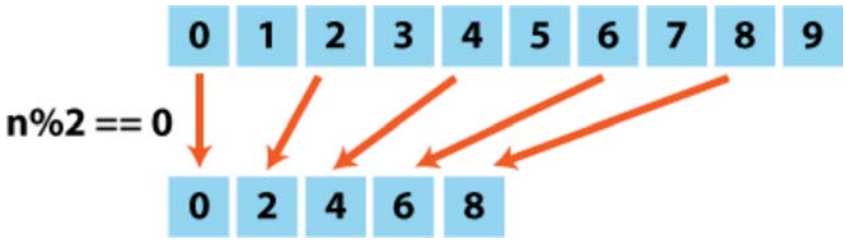
さて、この Lambda お稽古では以下の 5 種類のメソッドを使用します。

- 中間操作
  - filter 条件に合致した要素だけをフィルタリングする
  - map 要素を、他の値に変換する
  - flatMap 要素をストリームに変換し、それを連結して 1 つのストリームを生成する
- 終端操作
  - forEach 各要素に対して、何らかの処理を行う
  - collect 要素の集約処理を行う

forEach メソッドは for 文/拡張 for 文からの置き換えが容易なので、まず forEach メソッドを使用し、次章で collect メソッドを扱います。

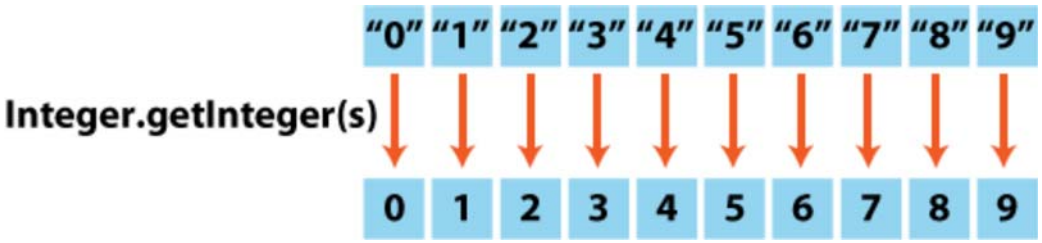
中間操作のfilter メソッドと map メソッドは中間操作なので、戻り値はストリームになります。

filter メソッドは条件に合致した要素だけをフィルタリングし、新たなストリームを生成するメソッドです。



filter メソッド

一方の map メソッドは各要素に対し、何らかの処理を行い、新たな値を作成することで、新たなストリームを作成します。以下の図では文字列のストリームの個々の要素を、Integer オブジェクトに変換しています。最終的に Stream<Integer> オブジェクトを生成します。



map メソッド

mapToInt メソッドなどの mapToX メソッド群は、map メソッドの一種で、Stream オブジェクトからプリミティブ型に対応した InstStream オブジェクトなど、もしくはプリミティブ型に対応したストリームから Stream オブジェクトへの変換を行うメソッドです。

また、flatMap メソッドも map メソッドの一種ですが、個々の要素をストリームに変換します。通常の map メソッドだと、個々の要素をストリームにすると、ストリームの要素にストリームという入れ子のストリームになります。これに対し、flatMap メソッドでは個々の要素を変換してできたストリームを展開し、最終的に 1 つのストリームに変換するところが map メソッドと異なります。

では、まず forEach メソッドで書き換えを行ってみましょう。

3-1. forEach メソッドで書き換えてみましょう

次の printList メソッドは文字列のリストを引数にとって、そのリストの要素を標準出力に出力するメソッドです。

これを Stream API で書きかえてみましょう。

```
    private void printList(List<String> texts) {
        for (String text: texts) {
            System.out.println(text);
        }
    }
```

3-2. forEach メソッドで書き換えてみましょう

先ほどは、単純に要素を標準出力に出力していました。では、要素のうち、aで始まるものを出力してみましょう。

拡張 for 文で記述すると次のようになります。

```
    private void printList(List<String> texts) {
        for (String text: texts) {
            if (text.startsWith("a")) {
                System.out.println(text);
            }
        }
    }
```

for 文の中に if 文が入っている構造です。

このように if 文で処理を行う対象をより分ける操作、つまりフィルタリングは、ストリームのあのメソッドを使って書けますよね。

3-3. forEach メソッドで書き換えてみましょう

今度は、アルファベットの大文字と小文字が混じっている文字列を要素とするリストを対象とします。

JavaとjAVAのように大文字と小文字が混ざっていると比較する場合などに面倒ですね。ですから全部小文字にしてしまいましょう。文字列をすべて小文字で表すには String クラスの toLowerCase メソッドを使用します。

