Java 8 のイディオム: 関数型インターフェース

カスタム関数型インターフェースを作成する方法を学び、可能な場合は常に組み込み関数型インターフェースを使うべき理由を学ぶ

Venkat Subramaniam
Founder
Agile Developer, Inc.

2018年 3月 15日

Java 8 では、ラムダ式が関数型インターフェースの型として扱われています。この記事では、この設計上の意思決定が Java 言語の古いバージョンとの後方互換性をサポートする仕組みを説明した後、Java プログラム内でのカスタム関数型インターフェースと組み込み関数型インターフェースの両方の例を見ていきます。カスタム・インターフェースを使うのが当然のように思える場合でも、通常は、組み込みインターフェースを使用するのが最善の結果となる理由を理解してください。

このシリーズの他の記事を見る

ラムダ式の型は何なのでしょう？一部の言語では、関数の値や関数オブジェクトを使ってラムダ式を表現しますが、Java 言語では違います。Java では、関数型インターフェースを使用してラムダ式の型を表現します。初めは奇妙に思えるかもしれませんが、実のところ、これは Java の古いバージョンとの後方互換性を確実にする効率的な方法なのです。

以下のコードはお馴染みのものでしょう。

```
Thread thread = new Thread(new Runnable() {
  public void run() {
    System.out.println("In another thread");
  }
});

thread.start();

System.out.println("In main");
```

`Thread` クラスとそのコンストラクターは、Java 1.0 で導入されました。つまり 20 年以上も前に導入されたことになりますが、それ以来、コンストラクターは変わっていません。コンストラクターには、`Runnable` の匿名インスタンスを渡すことが慣例となっています。けれども Java 8 からは、ラムダ式を渡すという方法を選択することができます。

```
Thread thread = new Thread(() -> System.out.println("In another thread"));
```

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。

`Thread` クラスのコンストラクターが期待するのは、`Runnable` を実装するインスタンスです。上記の例では、オブジェクトを渡すのではなく、ラムダ式を渡しています。このようにラムダ式を渡すという選択肢は、さまざまなメソッドとコンストラクターで使用できます。これには、Java 8 より前に作成されたものも含まれます。こうすることが可能なわけは、Java ではラムダ式が関数型インターフェースとして表現されるためです。

関数型インターフェースには、以下の 3 つの重要なルールが適用されます。

1. 関数型インターフェースには抽象メソッドを 1 つだけ定義できます。
2. この数には、`Object` クラスの public メソッドでもある抽象メソッドは含まれません。
3. 関数型インターフェースに、デフォルト・メソッドと静的メソッドを持たせることもできます。

単一の抽象メソッドというルールに違反していないインターフェースは、どれも当然のこととして、関数型インターフェースと見なされます。このことは、`Runnable` や `Callable` のような従来からのインターフェースにも、独自に作成するカスタム・インターフェースにも適用されます。

組み込み関数型インターフェース

前述の抽象クラスが 1 つだけ定義されたインターフェースに加え、JDK 8 ではいくつかの新しい関数型インターフェースを導入しています。そのうち最もよく使われているのは、`Function<T, R>`、`Predicate<T>`、`Consumer<T>` の 3 つです。これらの関数型インターフェースは `java.util.function` パッケージ内に定義されています。`Stream` の `map` メソッドは、パラメーターとして `Function<T, R>` を取ります。同様に、`filter` メソッドでは `Predicate<T>` を使用し、`forEach` メソッドでは `Consumer<T>` を使用します。このパッケージには他にも `Supplier<T>`、`BiConsumer<T, U>`、`BiFunction<T, U, R>` などの関数型インターフェースがあります。

独自のメソッドに、パラメーターとして組み込み関数型インターフェースを渡すこともできます。例えば、`Device` というクラスに、デバイスが使用中であるかどうかを示す `checkout` と `checkin` というメソッドが定義されているとします。ユーザーが新しいデバイスをリクエストすると、`getFromAvailable` メソッドが使用可能なデバイスのプールから 1 つのデバイスを返すか、必要に応じて新しいデバイスを作成します。

