列挙型の活用

MICHAEL KÖLLING

既知の定数値の集合を型保証（type-safe）された列挙型で表現し、起こりがちな問題を未然に防止する

enum型とも呼ばれる列挙型は、もっと頻繁に使われてよいJavaの構成要素です。確かに列挙型は人々を熱狂させるような際立ったコンセプトではなく、使用を強制されるようなものでもありません。どちらかと言えば、コードの信頼性を向上させ、読みやすくすることによって、静かに改善を図るものです。

Java初心者であっても、列挙型を使わずにきちんと動作するよいコードを書けるという方はたくさんいます。

列挙型を使わない方々にも理由があります。まず、初期のJavaには列挙型が存在しませんでした。プログラマーの中には、列挙型が追加されたJava 5より前からJavaを学んでおり、その頃の習慣を変えていないという方もいるでしょう。また、列挙型がサポートされていない言語から移ってきた方もいるでしょう。さらに、列挙型を使わなくても問題を完全に解決できるため、使う必要性を感じないという方もいるかもしれません。しかし、いずれも列挙型を無視し続ける正当な理由にはなりません。

列挙型を使用すると、コードは堅牢になり、型を保証でき、エラーが起こりにくくなり、美しさも向上します。すなわち、コードを劇的に改善できます。これはとても重要なことです。ぜひ本記事をじっくりとお読みください。

列挙型を使う場面と理由

それでは、例を通して列挙型の使用方法を見てみましょう。ここでは、コンピュータ・ゲームの名作コロッサル・ケーブ・アドベンチャー（Colossal Cave Adventure）やゾーク（Zork）のようなテキストベースのアドベンチャー・ゲームを書くことを考えます。そのために、ユーザーが入力できるコマンドの集合を用意します。有効なコマンドを、「go」「look」「take」「help」「quit」とします。

コードのどこかで、上記のコマンドを定義することになるでしょう。最初に単純な実装として、次のように文字列の配列で定義することを考えます。

```
private static final String[] validCommands = {
    "go", "look", "take", "help", "quit"
};
```

入力されたこれらのコマンドを処理するコードを、プログラムの別の場所に書くことになります。そのコードでは、コマンドを実行する適切なメソッドを呼び出します。次のコード・スニペットは、入力されたコマンドがString型の変数commandWordに格納されているものとしています。

```
switch (commandWord) {
    case "go":
        goRoom(secondWord);
        break;
    case "look":
        look();
        break;
    case "take":
        takeItem(secondWord);
        break;
    case "Help":
        printHelp();
        break;
    case "quit":
        quit();
        break;
}
```

写真：JOHN BLYTHE

（それぞれのコマンドに対して一連のint定数を定義し、入力文字列を数値に変換してからswitch文でint定数を使うという方法も考えられます。これもよく使われる方法ですが、上記の例と同じ問題を抱えています。）

この方法の何がいけないのでしょうか。実は、即座に明白となる2つの根本的な問題が存在します。型の安全性と国際化です。

まず、型の安全性について説明しましょう。前述の短いコード・セグメントは、とても単純でわかりやすいものです。もちろん、きちんと動作すると思うでしょう。実はそうではありません。このコードにはバグがあります。見つかったでしょうか。

問題は、switch文のhelpコマンドにタイプ・ミスがあり、Helpとなっていることです。commandWordは常にいずれかの有効なコマンド文字列であるべきと意図してはいましたが、この方法では、無効なコマンドの割当てや、無効なコマンドとの比較を防ぐすべはありません。型をStringと宣言しているため、あらゆる文字列を格納することができます。つまり、この型システムはあまりよいものではありません。宣言済みの型では、許容する値の集合を厳密に記述することができません。型システムで検知できない限り、（論理的に）正しくない値を使うこともできてしまいます。

解決策は列挙型

この問題を避けるために、列挙型を使ってコードを書き直すことができます。まずは、列挙型の宣言を記述します。

```
public enum CommandWord
{
    GO, LOOK, TAKE, HELP, QUIT
}
```

この宣言はクラスと同様に扱い、独立したファイルに記述する必要があります。宣言によってCommandWord型が定義され、リストされている5つの名前がこの型の有効な値となります。別のクラスでは、この型の変数を宣言して値を割り当てることができます。たとえば、次のようになります。

```
CommandWord command = CommandWord.GO;
```

そして、前述のswitch文を次のように書き換えることができる点がポイントです。

```
switch (commandWord) {
    case GO:
        goRoom(secondWord);
        break;
    case LOOK:
        look();
        break;
    case TAKE:
        takeItem(secondWord);
        break;
    case HELP:
        printHelp();
        break;
    case QUIT:
        quit();
        break;
}
```

