Java 8 のイディオム: 完璧なラムダ式がたった 1 行である理由

コードを理解しやすく、テストしやすく、再利用しやすいコードにするために、単一行のラムダ式を作成する

Venkat Subramaniam
Founder
Agile Developer, Inc.

2018年 3月 15日

ラムダ式を短く簡潔に作成できれば、関数型スタイルのプログラミングがもたらす主なメリットの 1 つである、コードの可読性を高めることになります。読みにくく、複数行にわたるラムダ式は、読んで理解しにくいだけでなく、テストするにも再利用するにも難しいことから、重複する作業を繰り返したり、コードの品質を低下させたりする原因となりがちです。今回のチュートリアルで、理解しやすく、テストしやすく、再利用しやすいコードにするために、単一行のラムダ式を作成する方法を学んでください。

このシリーズの他の記事を見る

このシリーズについて
Java 始まって以来の最も大々的な更新となっている Java 8 には、どこから手を付けてよいのか戸惑ってしまうほど新しい機能が満載されています。このシリーズでは、教育者である著者の Venkat Subramaniam がイディオムを考慮した Java 8 手法を簡潔に紹介し、当たり前のように思ってきた Java の慣例を見直して、プログラムに新しい手法と構文を徐々に取り込んでいくよう読者を導きます。

このシリーズでこれまで学んできたように、関数合成がもたらす主なメリットは、極めて表現豊かなコードを作成できることです。その表現力の鍵を握るのは、短く簡潔なラムダ式を作成することですが、多くの場合、それは口で言うほど簡単なことではありません。この記事を読んで、これまでに皆さんが学んだ、ラムダ式を作成する上での理論と実践に関する理解をさらに深めてください。関数合成の構造とメリットを調べることで、完璧なラムダ式、つまり単一の行に収まるラムダ式をマスターするという目標にぐんと近づけるはずです。

ラムダ式を作成する 2 つの方法

ラムダ式は、ご存知のとおり、匿名関数です。また、ラムダ式は本質的に簡潔なものです。関数やメソッドには、通常、以下の 4 つの要素があります。

- 名前

- 戻り値の型
- パラメーターのリスト
- 本体

この 4 つのうち、以下に示すように、ラムダ式には最後の 2 つの要素しかありません。

```
(parameter list) -> body
```

`->` は、パラメーターのリストを関数の本体から分離するためのものです。関数の本体の目的は、指定されたパラメーターを使用して何らかの処理を行うことです。関数の本体は、単一の式または文にすることができます。以下に一例を示します。

```
(Integer e) -> e * 2
```

上記のコードでは、関数の本体はたった 1 行となっています。この場合の本体は、指定されたパラメーターを 2 回返す式です。信号対ノイズ比は高く、セミコロンや `return` キーワードを使う必要はありません。これが、理想的なラムダ式です。

複数行のラムダ式

Java では、ラムダ式の本体を複合式や複合文にすることもできます。つまり、ラムダ式を複数の行で構成するということですが、その場合はセミコロンが必要です。ラムダ式から結果を返すとしたら、`eturn` キーワードも必要になります。以下に一例を示します。

```
(Integer e) -> {
  double sqrt = Math.sqrt(e);
  double log = Math.log(e);

  return sqrt + log;
}
```

この例のラムダ式から返されるのは、指定されたパラメーターの `sqrt` と `log` の合計です。本体は複数の行で構成されていることから、中括弧 (`{}`)、セミコロン (`;`)、`return` キーワードのすべてが必要になります。

まるで、複数行のラムダ式を作成しようとすると、Java から罰を受けるかのようです。おそらく私たちはこれを受けて気を利かせるべきでしょう。

関数合成の威力

関数型スタイルのコーディングでは、関数合成の表現力を利用します。2 つのコードを比較すると、その表現力のメリットを簡単に見て取れます。以下の最初のコードは、命令型スタイルで作成されたものです。

```
int result = 0;
for(int e : values) {
  if(e > 3 && e % 2 == 0) {
    result = e * 2;
    break;
  }
}
```

上記と同じ内容のコードを、関数型スタイルで作成すると、以下のようになります。

```
int result = values.stream()
  .filter(e -> e > 3)
  .filter(e -> e % 2 == 0)
  .map(e -> e * 2)
  .findFirst()
  .orElse(0);
```

どちらのコードでも同じ結果を得られますが、命令型スタイルのコードでは、`for` ループの内容を読んで、分岐や脱出を経由してフローを辿らなければなりません。関数複合を使用した 2 番目のコードは、遥かに読んで理解しやすいものになっています。このコードは上から下への順で読み進めるようになっているので、一度目を通すだけでコードの内容を理解できます。

基本的に、2 番目のコードでは問題を次のように記述しています。「指定された値のうち、3 より大きな値だけを選ぶ。それらの値から偶数だけを選んで、その値を 2 倍にする。最後に最初の結果を選ぶ。値が存在しなければ、ゼロを返す」。

このコードは簡潔であるだけでなく、その処理量は命令型バージョンのコードと変わりません。`Stream` の遅延評価機能のおかげで、無駄な計算はまったくないのです。

関数合成の表現力は、ラムダ式それぞれの簡潔さに大きく依存します。ラムダ式が複数行にわたっている場合、それが例え 2 行だけであるとしても、関数型プログラミングの急所から逸れてしまうことになるでしょう。

危険なほど長いラムダ式

短く簡潔なラムダ式を作成するメリットをより深く理解するために、逆の場合として、コードが複数行にわたる肥大化したラムダ式の例を見てください。

```
System.out.println(
values.stream()
  .mapToInt(e -> {
    int sum = 0;
    for(int i = 1; i <= e; i++) {
      if(e % i == 0) {
        sum += i;
      }
    }

    return sum;
  })
  .sum());
```

