3-8.final修飾子・static修飾子

■ final修飾子

はじめに

Javaには継承やオーバーライドなど便利な機能があることを学んできました。
しかし場合によっては元のクラスやメソッドの機能を変えるべきでないときもあります。
その時に使うのが **final修飾子** です。

Step1: 概念を知る

final修飾子
　決めたことを変更できなくするように、プログラマーが指定する機能です。
　クラス、メソッド、変数に指定することができ、その働きは以下の通りです。

① クラスに付けた場合　→　そのクラスを継承することができなくなる
② メソッドに付けた場合　→　そのメソッドをオーバーライドすることができなくなる
③ 変数に付けた場合　→　その変数は変更不可（定数）となる

Step2: 使い方を知る

> **final の使い所**

変数として扱う必要が無い場合は `final` を付与する癖をつける 。

特に **メソッドのパラメーターや、メソッドのローカル変数** において、
条件分岐ごとの処理による値の書き換えの可能性が無いような場合は、 `final` で定義するほうがよいです。

【サンプル: 定数】

```
final String CONST_MSG_SUCCESS_LOGIN = "ログイン成功です。";
final String CONST_MSG_ERROR_INPUT = "入力情報に誤りがあります。";
```

> **解説**

・定数を定義する
固定で表示するダイアログのメッセージや、
基本的に **一度確定したらそれ以降変更する必要がないような値** に関しては定数とし定義しておくとよいです。
（※業務上は、定数を定義することを「定数を切る」など表現したりします。

また、個々のjavaファイルへ定義するよりかは、 `CommonConstant` などと定数専用のクラスを定義して、
その中にまとめて定義することが多いです。

【サンプル: ロジック】

■ 仕様（ざっくり）
入力値を受け取り、入力値に沿ったメッセージを返し、コンソールへ出力する
としましょう！

?

```
import java.util.Objects;

/**
 * 入力値ごとのメッセージを返却する
 * @param paramInput <pre>入力値</pre>
 * @return
 */
public String validateInput(String paramInput) {
    String message = null;
    if (Objects.isNull(paramInput)) {
        // 入力値が未存在
        message = "nullです！";

    } else if (paramInput.isBlank()) {
        // こちらはJava 11～提供のメソッド
        // 入力値がブランク（” ”）←空文字ではなく、半角スペースがある状態
        message = "ブランクです！";

    } else if (paramInput.isEmpty()) {
        // 入力値が空文字（””）
        message = "空文字です！";

    } else {
        // 入力値が存在
        message = "入力値あります！";
    }
    return message;
}

// 呼び出し
String message = validateInput("入力値");
System.out.println(message);
```

> **解説**

1. `validateInput` メソッドのパラメーター `String paramInput`
2. `validateInput` メソッドのローカル変数 `String message = null;`
3. `validateInput` メソッドの返り値を扱う `String message`

もし仮に上記のサンプルをAさんがコーディングしたとしましょう。
その後、仕様を知らない（or 忘れている）Bさんが修正を入れたとした場合、
`message` の値が `message = "テキトーな値"` と書き換えられてしまう可能性は否めません。