デバイスを借りるための関数を実装するとしたら、以下のような関数になります。

```
public void borrowDevice(Consumer<Device> use) {
  Device device = getFromAvailable();

  device.checkout();

  try {
    use.accept(device);
  } finally {
    device.checkin();
  }
}
```

上記の `borrowDevice` メソッドの内容は以下のとおりです。

- `Consumer<Device>` をパラメーターとして取ります。
- プールからデバイスを取得します (この例では、スレッド・セーフについては考えません)。
- `checkout` メソッドを呼び出して、デバイスのステータスを「チェックアウト済み」にします。
- デバイスをユーザーに配布します。

ユーザーの `accept` メソッドに対する呼び出しからデバイスが戻ると、`checkin` メソッドの呼び出しによって、デバイスのステータスは「チェックイン済み」に変更されます。

以下に `borrowDevice` メソッドの使用例を示します。

```
new Sample().borrowDevice(device -> System.out.println("using " + device));
```

このメソッドは関数型インターフェースをパラメーターとして取るため、ラムダ式を引数として渡すことは許容されます。

カスタム関数型インターフェース

可能な場合は常に組み込み関数型インターフェースを使用するのが最善策ですが、場合によっては、カスタム関数型インターフェースが必要になることもあります。

独自の関数型インターフェースを作成するには、2 つの慣例に従う必要があります。

1. インターフェースに `@FunctionalInterface` でアノテーションを付けます。これは、Java 8 でのカスタム関数型インターフェースの表記規則です。
2. 関数型インターフェースには抽象メソッドを 1 つだけ定義します。

以上の慣例によって、インターフェースがラムダ式を受け取るように意図されていることが明確になります。コンパイラーは `@FunctionalInterface` アノテーションを見つけると、そのインターフェースに定義されているのが 1 つの抽象メソッドだけであることを確認します。

このアノテーションを使用すると、将来インターフェースに変更を加えるときに誤って抽象メソッドの数のルールに違反した場合には、エラー・メッセージを受け取れるようになります。これによって問題をすぐに捉えられるため、問題をそのままにして、別の開発者が後で対処しなければならないといった事態を避けることができます。他の開発者が作成したカスタム・インターフェースにラムダ式を渡してエラー・メッセージを受け取るのは、誰にとっても嫌なことです。

カスタム関数型インターフェースを作成する

一例として、`OrderItems` のリストとそのリストを変換して出力するメソッドを持つ `Order` クラスを作成しましょう。まずは、インターフェースから取り掛かります。

以下のコードでは、`Transformer` 関数型インターフェースを作成します。

```
@FunctionalInterface
public interface Transformer<T> {
  T transform(T input);
}
```

インターフェースには `@FunctionalInterface` アノテーションでタグを付けて、これが関数型インターフェースであることを表明します。このアノテーションは `java.lang` パッケージに含まれているので、インポートは必要になりません。このインターフェースに定義された、`transform` という名前のメソッドは、パタメーター化された型 `T` のオブジェクトを取り、同じ型の変換後のオブジェクトを返します。変換のセマンティクスは、インターフェースの実装によって決まります。

`OrderItem` クラスは以下のとおりです。

```
public class OrderItem {
  private final int id;
  private final int price;

  public OrderItem(int theId, int thePrice) {
    id = theId;
    price = thePrice;
  }

  public int getId() { return id; }
  public int getPrice() { return price; }

  public String toString() { return String.format("id: %d price: %d", id, price); }
}
```

`OrderItem` クラスは、2 つのパラメーター (`id` および `price`) と `toString` メソッドだけが含まれる単純なクラスです。

次は、`Order` クラスを見てください。

```
import java.util.*;
import java.util.stream.Stream;

public class Order {
  List<OrderItem> items;

  public Order(List<OrderItem> orderItems) {
    items = orderItems;
  }

  public void transformAndPrint(
    Transformer<Stream<OrderItem>> transformOrderItems) {

    transformOrderItems.transform(items.stream())
      .forEach(System.out::println);
  }
}
```

`transformAndPrint` メソッドはパラメーターとして `Transform<Stream<OrderItem>>` を取り、`transform` メソッドを呼び出して `Order` インスタンスに属する注文品目を変換し、変換した順に注文品目を出力します。