上記のコードは関数型スタイルで作成されているものの、関数型プログラミングのメリットとは無縁です。なぜなのかを考えていきましょう。

1. 読みづらい

優れたコードは、読みたくなるようなものであるはずです。上記のコードを読むには、精神力が必要です。目を凝らして見なければ、コードの各構成部分の始まりと終わりを見つけられないからです。

2. 目的が明確でない

優れたコードは、パズルを紐解くのではなく、物語のように読み進められるものであるはずです。上記のように長々とした匿名のコードでは、その具体的な目的が見えてこないため、読み進めるのに時間も労力も必要になります。このコードを名前付き関数の中に格納すれば、コードがモジュールになるだけでなく、関数の名前から、その目的が見えてきます。

3. コードの品質が悪い

コードの目的が何であれ、いつかは再利用したいと考えるのが当然でしょう。このコードのロジックはラムダ式に埋め込まれていて、ラムダ式は引数として別の関数 `mapToInt` に渡されます。プログラムの他の部分でこのコードが必要になった場合、コードに手を加えて利用したいと思うかもしれませんが、そうするとコード・ベースに不整合が生じることになります。あるいは、コードをそのままコピーして貼り付けるという方法もありますが、いずれにしても、最終的に優れたコード、あるいは高品質のソフトウェアにはなりません。

4. テストしにくい

コードは常に型にはめられたことを行うのであって、必ずしも目的とされていることを行うわけではありません。したがって、重要なコードはいずれもテストするのが当然です。ラムダ式の中にあるコードを単体として使用できなければ、単体テストを行うことはできません。統合テストは実行できるでしょうが、特にかなりの処理を行うコードの場合、統合テストは単体テストに代わるものではありません。

5. コード・カバレッジが限られる

最近のことですが、Java 8 コースの受講者の一人が、ラムダ式は嫌われていると指摘しました。私がその理由を尋ねると、同僚の成果物を見せてくれました。そこでは、数百行にも及ぶコードのラムダ式が使われていたのです。引数に埋め込まれたラムダ式を単体として抽出するのは簡単なことではありません。このことから、カバレッジ・レポートには多数のラムダ式が赤で示されましたが、単体テストを実行できないため、チームはそれらのコードが機能すると前提するしかありませんでした。

グルー・コードとしてのラムダ式

以上の問題のすべてを解決する方法は、ラムダ式をとことん簡潔にすることです。そのための最初の、そして非常に効果的なステップは、単にラムダ式の中で括弧を使用しないようにすることです。以下に、この手法に従って、前の肥大化したラムダ式を簡単に手直しする例を示します。

```
System.out.println(
values.stream()
  .mapToInt(e -> sumOfFactors(e))
  .sum());
```

このコードは完全ではないとしても、簡潔です。また、極めて読みやすいコードです。このコードで表現しているのは、「指定された値の集合を基に、このリストを各数値の因数の合計からなる集合に変換し、その集合の値を合計する」ということです。以前は括弧内にあったコードの本体に明示的に名前を付けると、コードが遥かに読みやすく、理解しやすくなります。これにより、関数パイプラインは短く明瞭になり、そのフローに簡単に従えるようになります。

上記のコードの目的は一目でわかるので、パズルを紐解く必要はありません。因数の合計を計算するコードは、`sumOfFactors` という名前の別個のメソッドにモジュール化されているため、再利用することが可能です。このコードはスタンドアロンのメソッド内にあることから、ロジックの単体テストを簡単に実行することができます。また、このコードはテストしやすいことから、コード・カバレッジが十分であることも保証されます。

要するに、かつての肥大化したラムダ式はグルー・コードになり、ラムダ式で処理の大部分を行うのではなく、名前付き関数を `mapToInt` 関数につなげるだけの役割が与えられたということです。

メソッド参照を使用した微調整

このシリーズで以前に説明したように、ラムダ式をメソッド参照で置き換えることによって、上記のコードをもう少し表現力豊かにすることができます (以下に示されている `sumOfFactors` は、`Sample` という名前のクラスのメソッドです)。

```
System.out.println(
values.stream()
  .mapToInt(Sample::sumOfFactors)
  .sum());
```

作成し直した `sumOfFactors` メソッドは以下のとおりです。

```
public static int sumOfFactors(int number) {
  return IntStream.rangeClosed(1, number)
    .filter(i -> number % i == 0)
    .sum();
}
```

これで、1 つの簡潔なメソッドになりました。メソッドに含まれているラムダ式も同じく簡潔で、余分な形式もノイズもなく、1 行に収まっています。

まとめ

ラムダ式を短く簡潔に作成できれば、関数型スタイルのプログラミングがもたらす主なメリットの 1 つである、コードの可読性を後押しすることになります。複数行にわたるラムダ式には、ノイズを増やしてコードを読みにくくするという逆の効果があります。複数行のラムダは、テストするにも再利用するにも難しいことから、重複する作業を繰り返したり、コードの品質を低下させたりする原因となりがちです。幸い、これらの問題は簡単に回避することができます。その方法は、複数行のラムダ式の本体を名前付き関数に移動して、ラムダ式の中からその関数を呼び出すというものです。また、可能な場合は常に、ラムダ式をメソッド参照で置き換えることを推奨します。

手短に言えば、複数行のラムダ式を使用するのは、複数行のラムダ式が好ましくない理由を説明するときだけにすることをお勧めします。

関連トピック： Using method references in Java 8 Java programming with lambda expressions Java 8 言語での変更内容 Javaによる関数型プログラミング — Java 8ラムダ式とStream (オライリージャパン、2014年) IBM Code: Java Journey