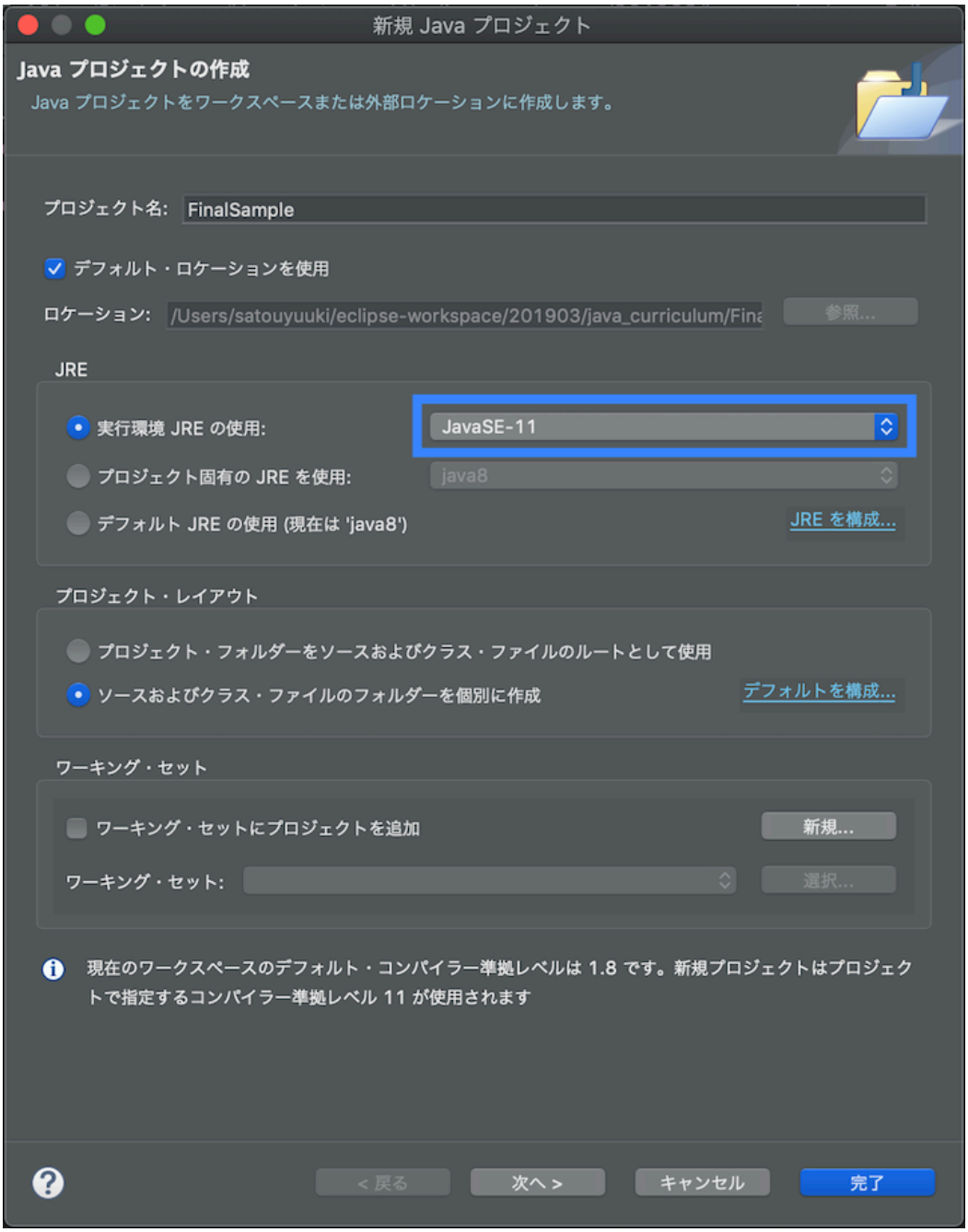
そのため、上記3つのうち1、3に関しては、
仕様 を考慮した上で見ていくと、
使用箇所以外での使いみちが無いため、本来であれば `final` が望ましいです。

こういう些細な気配りが見えるコードは、バグを生み出す原因を潰してくれているので、
共同開発者としてはとても嬉しいものになります。

> **Java 11 のプロジェクト**

`} else if (paramInput.isBlank()) {` の `isBlank()` **メソッド** を使用したい場合は、
Javaプロジェクト自体のコンパイラーのバージョンを上げる必要が出てきます。

画像の青枠部分を切り替えることで作成するプロジェクトのJavaのコンパイラーバージョンを操作することができます。
（※eclipse 2018-12 以降であれば、デフォルトでJavaSE-11が選択できるはずなので、試したい場合は画像のように変更して作成してみるといいでしょう。



■ static修飾子

はじめに

まだstaticの意味がよく分かっていない人も、書いたことは何回もあるんじゃないでしょうか？
mainメソッドを書く時に出てきますよね。mainメソッドはstaticなメソッドなんです。

Step1: 概念を知る

・static修飾子
メソッドや変数に付けることによって、
インスタンス化しなくても、クラスから直接メソッドや変数にアクセスすることができるようになります。
staticなメソッドや変数のことを静的メソッド、静的変数あるいはクラスメソッド、クラスフィールドと呼びます。

一方で、 `static` を付けずにインスタンスを生成してアクセスするものは、インスタンスメソッド、インスタンスフィールドと呼びます。
こちらはインスタンス化が必要です。

クラスメソッドからインスタンスメソッドやインスタンスフィールドにアクセスすることはできませんので注意が必要です。
（※インスタンスクラスからクラスメソッド、クラスフィールドにアクセスすることは可能です。）