まず、拡張 for 文で書いてみましょう。

```
    private void printList(List<String> texts) {
        for (String text: texts) {
            String lowerText = text.toLowerCase();

            System.out.println(lowerText);
        }
    }
```

これをストリームで書きかえてみましょう。個々の値に対して、新しい値を作成するには、あのメソッドが使えますよ。

3-4. forEach メソッドで書き換えてみましょう

次に、3-2 と 3-3 を組み合わせてみましょう。

すべて小文字に変換してから、jで始まる文字列を出力します。

拡張 for 文だとこうなります。

```
    private void printList(List<String> texts) {
        for (String text: texts) {
            String lowerText = text.toLowerCase();

            if (lowerText.startsWith(("j"))) {
                System.out.println(lowerText);
            }
        }
    }
```

これをストリームで書きかえてみましょう。今までの応用ですから、簡単ですね。

3-5. forEach メソッドで書き換えてみましょう

文字列のリストに英語の文章が入っているとします。このとき、文章の中にどういう単語が使われているかを調べてみましょう。

これを行うには、文字列を単語単位に切り出す必要があります。

たとえば、使われている単語を出力するのであれば、次のように for 文で記述できます。

```
    private void splitWords(List<String> texts) {
        for (String text: texts) {
            // 文字列から単語を切り出す
            // 正規表現を使って、単語間にあるスペースなどを使用して切り出す
            String[] words = text.split(" |\\.|\\,|\\?");

            for (String word: words) {
                System.out.println(word);
            }
        }
    }
```

for 文がネストしてしまうのが気になるところです。これをストリームの flatMap メソッドを使って書きかえてみましょう。

3-6. forEach メソッドで書き換えてみましょう

今までは拡張 for 文の書きかえでしたが、普通の for 文の書きかえも試してみましょう。

次のコードは 0 から 19 までの数値のうち、偶数を出力するものです。これもストリームで書いてみましょう。

```
    private void printEvens() {
        for (int i = 0; i < 20; i++) {
            if (i % 2 == 0) {
                System.out.println(i);
            }
        }
    }
```

今までのように Stream インタフェースではなく、IntStream インタフェースを使用します。IntStream インタフェースを生成するためのメソッドがありますよね。

4. for 文の変換 (collect)

最後は、終端操作の collect メソッドを使った書き換えです。

collect メソッドは、java.util.stream.Collectors クラスと共に使用して、ストリームの要素を集約する処理を行います。

Collectors クラスにはさまざまなメソッドが定義されています。主なものを以下に示します。

表1 Collectorsクラスの主なメソッド

メソッド名	説明
toList toSet	ストリームをリストもしくはセットに変換する
joining	文字列のストリームの要素を連結した文字列を生成する
groupingBy	ストリームの要素をグルーピングする
partitioningBy	ストリームの要素を条件合致するものと、しないものに分割する
sumingInt sumingLong sumingDouble	ストリームの要素を数値化し、合計する
averagingInt averagingLong averagingDouble	ストリームの要素を数値化し、平均を算出する
summarizingInt summarizingLong summarizingDouble	ストリームの要素を数値化し、統計情報を算出する
counting	要素数を数える
maxBy minBy	要素の最大/最小を求める
reducing	reduceメソッドと同じ処理を行う
mapping	mapメソッドと同じ処理を行う
toMap	マップに変換する

では、これらのメソッドを使って、for 文をストリームに書き換えて見ましょう。

4-1. collect で書き換えてみましょう

はじめに行うのは、リストのコピーです。

for 文でリストのコピーをすることはまずないとは思いますが、練習だと思って、ご了承ください。

```
    private List<String> copyList(List<String> src) {
        List<String> dest = new ArrayList<>();

        for (String element: src) {
            dest.add(element);
        }

        return dest;
    }
```

collect メソッドを使えば、もっと簡潔に書けるはずです。表 1 の一番上に、そういうメソッドがありますね。

4-2. collect で書き換えてみましょう

整数のリストのうち、偶数だけを取り出したリストを作ってみましょう。for 文で書くと次のようになります。

```
    private List<Integer> expressEvens(List<Integer> numbers) {
        List<Integer> evens = new ArrayList<>();

        for (int x: numbers) {
            if (x % 2 == 0) {
                evens.add(x);
            }
        }

        return evens;
    }
```

