3-7.thisとsuperについて

■ thisとsuperについて

はじめに

`this` と `super` っていうのがあるんだなとは学んできましたが、
実際にはどういったケースで使用するのか？

ふわっとした理解の方も少なくないと思いますので、本章でしっかりと解説していきます！

Step1: thisとsuperの復習

概念的な部分を簡単に復習していきます。

・`this`
自クラスのインスタンスであることを明示的に指し示す機能を持ちます。

・`super`
継承（`extends`）元の親クラスのインスタンスであることを明示的に指し示す機能を持ちます。

> **サンプル: 継承**

【会社員: Employee】

今回は会社の社員をイメージしていきます。

```
// （大きな概念としての）社員クラス
public class Employee {

    /** empId: 社員番号*/
    private int empId;

    /** name: 氏名*/
    private String name;

    /** コンストラクタ: 引数なし */
    public Employee() {
    }

    /**
      * コンストラクタ: 引数あり
      * @param empId
      * @param name
      */
    public Employee(int empId, String name) {
        this.empId = empId;
        this.name = name;
    }

    // ... フィールド変数のゲッターとセッター（※省きます）

    /**
     * 社員情報をコンソールへ出力するメソッド
     */
    public void printEmployeeInfo() {
        System.out.println("社員番号: " + this.empId);
        System.out.println("氏名    : " + this.name);
    }
}
```

【自社の社員: MyEmployee】

?

```
// 社員（Employee）を継承した会社の社員クラス
public class MyEmployee extends Employee {

    /** isTrainee: 研修生フラグ（研修生であるかどうかを true/false で判断するためのフィールド変数） */
    boolean isTrainee;

    public MyEmployee() {
    }

    /**
      * コンストラクタ: 引数あり
      * @param empId
      * @param name
      */
    public MyEmployee(int empId, String name) {
        // ①挙動確認: 親クラス（Employee）のコンストラクタを呼び出す
        super(empId, name);

        // ③挙動確認: thisを指定せずに格納する その1
        boolean isTrainee;
        if (empId != 10192) {
            isTrainee = true;
        } else {
            isTrainee = false;
        }
        System.out.println(this.isTrainee ? "研修生です" : "研修生ではありません");
    }

    /**
      * コンストラクタ: 引数あり
      * @param empId
      * @param name
      * @param isTrainee
      */
    public MyEmployee(int empId, String name, boolean isTrainee) {

        // ①挙動確認: 親クラス（Employee）のコンストラクタを呼び出す
        super(empId, name);

        // ②挙動確認: 自クラスのフィールド変数（isTrainee）へコンストラクタの引数を格納する
        this.isTrainee = isTrainee;

        // ③挙動確認: thisを指定せずに格納する その2
        isTrainee = true;
        System.out.println(this.isTrainee ? "研修生です" : "研修生ではありません");

        // ④挙動確認: thisとsuperで同じメソッドを呼び出す
        this.printEmployeeInfo();
        super.printEmployeeInfo();
    }

    // ... フィールド変数のゲッターとセッター（※省きます）
}
```

Step2: this／superの使い方を知る

> **使い方**

自クラスのインスタンスのフィールド変数／メソッドの前に付与して、、、

- 自 or 親クラスのフィールド変数や、メソッド（コンストラクタ含む）を優先的に参照する場合
- 自 or 親クラスのフィールド変数と、メソッド（コンストラクタ含む）の引数や、メソッド内で一時的に使用するローカル変数を区別する場合

上記のような場合に使用します。

基本的には `this` と `super` を使用する際の軸となる考え方は一緒です。

Step3: thisとsuperの挙動について

実際にコードを通してみていきます。
先に提示した `Employee` と `MyEmployee` を参考に見ていきます。

> **①挙動確認: 親クラス（Employee）のコンストラクタを呼び出す**

・ `super` **の実体は、** `Employee` **（親クラス）**
`MyEmployee` のコンストラクタ（二箇所）で呼び出していますが、
継承先でオーバーライドせずに親クラスのコンストラクタを利用しています。

ポピュラーの使用方法ですね。
特に異なる初期化処理などを行う必要がなければこれでよいです。

> **②挙動確認: 自クラスのフィールド変数（isTrainee）へコンストラクタの引数を格納する**

・`this` **の基本の使い方**
`this` を付与して正しい値の受け渡しを行えています。

フィールド変数の `isTrainee` と 引数の `isTrainee` を明確に区別した使用法です。

> **③挙動確認: thisを指定せずに格納する**

・自クラスのフィールド変数なのか、ローカル変数／引数なのかが分かりづらい
`this` を付与していないことが原因で **ローカル変数／引数の値を変更してしまっています。**