Step2: 使い方を知る

`static` を付ける、または付けないという違いを知ってそれぞれを使い分けることができると、コーディングの幅が広がります！！

- **・クラス内の変数（クラス変数）**
  「static変数」は「 `クラス名.` 」の後に記述しますので「クラス変数」
- **・インスタンス固有の変数（インスタンス変数）**
  「非static変数」は「 `インスタンス名.` 」の後に記述しますので「インスタンス変数」

> **static変数**

static変数（クラス変数）は以下のように宣言します。

```
アクセス修飾子（publicなど） static 型名 変数名
```

宣言したクラス変数を呼び出す場合は以下のようにします。

```
クラス名.変数名
```

> **staticメソッド**

`staticメソッド` は、クラスに対してただひとつのメソッドです。

```
public class HelloJava {

    public static String getMessage() {
        return "Hello";
    }
}
```

`staticメソッド` は、そのクラスでどんなにインスタンスの生成をしても、 **ただひとつ** のメソッドを指します。
インスタンスフィールドはインスタンスごとに値を保存する ことができますが、
クラスフィールドはインスタンスを複数生成しても値が共有される ので、
ただ一つのクラスフィールドにアクセスするようになっています。

そのため、実はインスタンスを生成しなくてもメソッドを呼び出すことが可能です。
「 `クラス名.メソッド名()` 」という形式で `staticメソッド` を呼び出せます。
（※逆に、クラスフィールド・メソッドをインスタンスを生成した上で呼び出すと、その必要が無いためワーニングが表示されます。

```
String message = HelloJava.getMessage();
```

