第14回課題

【テーマ】
ファイル入出力

【目的】
適切なファイル操作が行える。
上記が出来るような前提知識を習得する。

【参考】
1．Files (Java Platform SE 8)
https://docs.oracle.com/javase/jp/8/docs/api/java/nio/file/Files.html
2．try-with-resources 文
https://docs.oracle.com/javase/jp/7/technotes/guides/language/try-with-resources.html
3．JavaのGC頻度に惑わされた年末年始の苦いメモリ (3/3)：現場から学ぶWebアプリ開発のトラブルハック（9） - @IT
http://www.atmarkit.co.jp/ait/articles/0712/27/news081_3.html

※難しい内容も含まれているので、一旦読んでみて(不明点があっても)課題に取り組む
その後、再度読んでみて不明点を質問して下さい。

ファイル入出力について

【ポイント】

特にファイル入力について考えると（箇条書き）

①行、ブロックetc単位での一連のデータ群である

②プログラムは入力によってその振る舞いを変化させられる

（→ユーザとのインタフェースとなる）

③リソースの解放（close）が必要（だが忘れてもなんとかなることもある）

が挙げられる。

①について

例として各行に2カラムのCSVファイルを入力として、加算するプログラムをあげると

1,2

3,1

…

これを入力として、両カラムを加算すると

3

4

…

というアウトプットを得られる。

→複数の入力を処理するバッチ処理となる。

②について

①で挙げたファイルに、演算子を足してみると

1,2,+　　（→逆ポーランド記法ですね）

3,1,-

…

これを入力として、両カラムを演算すると

3　（→1+2）

2　（→3-1）

…

というアウトプットを得られる。

→データ群の１つ１つに対して、振る舞いを変える。

③について

ファイルアクセスするクラスを扱う時は、必ず最後にclose()を呼ぶこと。

これを満たすのに一番簡単なのは、try-with-resourcesを用いる方法となる。　→【参考】２．を参照

以下は蛇足的ではあるが「（だが忘れてもなんとかなることもある）」の説明である。

しなくても許されるという意味ではないので、必ずclose()は呼ぶこと。

基本的なファイル入力では凡そ以下のクラスが用いられる。

・FileInputStream

・FileReader

※他にも同時に使用しているが、それらはこのクラスのアダプタとして使用している。

この時、FileReaderの内部を確認すれば分かるが

```
public FileReader(String fileName) throws FileNotFoundException {
    super(new FileInputStream(fileName));
}
```

という形でFileInputStreamが結局のところファイルとのインタフェースになる。

ここでFileInputStreamに話を移すが、以下のメソッドが定義されている。

```
protected void finalize() throws IOException {
    if ((fd != null) &&  (fd != FileDescriptor.in)) {
        /* if fd is shared, the references in FileDescriptor
         * will ensure that finalizer is only called when
         * safe to do so. All references using the fd have
```

```
             * become unreachable. We can call close()
             */
            close();
        }
    }
```

上記はGCが走って、このオブジェクトがGC対象となった後にFinalizerスレッドが呼び出すメソッドである。
→【参考】3．を参照
見て分かるが、必要ならclose()している。

なので、VM側で拾ってくれることもあるが、
・適切なタイミングで自分でするのに比べ、何度もGCがなされるまでリソースが解放されない。
・Finalizerが処理してくれるとは限らない（動く前にプロセスが終われば当然無理です…）
etc…
という理由から、期待してはいけない。
なので、結局のところ「自分でちゃんとリソースの解放をしましょう。」という結論となる。

【適切な実装方法】
まず②を行う為に、データ単位で読み込む必要があります。
ここでは単純化の為に行単位で考えます。
読み込みメソッドは数多ありますが、

トラディショナルに扱う場合
Files.newBufferedReader(path)　Java1.8以降
new BufferedReader(new FileReader(file))　Java1.1以降
→readLine()でnullが返却されるまで1行ずつ読み込む
※データの入れ物としてではなく、ファイルとしての特性を必要とするなら
さらにローレベルのものを使ってください。

Streamの方が簡単だよねという場合（またはcommons-ioのLineIteratorが好きなら）
Files.lines(path)　Java1.8以降
→forEachしましょう。

1passでは処理できない等の理由でメモリに展開したい場合
Files.readAllLines(path)　Java1.8以降
→メモリサイズと相談してください。

上記から分かる通り、Java1.8以降の場合はFilesクラスから適切なメソッドを探せば事足ります。
それ以前のVerの場合も、Filesから探した後にそれと同等機能となるクラスを探せば良いです。

①については、上記を使ったループ処理とすれば難しいことは何もないと思います。

出力についても同様にメソッドが用意されているので、自分で探して使ってみてください。
※これは課題２として後述します。

【実装イメージ】　※上述の通り、出力は敢えて標準出力にしてあります。
トラディショナルに扱う場合

```
    protected static void execBufferedReader() {
        // 好きな方を使って下さい
        //  （おススメはFilesです→今後のVerでさらに効率化された実装クラスに置き換わる可能性なども考えられる為）
        try (BufferedReader reader = new BufferedReader(new FileReader("./data/sample.csv"))) {
        try (BufferedReader reader = Files.newBufferedReader(Paths.get("./data/sample.csv"))) {
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println(sum(line));
            }
        } catch (IOException e) {
            throw new UncheckedIOException(e);
        }
    }
```

Streamの方が簡単だよねという場合

```
protected static void execStream() {
    try (Stream<String> stream = Files.lines(Paths.get("./data/sample.csv"))) {
        stream.forEach(line -> {
            System.out.println(sum(line));
        });
    } catch (IOException e) {
        throw new UncheckedIOException(e);
    }
}
```

1passでは処理できない等の理由でメモリに展開したい場合

```
protected static void execList() {
    List<String> lines;
    try {
        lines = Files.readAllLines(Paths.get("./data/sample.csv"));
    } catch (IOException e) {
        throw new UncheckedIOException(e);
    }
    for (String line : lines) {
        System.out.println(sum(line));
    }
}
```

【課題】

課題１～４まであります。

課題１

【実装イメージ】で3通り提示しましたが、すべて実行して試してみてください。
問題なく動くんだな、ということが理解できれば完了として構いません。
可能ならデバッグでステップ実行してもらうとより理解が深まると思います。

課題２

【実装イメージ】からメモリ展開しないどちらかを選択して、
現在はSystem.outに出力している計算結果をファイルに出力するようにしてください。
今回は出力方法は問いません。（目的とすることが実現できることが大事です）

課題３

現在は【ポイント】①の

```
1,2
3,1
…
```

を対象とした加算のみになっていますが、
これを改造して②の

```
1,2,+
3,1,-
…
```

が処理出来るようにしてみてください。
演算は
「+」：加算
「-」：減算
「*」：乗算
「/」：除算
の4通りで考えてください。
2項演算で構いませんが、お願いされていることすべて終わってしまってものすごく暇です・・・
となるようであれば逆ポーランド記法による複数回演算に改造してみてください。

課題４

出力内容をその行の演算だけではなく、
１つ前の行で行った演算の結果を加算して出力するようにしてみてください。
※１ ： １つ前の行でも、その前の行との加算は行っています。

※２：修正ポイントはsumメソッドでなくても構いません。（IIRフィルタが参考になるかもしれません）

以上