3-6.オーバーロード・継承・オーバーライド

提出する 評定を受ける

■オーバーロード・継承・オーバーライド

はじめに

この章では、Javaの便利な機能について学びましょう。現場で必ず使う機能になるかと思います。
基礎となる処理を呼び出したり、それに上書きできたりする機能です。
まずは触れてみて、理解を深めていきましょう！

Step1: オーバーロード

メソッド名が同じ場合、 **引数の型** や **引数の数** によって、メソッド又はコンストラクタなどの処理を別々のものとして扱えるという概念です。
メソッドなどの呼び出し時には、定義されている引数の型や数を合わせることで、
特定の引数を持ったメソッドまたはコンストラクタ の処理が行われます。

こんなややこしいものが必要な理由は一つ、 **設計が便利だから** です！

例えば、処理はほとんど変わらないのに引数が違うだけで、メソッド名が無限に増えていくと
SEもPGもメソッド名を見て、「あれ？これってどんな機能のメソッドだっけ？それぞれ違うやつ？」という風に混乱してしまいます。。

そこで、メソッド名を統一し、引数だけ変更すれば、
「渡す引数も出力結果も違うけど、やりたい事（軸）は変わらないんだな」と把握しやすくなります。

オーバーロードを使用することで、利用者の入口（メソッド名）は同じでも、
引数の型や数によって出口（戻り値の有無）を変更したり、中身（処理）を少し変えたりすることで、
最終的な結果にバリエーションを持たせることが可能になります！

オーバーロード: メソッド（Calc クラス）

```
public class Calc {

    // 引数の設定が1つ。引数1つのplusメソッド実行文で呼び出される。
    public int plus(int a) {
        return a + 1;
    }

    // 引数の設定が2つ。引数2つのplusメソッド実行文で呼び出される。
    public int plus(int a, int b) {
        return a + b;
    }

    // 引数の設定が3つ。引数3つのplusメソッド実行文で呼び出される。
    public int plus(int a, int b, int c) {
        return a + b + c;
    }
}
```

オーバーロード: コンストラクタ（Sum クラス）

?

```
public class Sum {

    private int a;
    private int b;
    private int c;

    public Sum(int a) {
        this.constructor(a, 0, 0);
    }

    public Sum(int a, int b) {
        this.constructor(a, b, 0);
    }

    public Sum(int a, int b, int c) {
        this.constructor(a, b, c);
    }

    private void constructor(int a, int b, int c) {
        this.a = a;
        this.b = b;
        this.c = c;
        System.out.println("abc = " + (this.a + this.b + this.c));
    }
}
```

補足

・デフォルトコンストラクタ
通常コンストラクタは、そのクラス内に何も記述しなければ **デフォルトコンストラクタ** が用意されます。
（※コード上にはないが、実処理（ `new クラス名` ）のタイミングで空のコンストラクタ（ `public Sum() {}` ）を呼び出している状態となります。

・オーバーロードを活かしたコンストラクタ
冗長な記述を減らし、可読性の高い初期化処理の実現に役立ちます。

しかし、以下のような場合はコンパイル時点で **エラー（再帰的コンストラクタ）** を吐いてしまい、
思ったような処理を実現することができません。

【サンプル: コンストラクタ内で自クラスのコンストラクタを呼び出す】

Main クラス

```
public class Main {

    public static void main(String[] args) {

        // [ERROR]①のコンストラクタを実行,②でも③でも実行するとエラー
        Sum s = new Sum(5);
    }
}

class Sum {
    // ①
    public Sum(int a) {
        this(a, 0, 0);
    }

    // ②
    public Sum(int a, int b) {
        this(a, b, 0);
    }

    // ③
    public Sum(int a, int b, int c) {
        this(a, b, c); // ←自分自身のコンストラクタを呼び出す（処理が終わらないのでエラー）
    }
}
```

エラーの原因は、 **自分自身のコンストラクタ（③）を呼び出すことによる処理の終結が無いこと（ループ）** です！

`this()`は自分のクラスのコンストラクタを呼び出す書き方です。
サンプルコードでは、３つのコンストラクタをオーバーロードしているので、
`this(a, 0, 0) // ③`のように引数を３つ渡すと、引数が３つ定義されているコンストラクタが呼び出されます。

どんな状況かというと以下みたいな感じです。

```
// ①のコンストラクタを実行
①→③→③→...（③を呼び出し続ける）

// ②のコンストラクタを実行
②→③→③→...（③を呼び出し続ける）

// ③のコンストラクタを実行
③→③→③→...（自分自身を呼び出し続ける）
```

これでは、永遠にコンストラクタの処理が終わりませんよね？
そのため、コンパイル時にはエラー：コンストラクタの呼出しが再帰的ですという形でエラーが発生することになります。