> **補足**

「ただひとつのメソッドを指します。 」と上述していますが、ピンとこない方もいるかと思いますので補足です。

世界には同姓同名の人々が多数存在しますよね。
しかしながら、 **あなたは世界に一人だけ** です。

例えば、あなたが佐藤さんだとして、男性なのか女性なのか。
あるいは、東京住みなのか、神奈川住みなのか。
そういった違いがありますよね。

「 **インスタンスの同一であるとか、ひとつであるとか** 」のイメージって、上記に近いものがあります。

簡単にですが、staticについて学ぶ前に **インスタンス** について復習 & 必要知識を伝授していきます！

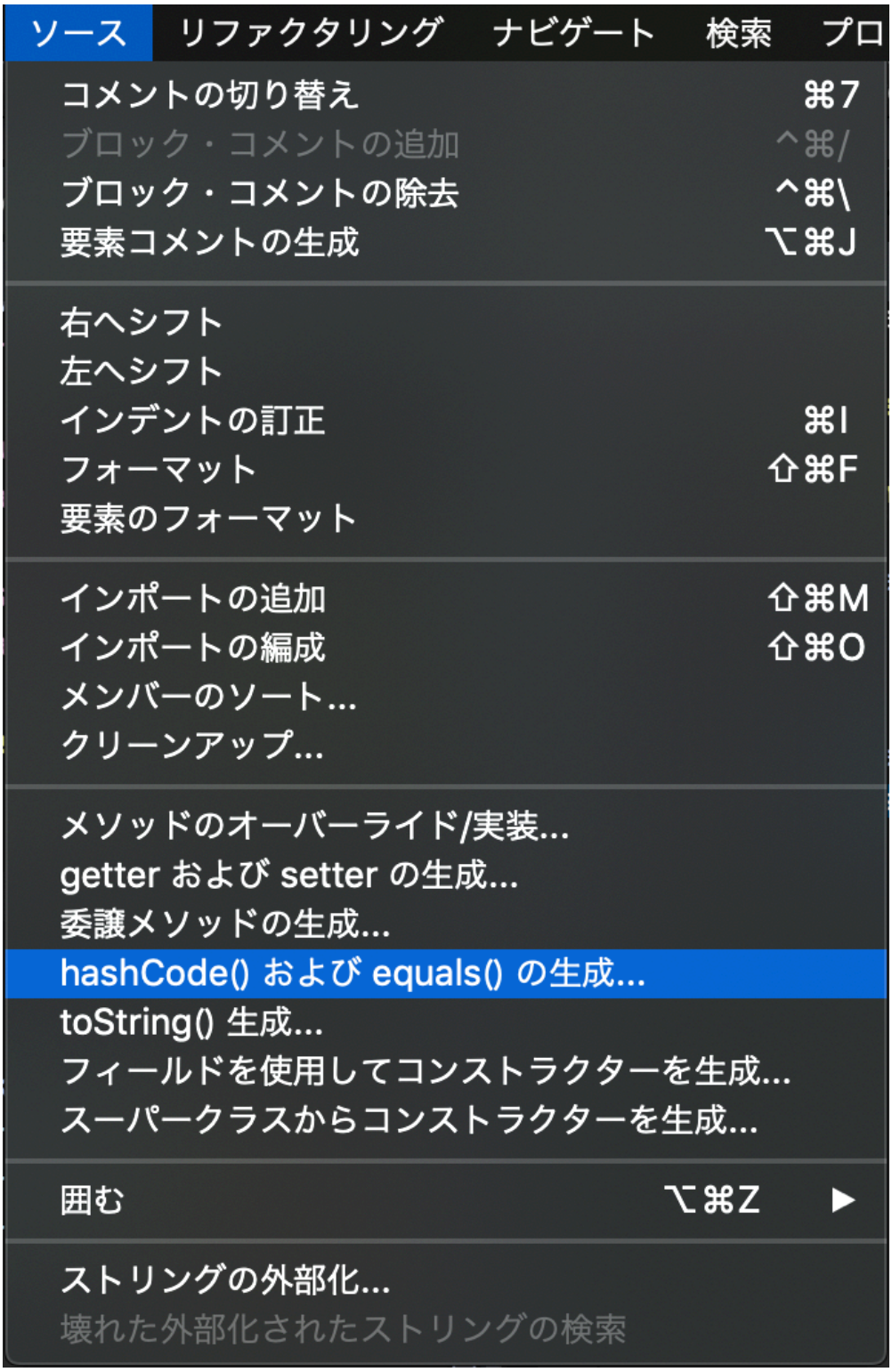
サンプル用のJavaプログラムを作成していきます。

1. インスタンスを生成するためのクラスの作成

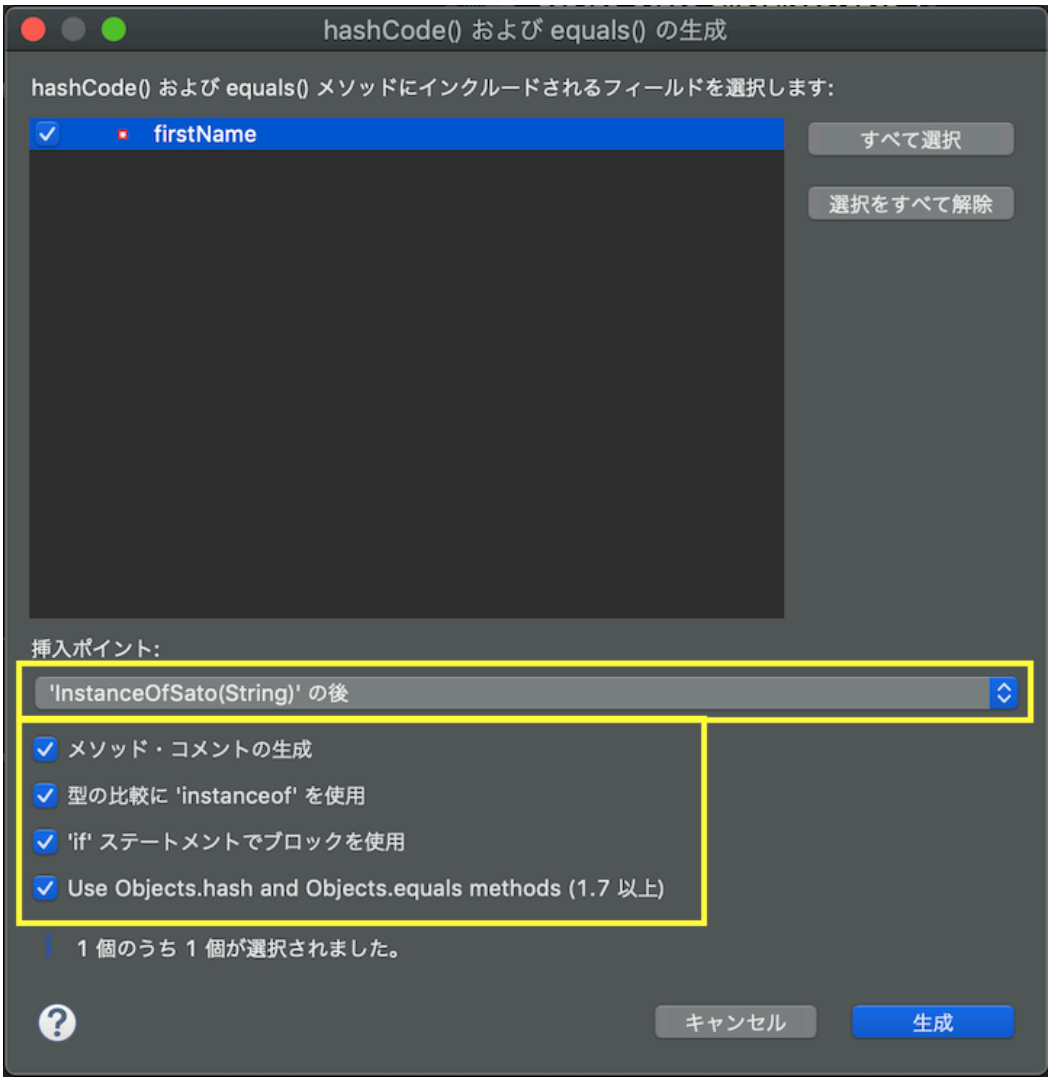
下準備として以下のコードを用意しておきます。
※今回は佐藤さんを題材にします。

2. Eclipseメニューにある自動生成機能を使って必要なメソッドを揃える

以下の画像のように「hashCode() および equals() の生成...」を選択します。
※該当クラスにフィールド変数がある状態で実施すること



3. 黄枠のチェックボックスを全てチェックして生成ボタンを押下



※自動生成するメソッドの挿入位置に関しては自由ですので、ドロップダウンリストに表示されたままの状態でもOKです。

4. 自動生成完了後の内容

黄枠が自動生成されたメソッドになります。

InstanceOfSato.java Main.java

InstanceSample ▶ src ▶ instance ▶ InstanceOfSato ▶

```java
package instance;

import java.util.Objects;

public class InstanceOfSato {

    private String firstName;

    public InstanceOfSato(String firstName) {
        this.firstName = firstName;
    }

    @Override
    public int hashCode() {
        return Objects.hash(firstName);
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null) {
            return false;
        }
        if (!(obj instanceof InstanceOfSato)) {
            return false;
        }
        InstanceOfSato other = (InstanceOfSato) obj;
        return Objects.equals(firstName, other.firstName);
    }

}
```

サンプル

フィールド変数のハッシュコードを元に、インスタンスの同一性をチェックするクラスです。
（コピーできるようコードとしても残していきます。

```
// ※パッケージはなくても結構です
package instance;

import java.util.Objects;

public class InstanceOfSato {

