第4章. 適切な変数のスコープを書こう

読みやすいプログラムを書くには、変数の使用用途に合わせて、適切なスコープを書くことが大切です。
スコープとは、その変数がクラスやメソッド内で使用できる適用範囲のことで、変数の適用範囲はプログラムの読みやすさに繋がり、なるべく狭くする必要があります。
この章では、スコープを狭くする方法やテクニックを説明します。

4－1. 適切なアクセス修飾子を使おう

スコープが広くなるということは、他のクラスからも自由に変数の値を参照したり変更したりすることができたり、予期せぬバグが起こることがあります。多くの場合、そのようなスコープの広い変数をむやみに多様すると、プログラムのデバッグが難しくなります。どこで値を変更し、どのタイミングで値を参照しているのかを解析するのに、苦労することがあります。

×：不適切なアクセス修飾子を使っている例

```
/*
 *メッセージクラス.
 *<br>
 *メッセージクラスは、メッセージを出力する処理を行う。
 */
public class Message {

    // フィールド
    public String messageStr = "Hello";

    // コンストラクタ
    public Message() {
    }

    public void print() {
        System.out.println(message);
    }
}

/*
 *メインクラス.
 *<br>
 *メインクラスは、mainメソッドを実行する。
 */
public class MessageMain {
    public static void main(String[] args) {
        Message message = new Message();

        // 不正な値の書き換え
        message.messageStr = "こんにちは";
        message.print();
    }
}
```

・フィールドのアクセス修飾子が「public」なので、Messageクラス以外の他のクラスからも自由に値を参照したり変更することができる。

○：適切なアクセス修飾子を使っている例

```
/*
 *メッセージクラス.
 *<br>
 *メッセージクラスは、メッセージを出力する処理を行う。
 */
public class Message {

    // フィールド
    private String messageStr = "Hello";

    // コンストラクタ
    public Message() {
    }

    public void print() {
        System.out.println(message);
    }
}

/*
 *メインクラス.
 *<br>
 *メインクラスは、mainメソッドを実行する。
 */
public class MessageMain {
    public static void main(String[] args) {
        Message message = new Message();

        // 不正な値の書き換え
        message.messageStr = "こんにちは";
        message.print();
    }
}
```

フィールドのアクセス修飾子「public」から「private」に変更。

また、フィールドは他のクラスから再度メソッドが呼ばれるまで覚えておかなくてはならない値をフィールド変数として保持する性質があります。フィールドを使用する場合は、フィールドのアクセス修飾子を「private」にして、メソッドのアクセス修飾子を「public」にし、カプセル化にしましょう。 カプセル化にすることで、他のクラスからのアクセスを制限することができ、スコープが狭くなります。

アクセスメソッドを使用するクラスで使うことが多いです。

アクセスメソッドでのカプセル化の例

```
/**
 * 顧客情報格納クラス.
 * <br>
 * 顧客情報格納クラスは、顧客情報を格納する。
 */
public class CustomerBean {

    /** 顧客情報登録フィールド */
    private String userid;
    private String password;
    private String familyname;
    private String firstname;

    // アクセスメソッド
    public String getUserid() {
        return userid;
    }

    public void setUserid(String userid) {
        this.userid = userid;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
    :
    :
}
```

・フィールドのアクセス修飾子を「private」にして、メソッドのアクセス修飾子を「public」にし、カプセル化になっている。

そして、変数の値を保持する必要がない場合は、フィールドに宣言したりしないで、そのメソッドを実行する時だけ有効な「ローカル変数」を使うようにしましょう。変数のスコープはなるべく狭くするように心掛けましょう。

4－2. 変数は必要になる箇所で、変数の宣言をしよう

変数は可能な限り使用する位置で宣言を行い、宣言時に初期化を行なうようにしましょう。

C言語のように、全ての変数をメソッドの先頭で宣言してしまうと、Javaでは、変数の宣言と変数の使用が切り離されるため、ソースコードを追うのが大変になり、読みづらくなります。 また、一つのローカル変数が、異なる用途で再利用される可能性が高くなり、予期せぬバグが発生することもあります。

×：不適切な位置で変数を宣言しているソースコード

```
public static void main(String[] args) {

    int[] nums = {3,2,4,1,5};
    int sum = 0;
    int i = 0;
    MyArrays arrays = new MyArrays();

    // ①nums配列要素を表示
    for (i = 0; i < nums.length; i++) {
        System.out.print(nums[i] + " ");
    }

    // ②nums配列要素の合計値を算出
    for (i = 0; i < nums.length; i++) {
            sum += num[i];
    }

    System.out.println("合計値は"+ sum);

    int[] sortedNums = arrays.sort(nums);

    // ③nums配列要素を昇順に並び替えて表示
    for (i = 0; i < sortedNums.length; i++) {
        System.out.print(sortedNums[i] + " ");
    }
}
```