以下に、このメソッドを使用する例を記載します。

```
import java.util.*;
import static java.util.Comparator.comparing;
import java.util.stream.Stream;
import java.util.function.*;

class Sample {
  public static void main(String[] args) {
    Order order = new Order(Arrays.asList(
      new OrderItem(1, 1225),
      new OrderItem(2, 983),
      new OrderItem(3, 1554)
    ));


    order.transformAndPrint(new Transformer<Stream<OrderItem>>() {
      public Stream<OrderItem> transform(Stream<OrderItem> orderItems) {
        return orderItems.sorted(comparing(OrderItem::getPrice));
      }
    });
  }
}
```

`transformAndPrint` メソッドには、引数として匿名内部クラスを渡します。`transform` メソッドの中で、指定されたストリームの `sorted` メソッドを呼び出して、注文品目をソートさせます。以下に示すこのコードの出力には、価格の昇順でソートされた注文品目が示されています。

```
id: 2 price: 983
id: 1 price: 1225
id: 3 price: 1554
```

ラムダ式の威力

関数型インターフェースを使用すべき場合は、常に 3 つの選択肢があります。

1. 匿名内部クラスを渡す
2. ラムダ式を渡す
3. 場合によっては、ラムダ式ではなく、メソッド参照を渡す

匿名内部クラスを渡すとなるとコードが冗長になります。また、メソッド参照を渡せるのは、パススルー・ラムダ式の代替手段としてメソッド参照を使用する場合のみです。`transformAndPrint` 関数の呼び出しを作成し直して、匿名内部クラスを渡すのではなく、ラムダ式を使用するようにしたら、どうなるでしょうか？

```
order.transformAndPrint(orderItems -> orderItems.sorted(comparing(OrderItem::getPrice)));
```

このほうが、最初の匿名内部クラスを使用した場合よりも、遥かに簡潔で、読みやすいコードになっています。

カスタム関数型インターフェースと組み込み関数型インターフェースの比較

例として作成したカスタム関数型インターフェースは、カスタム・インターフェースを作成するメリットとデメリットを明らかにしています。最初にメリットから見ていきましょう。

- カスタム・インターフェースには、他の開発者が変更または再利用する際にその内容がすぐにわかるような名前を付けることができます。`Transformer`、`Validator`、`ApplicationEvaluator` などの名前はドメインに固有のものなので、他の開発者がインターフェースのメソッドを読めば、引数として何が期待されているのかを推測できます。
- 抽象メソッドには、構文上有効な名前であれば、どのような名前でも付けることができます。この仕様は、インターフェースを受け取る側にとってだけのメリットであり、しかも抽象メソッドを渡す場合に限られます。ラムダ式やメソッド参照を渡す呼び出し側にとってのメリットはありません。
- インターフェース内で、パラメーター化された型を使用できます。あるいは、少数の型に固有の単純なインターフェースにすることもできます。その場合は、パラメーター化された型 `T` ではなく、`OrderItems` を使用するように `Transformer` インターフェースを作成することができます。
- カスタムのデフォルト・メソッドと静的メソッドを作成して、インターフェースの他の実装でも再利用可能にすることができます。

当然、カスタム関数型インターフェースを使用する場合のデメリットもあります。

- 複数のインターフェースを作成して、そのすべてに同じシグネチャーを持つ抽象メソッド (例えば、`String` をパラメーターとして取り、`Integer` を返すなど) を定義する場合を考えてください。メソッドの名前は違っているとしても、それらのメソッドのほとんどは冗長なものであり、総称名を持つ 1 つのインターフェースで置き換えられる可能性があります。
- カスタム・インターフェースを使用する場合は例外なく、学習し、理解し、記憶するための努力が余計に必要になります。Java プログラマーであれば、誰もが `java.lang` パッケージに含まれる `Runnable` を十分に理解しています。何度も繰り返し目にしているため、努力しなくても、`Runnable` の目的を記憶できます。その一方、私がカスタム `Executor` を使用したとしたら、他の開発者はこのインターフェースを使用する前に、インターフェースの目的を慎重に調べなければなりません。それだけの努力をする価値がある場合もありますが、`Executor` があまりにも `Runnable` に似ているとしたら、無駄な努力に終わってしまします。