collect メソッドだけではできませんが、フィルタリングするメソッドも一緒に使えばできますね。

4-3. collect で書き換えてみましょう

4-2 とほぼ同じですが、整数の配列から偶数のリストを作成してみましょう。

```
    private List<Integer> expressEvens(int[] numbers) {
        List<Integer> evens = new ArrayList<>();

        for (int x: numbers) {
            if (x % 2 == 0) {
                evens.add(x);
            }
        }

        return evens;
    }
```

for 文で記述すると、4-2 とほとんど同じになってしまいます。コードは同じなのですが、行っている処理は異なります。何が異なっているか分かりますか? 分かれば、ストリームに書き直す方法も分かるはず。

4-4. collect で書き換えてみましょう

次に文字列の連結を行ってみましょう。

```
    private String joinText(List<String> texts) {
        StringBuilder builder = new StringBuilder();

        for (String text: texts) {
            builder.append(text);
        }

        return builder.toString();
    }
```

これも collect メソッドを使うと、簡単にかけますよ。

4-5. collect で書き換えてみましょう

単語が要素となっているリストを、単語の頭文字でグルーピングしてみましょう。

グループは Map インタフェースで表し、キーが頭文字、バリューが単語のリストになるようにします。

```
    private Map<String, List<String>> groupWords(List<String> words) {
        Map<String, List<String>> groups = new HashMap<>();

        for (String word: words) {
            String key = word.substring(0, 1);
            List<String> group = groups.get(key);
            if (group == null) {
                group = new ArrayList<>();
                group.add(word);
                groups.put(key, group);
            } else {
                group.add(word);
            }
        }

        return groups;
    }
```

for 文で記述すると、ちょっと煩雑ですね。これも collect メソッドを使えば、簡単にかけますよ。

4-6. collect で書き換えてみましょう

4-5 では単語の頭文字でグルーピングしましたが、頭文字ごとに単語がいくつかあったか数えてみましょう。

結果は Map インタフェースで表し、キーが頭文字、バリューが個数とします。

```
    private Map<String, Integer> groupWords(List<String> words) {
        Map<String, Integer> groups = new HashMap<>();

        for (String word: words) {
            String key = word.substring(0, 1);
            Integer count = groups.getOrDefault(key, 0);
            count++;
            groups.put(key, count);
        }

        return groups;
    }
```

Map インタフェースの getOrDefault メソッドが見慣れないメソッドかもしれません。このメソッドは Java SE 8 から追加されたメソッドで、キーが登録されてない場合、第 2 引数がそのまま返るというメソッドです。

今まではキーが登録されていないと、null が戻るので、null かどうか調べてということをやっていました。でも、getOrDefault メソッドを使用すれば、null かどうか気にせずに記述できますよ。

さて、これを collect メソッドで書きかえるわけですが、ちょっと難しいですね。Collectors クラスのメソッドには Collector インタフェースを引数にとるものが多くあります。この Collector インタフェースの引数に、もう一度 Collectors クラスのメソッドを記述すると、集約処理を行った後に、もう一度集約処理を行うことができます。

このようなメソッドを使えば、できるはずですよ。

4-7. collect で書き換えてみましょう

最後はワードカウントを作ってみましょう。文章を要素にもつリストから、単語の使用頻度を調べてみます。

今までやってきたことを組み合わせればできるはず。

```
    private Map<String, Integer> countWord(List<String> texts) {
        Map<String, Integer> wordCounters = new HashMap<>();

        for (String text: texts) {
            // 文字列から単語を切り出す
            // 正規表現を使って、単語間にあるスペースなどを使用して切り出す
            String[] words = text.split(" |\\.|\\,|\\?");

            for (String word: words) {
                word = word.toLowerCase();

                Integer count = wordCounters.getOrDefault(word, 0);
                count++;
                wordCounters.put(word, count);
            }
        }

        return wordCounters;
    }
```

いかがでしたでしょうか。

ちなみに、個々の問題は正解が 1 つとは限りません。いろいろな書き方があり、それに応じて使用するメソッドも変化します。いろいろな書き方をできるようにしておくと、応用範囲が広がるとはずです。