・i変数はループカウンタなのに、メソッドの先頭で宣言しているので、読みづらい。
・sum変数は、②の処理で使用するが、①の処理が間に入っている状態なので、読みづらい。

メソッドの先頭でインスタンス化しているため、実際に使用するタイミングと離れているので、分かりづらい。

○：適切な位置で変数を宣言しているソースコード

```
public static void main(String[] args) {

    int[] nums = {3,2,4,1,5};

    // ①nums配列要素を表示
    for (int i = 0; i < nums.length; i++) {
        System.out.print(nums[i] + " ");
    }

    // ②nums配列要素の合計値を算出
    int sum = 0;
    for (int i = 0; i < nums.length; i++) {
            sum += num[i];
    }

    System.out.println("合計値は"+ sum);

    MyArrays arrays = new MyArrays();
    int[] sortedNums = arrays.sort(nums);

    // ③nums配列要素を昇順に並び替えて表示
    for (int i = 0; i < sortedNums.length; i++) {
        System.out.print(sortedNums[i] + " ");
    }
}
```

Javaでは、ループカウンタは、for文内で宣言、初期化する。

sum変数の位置を、②の処理の直前に変更。
どの変数を使うのか明確になり、処理が分かりやすい。

インスタンス化の位置を、メソッド呼び出しの直前に変更。
どのインスタンス変数を使うのか明確になり、処理が分かりやすい。

Javaでは、変数は可能な限り使用する位置で宣言することで、ソースコードはより安全になり、バグが発生しにくくなり、そして読みやすくなります。

４－３. 値を格納するだけの変数はやめよう

変数が多くなればなるほど、プログラムの行数が増えて複雑になり、プログラムの流れが追いづらくなります。

例えば、値が変更になったりインスタンス化する時等は、変数が必要になりますが、値を格納するだけの変数は、その時しか変数を使っていないことになります。処理が短ければどのような処理が行われているかは容易に理解できる場合が多いので、値を格納するだけの変数を用意することは、やめましょう。

４－４. ジェネリクスの型パラメータは、必ず型パラメータの指定をしよう

ジェネリクスを適用せずに、直接キャストする記述の場合、実行時に例外が発生することがあります.。
また、コレクションから取り出す時等に一つ一つキャスト(型変換)が必要になり、バグが発生しやすくなります。

・直接キャストとジェネリクス適用

```
×：List sampleList = new ArrayList();
○：List<String> sampleList = new ArrayList<String>();
```

コレクション等、ジェネリクスが適用されているインターフェースやクラスを使用する時は、必ず型パラメータの指定を行なうようにしましょう。

４－５. 絶対パスを使うのはやめよう

サーブレット環境における URL では、使用用途別に指定方法が異なります。
絶対パスとは、階層構造の頂点(最上位階層)から目的のファイルやフォルダまでの経路を全て記述する方式になり、 相対パス(URL)とは、起点となる現在位置から、目的のファイルやフォルダまでの経路を記述する方式になります。

例えば、「http://localhost:8080/webapp/sample」と入力したとします。
それぞれの名前意味は以下の通りになります。
「プロトコル://ホスト名:ポート番号/コンテキストパス/サーブレットパス」

```
プロトコル　　：http 通信方法をもちいて
ホスト名　　：自分のコンピューターに対して通信する
ポート　　：通信対象となるソフトウェアはである Tomcat である（8080 以外にも変更できます）
コンテキストパス：Tomcat で起動している Web アプリケーション名
サーブレットパス：実行するサーブレットの指定
```

ここで問題なのがコンテキストパスの部分です。これは、物理フォルダ名とは全く関係性がありません。

開発する時は、webappというフォルダの中に色々Webアプリケーションとして必要なファイルを配置しますが、 実際 Tomcat 上で Web アプリケーションを起動する際、フォルダ名を変えずに別の名前で起動することができます。

極端な話、起動する度にコンテキストパスを変更しようと思えば、書き換えることができてしまいます。

上記を踏まえて、使用用途別に以下のように URL を指定しましょう。

- 別のコンピュータ上で起動している Web アプリケーションにアクセスする場合
  →絶対パス　例）http://otherdomain/xxxx/xxxxxx

- 同一コンピュータにおいて起動している別の Web アプリケーションにアクセスする場合
  →コンテキストパスから指定　例）/otherContext/xxxxxx

- 同一コンピュータ・同一 Web アプリケーション内の別のサーブレット、html、JSP にアクセスする場合
  →現在位置から指定（相対パス） 例）./otherSevlet