どちらが最善か？

カスタム関数型インターフェースと組み込み関数型インターフェースの長所と短所を踏まえて、皆さんはどちらを使用するかをどのようにして決めますが？再び `#ransformer` インターフェースを取り上げて、その方法を考えましょう。

`Transformer` は、あるオブジェクトを別のオブジェクトに変換する際のセマンティクスを伝えるためのインターフェースであることを思い出してください。以下のコードでは、このインターフェースを名前で参照しています。

```
public void transformAndPrint(Transformer<Stream<OrderItem>> transformOrderItems) {
```

`transformAndPrint` メソッドが受け取る引数によって、変換の内容が左右されます。例えば、`OrderItems` コレクションに含まれる要素が並べ替えられることもあるでしょう。あるいは、変換によって、各注文品目の詳細の一部をマスクする場合もあれば、処理を行わないことを決定して元のコレクションを返すだけの場合もあります。実装は、呼び出し側に任されます。

肝心な点は、変換の実装を引数として `transformAndPrint` メソッドに渡せるということを、呼び出し側が把握することです。これらの詳細は、関数型インターフェースの名前ならびにその資料で明らかにしなければなりません。この例の場合、パラメーターの名前 (`transformOrderItems`) からも詳細がわかるので、`transformAndPrint` 関数のマニュアルにこのパラメーターの情報を含める必要があります。関数型インターフェースの名前は、その目的と使用法を知るのに役立ちますが、手掛かりはそれだけではありません。

`Transformer` インターフェースを細かく調べて、その目的を JDK の組み込み関数型インターフェースと比べると、`Function<T, R>` で `Transformer` を置き換えられる可能性があることがわかります。それをテストするために、`Transformer` 関数型インターフェースをコードから削除して、`transformAndPrint` を以下のように変更します。

```
public void transformAndPrint(Function<Stream<OrderItem>, Stream<OrderItem>> transformOrderItems) {
  transformOrderItems.apply(items.stream())
    .forEach(System.out::println);
}
```

この変更は大きなものではありません。具体的には、`Transformer<Stream<OrderItem>>` を `Function<Stream<OrderItem>, Stream<OrderItem>>` に変更し、メソッド呼び出しを `transform()` から `apply()` に変更しています。

`transformAndPrint` の呼び出しで匿名内部クラスを使用しているとしたら、それも変更する必要がありますが、この呼び出しではラムダ式を使用するように変更済みです。

```
order.transformAndPrint(orderItems -> orderItems.sorted(comparing(OrderItem::getPrice)));
```

関数型インターフェースの名前は、ラムダ式には関係しません。名前は、引数としてのラムダ式をメソッド・パラメーターに結び付けるコンパイラーに対してだけ関係します。メソッド名が `transform` であるか、`apply` であるかも、呼び出し側には無関係です。

組み込み関数型インターフェースを使用することで、インターフェースの数が 1 つ少なくなり、しかもメソッド呼び出しはまったく同じように動作します。また、コードの可読性にも妥協していません。この演習からわかるように、カスタム関数型インターフェースは組み込み関数型インターフェースで簡単に置き換えることができる可能性があります。その場合に必要なのは、(ここでは記載していませんが) `transformAndPrint` の資料を用意して、引数に記述的な名前を指定することだけです。

まとめ

ラムダ式を関数型インターフェースの型にするという設計上の意思決定により、Java 8 とそれ以前のバージョンの Java との後方互換性が促され、通常は単一の抽象メソッド・インターフェースを

受け取るような古い関数にもラムダ式を渡せるようになっています。メソッドがラムダ式を受け取るには、そのパラメーターの型が関数型インターフェースでなければなりません。

場合によっては、独自の関数型インターフェースを作成するのが妥当なこともありますが、その場合は慎重に作成する必要があります。カスタム関数型インターフェースを使用するかどうかを検討するのは、アプリケーションに極めて特化されたメソッドが必要な場合、あるいは必要を満たす既存のインターフェースがない場合のみにしてください。常に、JDK の組み込み関数型インターフェースのなかに、必要な機能があるかどうかを調べて、可能な場合は常に、組み込み関数型インターフェースを使用してようにしてください。

関連トピック： Functional interfaces 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)