この場合、起こりうる結果としては以下になります。

【その1の場合】

- フィールド変数へ値を格納することを意図していたとしても、実際に値が格納されるのはローカル変数

【その2の場合】

- フィールド変数へ値を格納することを意図していたとしても、実際に値が格納されるのは引数
- 仮に②と③の処理順が逆であった場合でも、実際に値が格納されるのは引数
-

何故か？

実はコンパイル時には以下のような優先順位が存在します。

「ローカル変数／メソッドの引数」 ＞ 「フィールド変数」

ローカル変数とメソッドの引数が同名の場合 は、コンパイル時に :warning: ワーニングが **表示されます** ので即時対応が可能なのですが、
フィールド変数とローカル変数（or メソッドの引数）の場合 は、:warning: ワーニングは **表示されません。**

> **【実行結果: その1】**

```
// mainメソッドなどで、以下のように定義
MyEmployee be = new MyEmployee(10192, "田中");

...

// ③挙動確認: thisを指定せずに格納する その1
boolean isTrainee;
if (empId != 10192) {
    isTrainee = true;
} else {
    isTrainee = false;
}
System.out.println(this.isTrainee ? "研修生です" : "研修生ではありません");
```

【出力結果】

```
研修生ではありません
```

> **解説**

`boolean` 型の値に関しては、初期値が `false` となっていますので、
この場合であれば、フィールド変数（ `isTrainee` ）の値と、 条件分岐後の格納値は一致しているため、
初期化時に期待する結果（フィールド変数の `isTrainee` が `false` であること）は変わりません。

しかし、 `MyEmployee be = new MyEmployee(0, "田中");` のようにコンストラクタを呼び出した場合は、
条件分岐の結果`true` が格納されてしまいますので、 フィールド変数（ `isTrainee` ）は意図した初期化ができないでしょう。

上記はフィールド変数に値を格納したいはずなので、
`this` **を付与すべきですし、更に言えば 同名のローカル変数は避けるべき** です！

それでも一時的にローカル変数を用意したいのであれば、以下のようなコーディングが望ましいです。

【「一時的な」という意味合いの単語: Temporary を使用する】

```
// Temporary を省略して tmp としている
boolean tmpIsTrainee;
if (empId != 10192) {
    tmpIsTrainee = true;
} else {
    tmpIsTrainee = false;
}
this.isTrainee  = tmpIsTrainee;
```

【一番スッキリ書けて無駄が無い書き方】

ベストは以下の書き方ですが、
初めのうちはワンラインコーディングを避け、
`if else` の **ブロック** の考え方を意識しましょう！

```
// (empId != 10192)の結果値である true/false が戻り値として格納される
this.isTrainee  = (empId != 10192);
```

> **【実行結果: その2】**

```
...
isTrainee = true;
System.out.println(this.isTrainee ? "研修生です" : "研修生ではありません");
```

【出力結果】

```
研修生ではありません
```

この出力結果は当然と言えば当然なのですが、
この処理において `isTrainee = true;` が、
`this.isTrainee = true` を意図してコーディングしたものである場合は話が変わってきますね。

このように、明示的に `this` を付与しない場合は、結果が大きく変わってしまう可能性があります。

> **④挙動確認: thisとsuperで同じメソッドを呼び出す**

・オーバーライドをしていない場合は、親クラスのメソッドを呼び出す
今回の場合、 `this.printEmployeeInfo()` と `super.printEmployeeInfo()` は、
両方共に親クラスである `Employee` の `printEmployeeInfo()` を呼び出しています。

逆に言えば、 オーバーライドすることにより、上記の `this` と `super` の呼び出し先は変わるということです！

【サンプル: MyEmployeeクラスで `printEmployeeInfo()` をオーバーライド】

```
...

/**
 * 社員情報をコンソールへ出力するメソッド（研修生フラグを含む）
 */
public void printEmployeeInfo() {
    super.printEmployeeInfo();
    System.out.println("研修生フラグ: " + this.isTrainee);
}
```

上記のようにオーバーライドした場合、 `this` は自クラスでオーバーライドした `printEmployeeInfo()` を参照します！

> **全体的なポイント**

実装時はそのクラスの内容に沿った（適した）変数名・メソッドを実装していく訳ですが、
その際に意識する重要な点が、、、

「値を受け渡す元と先の変数に関して、意味的にも実値的にも等価（値として同じ）である場合は、同様の命名規則とすること」

です。

難しく言葉を並べましたが、要するに、
`name` という変数に `shimei` という変数（引数） を格納するような、
「 別の値が格納されるのでは？？？」
という、 **疑問が生じるような命名** は極力避けるべきであるということです。

