オブジェクト指向と
ゲームプログラミング

○　C++編 － 第７回　ポリモーフィズム　○

ポリモーフィズム

オブジェクト指向の重要な概念のひとつに、ポリモーフィズム(polymorphism：多態性または多相性)があります。ポリモーフィズムは、オブジェクト指向プログラミングで必ず行わなければならないものではありませんが、プログラミングの効率化に効果があります。

ポリモーフィズムは、同じ名前を持った動作が状況に合わせて異なる動作をすることをいいますが、ソースコード上でのそれは、「同じ名前で異なる機能を持ったメソッドを複数定義する」ことをいいます。Javaでは、メソッドのオーバーロードまたはオーバーライドにより、これを実現しています。

オーバーロード

同じ名前のメソッドを複数定義することができます。これをメソッドのオーバーロード(overloading：多重定義)と呼びます。オーバーロードする際には、各メソッドの引数の型または個数が異なるようにしなければなりません。戻り値の型のみが異なる場合は、文法エラーとなり、オーバーロードできません。

```
class Test {
    int      n;
    float    f;
    double   d;

    public void setValue(int    val) { n = val; }
    public void setValue(float  val) { f = val; }
    public void setValue(double val) { d = val; }

    public void setValue(int n, float f,  double d)
    { this.n = n; this.f = f; this.d = d; }
}
```

上のクラスでは、SetValue関数が４つあります。

```
void setValue(int    val)
void setValue(float  val)
void setValue(double val)
void setValue(int n, float f, double d)
```

このように、同じ名前のメソッドを同じクラス内に定義することができます。この例では戻り値の型が同じですが、異なっても構いません。オーバーロードされたメソッドは、引数の型や個数から、それに応じたものが呼び出されます。

```
Test   test = new Test();
test.setValue(100);             // int型      のsetValue関数が呼び出される
test.setValue(89.3);            // double型   のsetValue関数が呼び出される
test.setValue(0, 1.0f, 8.93);   // 引数が３つのsetValue関数が呼び出される
```

オブジェクト指向では、ひとつの動作がその状況に応じて別々の働きを持つことをポリモーフィズムと呼びますが、このオーバーロードも、同じ名前のメソッドが引数の型や数に合わせた動作をすることから、そのひとつだと言えます。

オーバーライド

あるクラスを継承した場合、スーパークラスで定義されたメソッドの機能がサブクラスの目的に合わない場合があります。このような場合、サブクラスでそのメソッドを再定義し、機能を上書き変更することができます。これをメソッドのオーバーライド(overriding：再定義)と呼びます。何度も継承を行って階層的な構造になっているクラスのメソッドをオーバーライドした場合は、最後にオーバーライドしたものが優先されます(フィールドもオーバーライドすることができます)。

メソッドをオーバーライドするには、「メソッド名」「引数の型と数」「戻り値の型」のすべてがスーパークラスと同じである必要があります(アクセス修飾子は、より緩い方へ変更することができます)。

メソッド名がスーパークラスと同じでも、引数の型や数が異なるとオーバーロードとなります。戻り値の型だけが異なる場合は、オーバーライドにもオーバーロードにもならず、文法エラーとなります。

```
class Character {
    public void move() {}
}

class Enemy extends Character {
    public void move() {}       // moveメソッドのオーバーライド
}
```

サブクラス内で、オーバーライドしたスーパークラスのメンバを呼び出すことができます。オーバーライドしたメンバを呼び出したい場合は、superを使います。たとえば、上のEnemyクラス内で、CharacterクラスのMoveメソッドを呼び出したい場合は、以下のように記述します。

```
super.move();   // Characterクラスのmoveメソッドを呼び出す
```

階層的に複数回継承している場合、最後にオーバーライドしたメンバが優先されます。つまり、2つ以上さかのぼってスーパークラスのメンバにアクセスする場合は、1つ上のスーパークラスでオーバーライドされていない場合のみアクセスできるということです。

スーパークラスのメソッドに、サブクラス共通の処理を記述しておくと、サブクラスではそのメソッドをオーバーライドし、差分だけを作成すればよくなります。

final

クラスをこれ以上継承させたくない場合、クラス宣言にキーワードfinalをつければ継承を禁止することができます。

```
// 継承できないクラス
final class Character {
    …
}
```

finalなクラスにすると、少しだけ動作速度が向上します。また、メソッドにfinalをつけると、オーバーライドできないメソッドとなります。

```
// オーバーライドできないメソッド
public final void move() {}
```

アップキャストとダウンキャスト

スーパークラスの参照型変数に、サブクラスの参照を代入することができます。これをアップキャストと呼びます。たとえば、

```
Character   chara = new Enemy();    // サブクラスの参照をスーパークラスの参照に代入
```

ということができるのです。Javaでは、ポリモーフィズムや動的オブジェクトを表現するのに、アップキャストが重要な役割を担っています。

この逆の動作、サブクラスの参照型変数にスーパークラスの参照を代入することをダウンキャストと呼びます。ただし、アップキャストした参照をダウンキャストして元のクラスの参照に戻すような場合にしか行えません。なぜかというと、「スーパークラスのオブジェクトをサブクラスのオブジェクトとして使う」ことを意味するからです。スーパークラスのオブジェクトは、サブクラスのオブジェクトに比べ、メンバの数が足りない可能性があります。もし、Enemyクラスの参照にCharacterクラスの参照が代入可能だとしたら、Enemyクラス独自のメンバの部分が不定になってしまいます。CharacterクラスにはEnemyクラスで拡張された機能がないからです。