    private String firstName;

    public InstanceOfSato(String firstName) {
        this.firstName = firstName;
    }

    /* ----- 自動生成部分 ----- */
    @Override
    public int hashCode() {
        return Objects.hash(firstName);
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null) {
            return false;
        }
        if (!(obj instanceof InstanceOfSato)) {
            return false;
        }
        InstanceOfSato other = (InstanceOfSato) obj;
        return Objects.equals(firstName, other.firstName);
    }
    /* ----- 自動生成部分 ----- */

}
```

サンプル: Main

上記クラスのインスタンスを使用するために、最低限のmainメソッドを実行できるクラスを用意します。

```
public class Main {

    public static void main(String[] args) {

        // 複数のインスタンスを生成
        InstanceOfSato sato1 = new InstanceOfSato("一郎");
        InstanceOfSato sato2 = new InstanceOfSato("次郎");
        InstanceOfSato sato3 = new InstanceOfSato("一郎");

        System.out.println("--- インスタンスごとのハッシュコード ---");
        System.out.println("sato1のハッシュコード: " + sato1.hashCode());
        System.out.println("sato2のハッシュコード: " + sato2.hashCode());
        System.out.println("sato3のハッシュコード: " + sato3.hashCode());

        System.out.println("--- 「sato1 と sato2」のインスタンスが同一かのチェック ---");
        System.out.println("参照先インスタンスの比較    : " + (sato1 == sato2));
        System.out.println("参照先インスタンスの値の比較: " + (sato1.equals(sato2)));

        System.out.println("--- 「sato1 と sato3」のインスタンスが同一かのチェック ---");
        System.out.println("参照先インスタンスの比較    : " + (sato1 == sato3));
        System.out.println("参照先インスタンスの値の比較: " + (sato1.equals(sato3)));
    }
}
```