以上のような再帰的コンストラクタのループにハマらないために、
３つのコンストラクタ内で共通で使用できるメソッド（ `constructor()` ）を用意し、
一つの口で初期化できるようにしている訳です！

※「再帰」というのは、 **自分自身を呼び出すこと** を意味します。
ここで言う自分自身というのは、処理を記述しているメソッドやコンストラクタ自身のことを指します。

■参考URL（外部リンク）：
thisメソッドで自分自身のコンストラクタを呼ぶ書き方

オーバーロードについて（例題）

下記に示すプログラムを作成し、実行しなさい。
※課題の提出の必要はありません。

Sum クラス

```
public class Sum {

    public int plus(int a) {
        return a + 1;
    }

    public int plus(int a, int b) {
        return a + b;
    }

    public int plus(int a, int b, int c) {
        return a + b + c;
    }
}
```

SumMain クラス

```
public class SumMain {

    public static void main(String[] args) {
        Sum sum = new Sum();
        int sumValue = sum.plus(1, 2, 3);
        System.out.println(sumValue);
    }
}
```

Step2: 継承

継承 とは、任意のクラスのデータや機能を **他のクラスで受け継ぐこと** です。
任意の **元のクラス** を **スーパークラス** 、 **受け継ぐクラス** を **サブクラス** と呼びます。

継承することによって **スーパークラスの持ち物であるフィールドやメソッド** を **サブクラス** で使用することが可能となります。
但し、スーパークラスのフィールドやメソッドのアクセス修飾子が`private` の場合、
継承してもサブクラスで使用することはできませんので注意してください。
`private` 以外の場合、サブクラスで使用することが可能です。

Step3: 継承の方法

継承するには、 `extends` を記述します。 extendsとは、 **拡張する** という意味です。

```
public class サブクラス名 extends スーパークラス名 {
    ...
```

クラスの宣言文は、 **「クラスｘｘｘは、スーパークラスＳＳＳを拡張する」** といった感じで読むことができます。
但し、 継承できるクラスは１つのみ です。 **複数のクラスを同時に継承したりすることはできません。**

サンプルコード（継承のイメージ）

```
class スーパークラス名 {

    // フィールド変数A
    protected int A;

    // メソッドXXX
    protected int XXX(){
        // 処理
    }

}

public class サブクラス名 extends スーパークラス名 {

    /**
     *  見えないけれど、スーパークラス内の
     *  フィールド変数Aとメソッドxxxがこのクラス内で定義されているイメージ
     */

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

継承のメリット

・同じような機能をもつ重複したコードを書かないようになるので、コードの再利用性が高くなる
・メソッドを追加したり、オーバライド（上書き）することができるので自由に拡張する

重複したコードを書くと勘違いやミスが増え、
思わぬバグやトラブルを起こしてしまう可能性があり、メンテンスが難しくなります。
メンテナンスが難しくなることによって、「機能の追加」や「改善」が難しくなり拡張性が下がってしまいます。
効率よく少ないコードでプログラミングしないと、開発がどんどん難しくなり、コードを書くプログラマーも
苦しくなってきてしまいます。

例えば・・・

継承を用いる場合は、「骨組み」となるクラスを１つ作り、その１つのクラスを継承するのが一般的です。
具体例では、 **「RPGのキャラクター」** があります。

初期のキャラクターをスーパークラスとして、継承して戦士クラスとか魔法使いクラス などを作るイメージです。
そうすることで、戦士であれば剣を使う部分だけ、魔法使いであれば魔法を使う部分だけ、
つまり必要最低限のコードを、追加するだけでプログラミングができてしまうのです。

Step4: オーバーライド

オーバーライドとは、継承関係において **スーパークラスのメソッドの処理をサブクラスの同名のメソッドで上書きすること** です。
継承関係以外では、この現象は起こりません が、継承するサブクラスを作成するときは、
多少スーパークラスのことを知らないと意図せずオーバーライドしてしまうことがありますので注意してください。

サンプル: オーバライド その１

Greet クラス

```
public class Greet {

    public void morning() {
        System.out.println("Good Morning");
    }
}
```

GreetInJapaneseクラス

```
// extends Greet で Greet コントローラを継承
public class GreetInJapanese extends Greet {

    // morningメソッドをオーバライド（上書き）
    public void morning() {
        System.out.println("おはようございます");
    }
}
```

GreetMain クラス

```
public class GreetMain {

    public static void main(String[] args) {
        GreetInJapanese jp = new GreetInJapanese();
        jp.morning();
    }
}
```

【出力結果】

```
おはようございます
```

解説

・`morning` **メソッドを継承先クラスである GreetInJapanese で上書き**
継承先のクラスにて **スーパークラスの持つメソッドの処理** を上書いています。

`Good Morning` を表示したければ、下記のようにします。

サンプル: オーバライド その２