文字列配列ではなく列挙型を使用したこのバージョンでは、コマンドの定義がシンプルでわかりやすくなっただけでなく、型の安全性も実現しています。case文のラベルや割り当てる値にタイプ・ミスがあっても、コンパイラが検知して知らせるようになりました。これこそ本物の勝利です。Javaで意図されている強力な型システムを取り戻すことができました。

なお、列挙型では、等しいかどうかのチェックに、文字列に使用する必要のあったequals()メソッドの代わりに二重等号（==）を使うこともできます。

```
if (command == CommandWord.QUIT) ...
```

列挙型の正体

列挙型をint定数と同じようなもの、つまり割当て可能な、後で識別できる名前付きの値と考え、ここで説明した方法でのみ使用している方もいるでしょう。しかし、それは列挙型のすべての側面ではありません。ここで学習を止めてしまうと、表面をなぞっただけで終わってしまい、列挙

型の重要な機能のいくつかを見逃すことになります。

列挙型の宣言は完全なクラスで、リストされている値はそのクラスの別々のインスタンスを参照する定数名です。列挙型の宣言には、他のクラスと同じように、フィールド、コンストラクタ、メソッドを含めることができます。先ほどの列挙型を拡張したものを次に示します。

```
public enum CommandWord
{
    GO("go"), LOOK("look"), TAKE("take"),
    HELP("help"), QUIT("quit");
    private String commandString;

    CommandWord(String commandString)
    {
        this.commandString = commandString;
    }

    public String toString ()
    {
        return commandString;
    }
}
```

以下で、いくつかの重要な特徴について説明します。

- 列挙型の宣言はクラスであり、列挙型の値はオブジェクトを参照しています。
- 宣言されている列挙型の値1つにつきクラスのインスタンスが1つ作成され、その値に割り当てられます。
- 宣言後にクラスの別のインスタンスを作成することはできません。
- それぞれの列挙型の値は別々のオブジェクトを参照しており、同じ値は常に同じオブジェクトを参照しています。これを変更することはできません。
- 列挙型は自身の名前空間を作成します。複数の列挙型のクラスで同じ値を使用できますが、それらの値は別々のものとなります。たとえば、BoardGamesという列挙型クラスがある場合、列挙型の値BoardGames.GOとCommandWords.GOは別々のものであり、お互いに干渉することはありません。

別のインスタンスを作ることができないという最後の特徴は、コンストラクタをprivateにすることによって実現しています。明示的にprivateと宣言する必要はありません。コンストラクタは自動的にprivateになり、publicにしようとするとエラーになります。

先ほどのコードは、それぞれの値に1つずつ、合計5つの列挙型オブジェクトを生成します。他のコードから参照できるCommandWordオブジェクトは、この5つの列挙型オブジェクトのみです。別のオブジェクトを作成しようとすると、コンパイル時にエラーが発生します。

列挙型に含めることができるフィールド、コンストラクタ、メソッドの数には制限はありません。「通常の」クラスと根本的に違うのは、列挙型のインスタンスがどのように生成されるかという点です。通常のクラスでは、インスタンスが何も存在しないところから始まり、クライアントにコンストラクタを提供して必要なだけオブジェクトを作らせます。一方の列挙型は、（外部には）コンストラクタを提供せず、出来合いのインスタンスの集合を提供します。

列挙型の値はオブジェクトであり、intではないことがポイントです。つまり、列挙型は識別だけでなく、状態と振る舞いも提供します。

全容

ここで気になるのは、外部から呼べないのであれば、コンストラクタは何のために存在するのかという点でしょう。

その答えは、列挙型の値を列挙する際に変更した構文からわかります。最初の例では、

```
GO
```

と書いていたものを、次のように書き換えました。

```
GO("go")
```

実質的に列挙型の値にパラメータ・リストを追加することで、列挙型のコンストラクタが呼び出されます。括弧内の式は、コンストラクタに渡される実パラメータです。列挙型オブジェクトがCommandWordクラスであることは変わりませんが、内部に文字列の属性が保存されるようになりました。この属性値は、コンストラクタを通じて列挙型オブジェクトに渡されます。他のオブジェクトと同様に、列挙型オブジェクトの内部、すなわちインスタンスフィールドには、任意の型の属性を任意の数だけ格納できます。

ここでもう一度CommandWordの定義を見てみましょう。少しずつわかり始めてきたのではないでしょうか。

この方式の大きなメリットは、入力されたコマンド（たとえば「help」）を認識するために、入力文字列と、列挙型の内部に格納されているコマンド文字列とを比較する際にStringを再度使用できるようになったにもかかわらず、プログラム・ロジックとその文字列が切り離されている点にあります。