とは言いつつも、そういったコードが **絶対NGかと言われればそうではありません。**

```
// 社員情報をセットするようなメソッド
private void setEmployeeInfo(int paramEmpId, String paramName) {
    this.empId = paramEmpId;
    this.name = paramName;
}
```

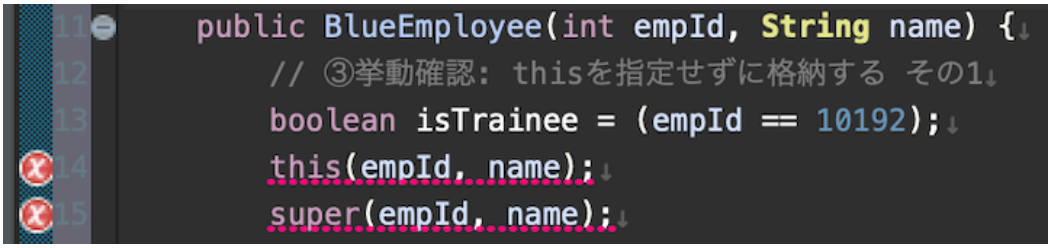
上記のように、 **（変数名と引数名は異なるが）等価のパラメーターであること** を明示的に示した引数名であれば、許容度は高いと言えます。

異なる命名ではありますし、`this` が無くても動作はしますが、
上記のように `this` **を「付与しない」 よりは「付与する」を選択してコーディング** できると、
コード量が膨大になり、似たような命名の変数やメソッドを使用するなった場合であっても、
コーディング／コードリーディングで迷子になることも減ることでしょう！

Step4: thisやsuperが許容されないケース

> **こんな場合はエラー その１**

【コンストラクタの前に処理が存在する】



③の処理をコンストラクタの前に移した状態です。
この場合、以下のエラーが表示されます。

「コンストラクター呼び出しは、コンストラクター内の最初のステートメントである必要があります」

画像のように、コンストラクタ内の処理としてコンストラクタ（ `this/super` ）を使用する場合は、
前処理が無い状態、つまり、処理の一番時最初に記述しなければなりません。

※例としてあげてはいますが、コンストラクタを2つ連続して並べる書き方は上記のエラーが出力されるため実現不可なので注意しましょう！

> **こんな場合はエラー その２**

【static修飾子が関係する】

static修飾子が付与されたクラス変数／定数、メソッドに関しては、
`this／super` は付与することができません！

【サンプル: staticブロック内で呼び出す】

mainメソッドが `static` であるため、 `callPrintln()` も必然的に `static` で定義する必要があります。

```
public class Main {

    public static void main (String[] args) {
        this.callPrintln(); // ←これはコンパイルエラー！
        Main.callPrintln(); // ←これが正解！
    }

    private static void callPrintln() {
        System.out.println();
    }
}
```

解説

・ `static` **メソッド内では、** `this` **を使用できない**
上記ケースで呼び出す場合は、 `Main` クラスを付与して呼び出します。

【サンプル: static以外のブロック呼び出す】

```
...
    // staticを外してMain内では呼び出さない状態のメソッドへ修正
    private void callPrintln() {
        System.out.println("callPrintln");
        this.callPrintln2();  // ←ワーニングが表示される！
        Main.callPrintln2(); // ←これが正解！
    }
    // もうひとつstaticのメソッドを作成
    private static void callPrintln2() {
        System.out.println("callPrintln");
    }
```

解説

この場合は以下のワーニングが表示されます。

「型 *Main* からの *static* メソッド *callPrintln2()* には *static* にアクセスする必要があります」

型 *Main* というのは、実際のところこのクラス名（ `Main` ）になります。

ワーニング解消パターンとしては、 基本的には二つです。

- 先頭にクラス名を付与
- callPrintln2()の定義にstaticを追加

結果、staticのブロック内／外どちらの場合においても、
`this` **ではなく、自クラスをしっかりと明記すべし！** ということです。

staticについての詳細は、 **final修飾子・static修飾子** にて。

さいごに

`this` と `super` はコーダー視点で、気配りであり、実装の最適化を実現するために必要ではありますが、
実際に動作するアプリケーションのユーザー視点ではあまり関係ないお話です。

しかし、アプリケーションの作りとしてわかり易く美しいことは、やはり正義です！
現場に入って他人が書いたコードを読み解く際に、 `this` や `super` が明示的に記述されているコードかどうかは、
そのコードやファイル単位という狭い範囲ではなく、ひとつのアプリケーションとして作りが精巧である理由としてひとつの判断材料と言えるでしょう！

課題

提出課題はありませんので、一通り学習が終わったら次の章に進んで下さい。

最終更新日時: 2022年 08月 21日(日曜日) 15:39