Javaの イケてるコード 残念なコード

そのコーディングスタイルはもう古い? Javaの新定石を学ぶ

Acroquest Technology
谷本 心

プログラミング言語には、決まりきった「定石」と呼べる書き方が多数存在します。多くの書籍やWebサイトでも、定石に沿ったコードや定石に類するコードを掲載していることが多々あります。

しかしその中には、一昔前は定石と考えられていたものの、現在は定石とは呼べなくなっているものがたくさんあります。Java自体が進化したために、現在ではより良い書き方ができるようになっているのです。そこで今回は、昔と現在では書き方が変わっている「Javaのイマドキの定石」について紹介します。

リスト1●一昔前のfor文

```
public void doSomething(List list) {
  for (int i = 0; i < list.size(); i++) {
    String str = (String) list.get(i);
    // strを使った処理
  }
}
```

ループカウンタを利用する

リスト2●イテレータとwhile文で処理を記述した

```
public void doSomething(List list){
  Iterator iterator = list.iterator();
  while (iterator.hasNext()) {
    String str = (String) iterator.next();
    // strを使った処理
  }
}
```

whileとイテレータでリストをループする

リスト3●ジェネリクスによってListの型を指定できるようになった

```
public void doSomething(List<String> list) {
  for (int i = 0; i < list.size(); i++) {
    String str = list.get(i);
    // strを使った処理
  }
}
```

Listの中身はString型に限定される

リスト4●foreach文を利用したプログラム

```
public void doSomething(List<String> list) {
  for (String str : list) {
    // strを使った処理
  }
}
```

Listが終わるまで順番に処理していく

リストのループ処理

Javaプログラムでは、Listを定義して各要素をループさせるといった処理をよく記述します。一昔前では、**リスト1**のように、Listを処理するためにループカウンタを使ったfor文を記述していました。ループカウンタをインクリメントしながら順番に処理します。C言語と記述が似ているため、とっつきやすい構文だといえます。また、性能を考慮して記述すると、イテレータ(Iterator)とwhile文を利用して処理を記述することもできます(**リスト2**)。

しかし、リスト1と2の書き方はもはや古い定石となっています。イマドキの定石を使うと、どのように記述できるのでしょうか。

J2SE 5.0からジェネリクス(Generics)が導入され、ListやMapの型を指定できるようになりました。これまでは、Listの要素をObject型としてしか取り出せなかったのですが、ジェネリクスを使えば好きな型で取り出せるようになります(**リスト3**)。

ジェネリクスは、＜型＞で記述します。リスト3のListは、String型限定のListになります。このようにジェネリクスを使うことで、Listから取り出したObject型を別の型にキャストする必要もなくなったほか、Listに誤って意図しない型のインスタンスを入れてしまいClassCastExceptionを発生させることもなくなりました。J2SE 5.0以降では、ジェネリクスの利用は、もはや定石というより鉄則ともいえるでしょう。

さらにJ2SE 5.0では、ループに関する便利な構文が導入されました。それがforeach文、あるいは拡張for文と呼ばれる構文です。Listや配列などを順番に処理する際に利用できる構文で、他のプログラミング言語でもよく見かけます。この構文を利用することで、先ほどのコードは**リスト4**になります。

もちろん、for文の中でループカウンタの値を使いたい場合は、これまで通りのfor文で記述する必要がありますが、それ以外の場合には、コードを短く記述できるため積極的にforeach文を使うべきでしょう。

このジェネリクスとforeach文のおかげで、Listや配列の処理をより安全かつシンプルに記述できるようになりました。

リソースのクローズ処理

この連載でも以前に紹介しましたが、ファイルの読み込みやデータベースの接続などで利用するリソースは、必ずfinally節の中でクローズすること、という定石があります。まずは、**リスト5**をみてください。

実は、リスト5は残念なコードです。なぜ残念なコードなのかわかりますか。この書き方では、1つ上の行のreadLineメソッドの実行時に例外が発生した場合、closeメソッドが呼び出されないのです。その結果、readerがクローズされないままで処理が進んでしまい、ファイルを開いたままの状態になってしまいます。

この問題を防ぐために、finally節でcloseするという定石があります(**リスト6**)。処理が正常に終わった際も、例外が発生した際も、closeメソッドが確実に呼び出されるため、ファイルが開きっぱなしになることはありません。

しかし、どうしても記述が冗長になってしまうことや、closeメソッドで例外が発生した際の例外処理をどうすべきか決めにくい、などの問題がありました。多くの場合は、closeメソッドの例外を無視していました。それでは、イマドキのコードはどう書くべきか紹介しましょう。

リスト6の問題をまとめて解決してくれるのが、Java SE 7から導入された「try-with-resources」です。tryの中でリソースを宣言することで、自動的にリソースをクローズしてくれるというものです(**リスト7**)。

tryの丸カッコ内でリソースを宣言している部分が、try-with-resourcesのポイントです。一見すると、Readerなどのリソースのcloseメソッドが呼ばれていないように見えます。ですが、実際にはtryで宣言したリソースは、finallyと同じタイミングで自動的にcloseメソッドが呼び出されます。しかも、このtry-with-resourcesでは、例外が発生した場合に「最初に発生した例外」をthrowするようになっています。

例えば、closeメソッドだけで例外が発生した場合には、closeのIOExceptionを投げ(throw)、readLineとcloseの両方で例外が発生した場合には、先に発生したreadLineメソッドの実行時のIOExceptionを投げるようになっています。さらに、throwされたExceptionのgetSuppressedメソッドを利用すれば、後から発生した例外を取り出すこともできます。

Java SE 7以降のリソース処理では、このtry-with-resourcesを使うことで、安全にリソースをクローズできる上に、うまく例外も取り出すこともできるのです。

定数クラスは static import を使う

アプリケーション開発において、画面サイズや処理コードなどの定義するために、定数を宣言することがあります。その際に、たくさんの定数を1つのクラスにまとめた定数クラスや定数インタフェースを作る、という定石があり

リスト5●readLineメソッドで例外が発生してしまうとクローズ処理ができない

```
String message = null;
try {
  Reader reader = new BufferedReader(
    new FileReader("test.txt"));
  message = reader.readLine();
  reader.close();
} catch (IOException ex) {
  // 例外処理
}
```

クローズ処理を記述しているがreadLineメソッドで例外が発生すると呼ばれない

リスト6●必ずfinally節の中でクローズするという古い定石で書いたプログラム

```
String message = null;
Reader reader = null;
try {
  reader = new BufferedReader(
    new FileReader("test.txt"));
  message = reader.readLine();
} catch (IOException ex) {
  // 例外処理
} finally {
  if (reader != null) {
    try {
      reader.close();
    } catch (IOException e) {
      // この例外は無視する?
    }
  }
}
```

closeメソッドの例外はどうする?

リスト7●try-with-resourcesを使ったプログラム

```
try (InputStream in = new FileInputStream("test.txt");
Reader inReader = new InputStreamReader(in);
Reader reader = new BufferedReader(inReader)) {
  reader.readLine();
} catch (IOException e) {
  // 例外処理をする
}
```

tryの中でリソースを宣言している