先ほど、最初のバージョンのプログラムが抱えるもう1つの問題は国際化だと述べました。このプログラムを別の言語（「help」コマンドが「hilfe」になるとします）に翻訳する場合、不具合が混入する危険があります。配列の中のコマンドを変更しただけでは、コンパイルはできますが動作しません。コマンドはまったく認識されないにもかかわらず、エラーが起きることはありません。問題は、文字列が入力だけでなく、プログラム・ロジックでも使われていることです。これはいけません。

新しいバージョンの列挙型を使うと、この問題は解消します。実際にコマンド文字列を記述するのは一度だけで済む上、変更する場合も1箇所を変更すればよいだけです。プログラム・ロジックは論理値（列挙型の定数）と連携して動作し続けます（実際には、入力コマンドはロケールに依存したテキスト・ファイルから読み出すのが一般的ですが、原理は変わりません）。

内部の仕組み

列挙型は実際にはクラスとして実装されており、列挙型の値はそのインスタンスです。これには少しばかり特別な点があります。それを知っておくと、列挙型の仕組みや列挙型を使って何ができるかを理解するのに役立ちます。

すべての列挙型のクラスは、EnumというJava標準クラスを自動的に継承します（これは、別のクラスを拡張して列挙型のクラスにすることはできないことも意味します）。その際に、いくつかの便利なメソッドも継承されます。

継承されるメソッドで知っておくべきなのは、name()とordinal()に加え、静的メソッドのvalues()です。

name()メソッドは列挙型の値で定義された正確な名前を返します。ordinal()メソッドは、列挙型が宣言された順序を表す、0から始まる数値を返します。たとえば、

```
CommandWord cmd = CommandWord.GO;
System.out.println(cmd.name());
System.out.println(cmd.ordinal());
```

を実行すると、次の結果が表示されます。

```
GO
0
```

実際には、この2つのメソッドを使う場面は想像ほど多くはないでしょう。通常は、実際の列挙型の名前に依存するようなコードを書くべきではありません（つまり、name()メソッドはあまり役に立たないということです。この目的では、toStringメソッドをオーバーライドして使う方がはるかによいでしょう）。同様に、うまく書けているコードで序数を使う必要があることはめったにないでしょう。

それに比べて、静的メソッドのvalues()ははるかに便利です。列挙型のすべての値で構成される配列を返すため、それらの値に対して反復処理を行うことができます。例を見てみましょう。

```
CommandWord[] ca = CommandWord.values();

for (CommandWord cw : ca) {
    System.out.println(cw);
}
```

Java 8のストリームを使いこなせる方なら、次のように書くこともできます。

```
Arrays.stream(ca).forEach(System.out::println);
```

列挙型シングルトン・パターン

内部での列挙型の実装の仕組み（もっとも重要な点は、インスタンス作成のメカニズムは異なるものの、クラスにすぎないという点です）がわかったところで、便利な使い方について見てゆきます。著者がよく紹介する例の1つに、シングルトン・パターンの実装に列挙型を使うというものがあります。

シングルトンは、そのクラスのインスタンスが1つしか存在しないことを保証したい場合に使用します。一般的には、クラス内でインスタンス

を作成し、コンストラクタをprivateにして、インスタンスを渡す静的なファクトリ・メソッドを提供することによってシングルトンを実現します。次の例をご覧ください。

```
public class Singleton {
    private static Singleton instance =
        new Singleton();

    public static Singleton getInstance()
    {
        return instance;
    }

    private Singleton()
    {
        ...
    }
}
```

外部からは、次のように記述してシングルトンのインスタンスにアクセスできます。

```
Singleton s = Singleton.getInstance();
```

列挙型を使うと、シングルトンをとても簡単に定義できます。

```
public enum EasySingleton {
    INSTANCE;
}
```

他に何の作業も必要なく、インスタンスにはクライアントのコードから簡単にアクセスできます。

```
EasySingleton s = EasySingleton.INSTANCE;
```

シングルトンのクラスには、今までと同じようなフィールドやメソッドを追加できます。列挙型のインスタンス作成はデフォルトでスレッド・セーフであるため、この手法はマルチスレッド・アプリケーションでも安全に使用できます。

まとめ

今回の簡単な説明で、列挙型のメリットを十分に感じていただけたのではないかと思います。定数値の集合を定義する際は常に、表現の選択肢として列挙型を最優先に考えるとよいでしょう。この選択により、型の安全性、国際化のサポート、コンパイル時のコーディング・エラー警告などが実現します。何よりも、コードの可読性が向上し、エラーが起こりにくくなります。**</article>**

Michael Kölling: イギリスのケント大学教授。オブジェクト指向やコンピューティング教育について多数の論文を発表しているほか、Java教本2冊を出版。教育用プログラミング環境であるBlueJおよびGreenfootの開発リーダーも務める。
Oracle Java Champion、UK National Teaching Fellow、UK Higher Education Academyのフェロー、およびACMのDistinguished Educator。

learn more

[オラクルによるJavaの列挙型に関するチュートリアル]