```
public class GreetInEnglish extends Greet {

    public void morning() {
        // スーパクラス（Greetクラス）の morning メソッドの呼び出し
        super.morning();
    }
}
```

解説

・`super` **キーワードを指定しての** `morning()` **メソッドの呼び出し**
こうすることでスーパークラスであるGreetクラスの`morning()` メソッドが呼び出されます。
オーバーライドの利用例として、 **特定のサブクラスにおいて、当該メソッドの処理内容を修正・変更する** という利用例があります。

【例: 既存クラスのメソッドの処理内容を変更したい場面】

このとき、そのスーパークラスのメソッドの処理内容を直接修正することが可能だったとします。
（※今回の場合は、 `morning()` の `System.out.println("Good Morning");` 部分を指します。

しかし、他のクラスでもそのクラスのメソッドを使っている可能性は否めません。
そのため、そのメソッドを修正したことによって他のクラスに影響が出る可能性があります。

上記サンプルの **Greet** で言えば、
GreetInJapanese 以外のオーバーライド先の処理内容として、
良い朝ですね！ などと出力するようにしている場合です。

そのような場合は、スーパークラスのメソッドを直接修正せず、
クラスを継承したサブクラスを用意して当該メソッドをオーバーライドし、
処理内容を修正（上書き）することで、
修正内容をサブクラスのみに適用することができます！

つまり、

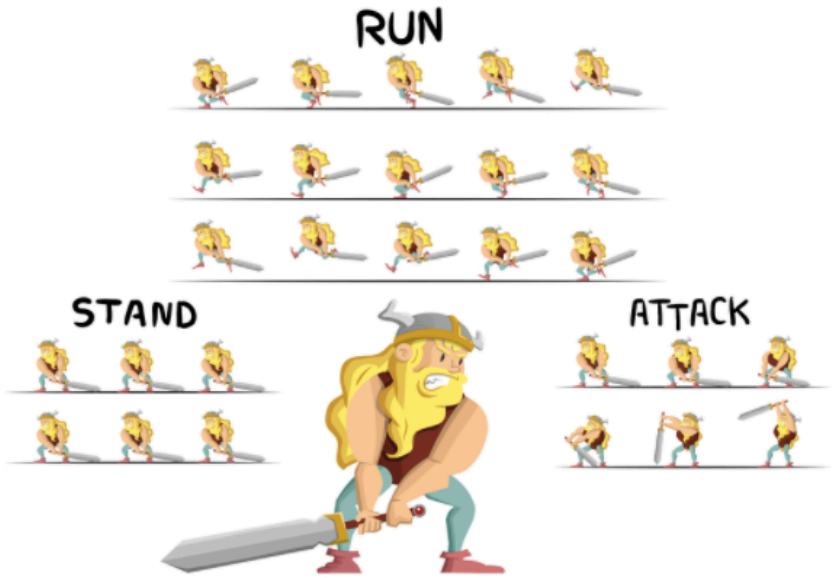
- プログラムの影響範囲を限定する
- コードを再利用する

という2点に重きを置いた実装方法になります。

大本は変えず、使用先のみに修正や変更を加えよう！ということです。

> **RPGの例で考えよう**

具体的な使い方について触れていきます！



継承したサブクラスでは、スーパークラスのメソッドを使うことができます。
新しく追加して機能を追加することもできますが、実はスーパークラスのメソッドを「上書き」することもできます。

スーパクラスのメソッドをサブクラスで上書きすることを「オーバーライド」といいます。
このオーバーライドが継承を使う上でもっとも重要な機能 となります。

具体例としては、

・RPGのキャラクターが行う特定のアクションを上書きする場合
オーバーライドはRPGのキャラクターが行う特定のアクションを上書きする場合によく使います。

攻撃する処理を書いたメソッド（ `attack()` ）をオーバーライドし、
「忍者は通常攻撃で一定確率で即死攻撃」
「白魔道士は攻撃で回復を行う」
といった具合で上書きするようなイメージです。

```
// 忍者 クラスの場合
public void attack() {
    // 一定確率で即死 にする処理を実装
}

// 白魔道士 クラスの場合
public void attack() {
    // （攻撃だけど）回復を行う
}
```

これも上記のように「一定確率で即死」「回復する」などの違う部分だけをプログラミングするだけで済みますし、
同じメソッドを使っても実行するクラスによって違う動作を行うようにできます。
こちらの方が効率よくプログラミングすることができますね！

課題

インポートした3-6のフォルダの中にプロジェクトがありますので、
指示通りにコーディングして、ファイルを提出して下さい。

評定概要

学生から秘匿	No
参加者	75
提出	50
要評定	1