【出力結果】

```
--- インスタンスごとのハッシュコード ---
sato1のハッシュコード: 680786
sato2のハッシュコード: 784692
sato3のハッシュコード: 680786

--- 「sato1 と sato2」のインスタンスが同一かのチェック ---
参照先インスタンスの比較    : false
参照先インスタンスの値の比較: false

--- 「sato1 と sato3」のインスタンスが同一かのチェック ---
参照先インスタンスの比較    : false
参照先インスタンスの値の比較: true
```

解説

・ `hashCode()` **にて対象の値のハッシュコードを取得する**

Javaにおけるハッシュコードは、簡単に言えば、 **対象の値を整数値に変換したデータ** になります。

（※「パスワード等に用いられるハッシュ」とは少々異なるため、ここではインスタンスに対するハッシュの認識でいてください。

`hashCode()` によって、インスタンスを生成後の `sato1, 2, 3` のチェックしてみたところ、
`sato1, 3` は同一のハッシュコード（ `680786` ）であることが見て取れましたね。

ここで呼び出した `hashCode()` は、InstanceOfSatoクラスの自動生成されたメソッドを指しています。
実際の動きとしては、 InstanceOfSatoクラスのフィールド変数である`firstname` が `hashCode()` の評価対象となり、
`firstname`のハッシュコードを計算して戻り値として返しています。

`sato1, 2, 3` を比較したところで、異なるインスタンスを生成していることは明白であるため、
フィールド変数である`firstName` を比較の軸として利用 しているわけです。

・ `equals()` **メソッドにてインスタンスが等価であるかのチェックを実施**
`equals()` メソッドも何度か目にしているかと思いますが、Stringクラスの文字列を比較する際に `equals()` を使用する理由はまさにこういうことなのです。
`String` は元々 `Object（参照型のクラス）` を継承しているため、単なる値の比較で用いる `==` は上手く機能しない場合があります。
（ `==` を用いる対象としては、値型である`int` や`boolean` などが適切ですね。

長くなりましたが、上記を踏まえてstaticが「ただひとつである」ということを再度見ていきましょう。

> **サンプル: staticな変数の値の動き**

【InstanceOfSato】へ以下の内容を追記します。

```
public class InstanceOfSato {

    /** 変数（フィールド変数） */
    private int id = 0;
    /** static 変数（クラス変数） */
    private static int staticVarId = 0;

    /**
     * IDをインクリメント
     */
    public void incrementId() {
        this.id++;
        System.out.println("id: var          = " + this.id);
        InstanceOfSato.staticVarId++;
        System.out.println("id: static var   = " + InstanceOfSato.staticVarId);
    }

    ...
}
```

【Main】には以下を追記します。

```
public class Main {

    public static void main(String[] args) {

        ...
        System.out.println("--- incrementId() ---");
        System.out.println("sato1:");
        sato1.incrementId();
        System.out.println("sato2:");
        sato2.incrementId();
        System.out.println("sato3:");
        sato3.incrementId();
    }
}
```

【出力結果】

```
--- incrementId() ---
sato1:
id: var          = 1
id: static var   = 1
sato2:
id: var          = 1
id: static var   = 2
sato3:
id: var          = 1
id: static var   = 3
```

> **解説**

・非static と static の性質の差
動的に使用した場合には、その差がハッキリと見受けられます！

今回は生成された `sato1, 2, 3` のインスタンス変数より、
フィールド変数とクラス変数に対し、それぞれをインクリメントした後の値を出力しています。

- 非staticなフィールド変数である `id`
  → インスタンスを生成する度に初期化されるため、インクリメント後の出力値は常に `1`

- staticなクラス変数 `id`
  → **InstanceOfSatoクラスにおいて一意である**
  → `incrementId()` **メソッドの呼び出し元（sato1, 2, 3など）が異なっていても、同じ `id` が使用されている**
  → 同一の値を使用するため、インクリメント後の出力値は、 `incrementId()` メソッドを使用する回数に比例する

出力結果の動きや解説を見て、
「ただひとつ」ってそういうことなのか〜！
と理解の助けになれば幸いです！

> **注意点**

1. staticメソッド内で扱う変数は、全てクラス変数かローカル変数（メソッド内で宣言した変数）でなければなりません。
2. staticメソッドは、サブクラスに継承されず、オーバーライドすることもできません。

どの内容にも言えることですが、特に `final` や `static` に関してはなんとなくで使用せず、
しっかりと使用するシチュエーションを見極めて使えるようになると、無駄の無い効率的な開発が行えるようになります！

本章に関しては課題はありませんが、
サンプルコードを参考に実際に自分の手でコードを記述し、実行し、結果を確認してみましょう！

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) 15:48