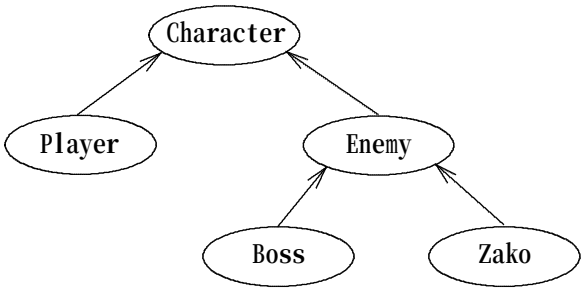
アップキャストを行った場合、サブクラスで追加したメンバを呼び出すことができなくなります。スーパークラスの参照は、あくまでもスーパークラスとみなされるからです。

```
chara.Enemyクラス独自のメンバ;
```

このような参照はできないのです。どうしても参照したい場合は、ダウンキャストしてサブクラスにします。キャストできない場合は、例外が発生します。

```
(Enemy)chara.Enemyクラス独自のメンバ; // ダウンキャストしてEnemyクラスのメンバにアクセス
```

アップキャストとダウンキャストという名前の由来は、クラス階層図での方向から来ています。クラス階層図とは、クラスの継承関係を表すもので、次のようなものです。



クラス階層図は、このようなツリーで記述されます。下のクラスは、上のクラスから派生していることを示しています。たとえば、「Player」は「Character」、「Zako」は「Enemy」、「Enemy」は「Character」を継承していることが読みとれます。

アップキャストは、上のクラスに向ってなされ、ダウンキャストは、下のクラスに向ってなされます。このように、アップキャストとダウンキャストは、キャストの方向を示しているのです。

抽象メソッド

たとえば、キャラクターの基本的な動作と属性を管理するCharacterクラスをスーパークラスに、サブクラスとしてプレイヤーを表すPlayerクラス、ザコ敵を表すZakoクラス、ボス敵を表すBossクラスを作成し、各クラスには移動処理を行うmoveメソッドがあるとします。

ポリモーフィズムを実現するため、サブクラスでは、スーパークラスのmoveメソッドをオーバーライドします。しかし、プレイヤーはキーボードによって移動、ザコは決められたルートを移動、ボスはプレイヤーや自身の状態を判断して移動パターンを変える、というように、移動処理は各クラスによって大きく異なります。

このような場合、スーパークラスではサブクラスで共通する処理というものを記述することはできません。これは、Characterクラスでは、moveメソッド本体の実装ができないということになります。このメソッドは、サブクラスで実装されるべきなのです。
実装できないので「何も処理をしない」というメソッドにすることもできますが、このような場合、Javaでは、メソッドを抽象メソッドというものにすることができます。抽象メソッドとは、メソッドはあるものの実体がない－実装はサブクラスでする、というものです。作り方はとても簡単で、キーワードabstractをつけます。
たとえば、Characterクラスのmoveメソッドを抽象メソッドにするには、以下のようにします。

```
public abstract void move();     // moveメソッドを抽象メソッドにする
```

抽象メソッドは、宣言のみ可能でメソッド本体の実装を行うことはできません。また、抽象メソッドは、実装されるまでどのような手段を使っても呼び出すことはできません。

抽象クラス

抽象メソッドを１つでも持つクラスは、キーワードabstractをクラス宣言につけ、抽象クラスとして宣言しなければなりません。抽象クラスであっても、通常のクラスのようにメソッドやフィールドを定義することができます。

```
// 抽象クラスの宣言
public abstract class Character {
    public abstract void move();
}
```

抽象クラスは、抽象メソッドを必ず持っている、つまりメソッドの実装がないメンバを持っているため、インスタンスを生成することはできません(抽象クラスの参照型を格納する変数は宣言できます)。
抽象クラスは、継承によって派生され、そこでメソッドの実体が定義されることを前提にしたクラスを作成したり、ポリモーフィズムのためのインタフェースを提供したりする場合に使用します。

練習問題

1　以下の文章の内容が「カプセル化」の場合はA，「オーバーロード」の場合はB，「オーバーライド」の場合はCを付けましょう。

(1)スーパークラスの参照からサブクラスのメソッドを呼び出す。
(2)フィールドに不適切な値が書き込まれるのを防ぐ。
(3)クラス内部の変更が、クラスを使う側に影響を与えないようにする。
(4)スーパークラスのメソッドの処理内容をサブクラスで上書き変更する。
(5)ひとつのクラスに複数のコンストラクタを定義する。
(6)ひとつのクラスに同じ名前のメソッドを複数定義する。

2　以下の文章の内容が正しい場合には○、間違っている場合には×を付けましょう。

(1)抽象メソッドとは、メソッドの処理内容だけを記述したものである。
(2)抽象メソッドの定義では、引数や戻り値を指定しない。
(3)抽象メソッドを1つ以上持つクラスを抽象クラスと呼ぶ。
(4)抽象メソッドは、それを継承したクラスで必ずオーバーライドしなければならない。
(5)抽象メソッドは、それを継承したクラスで必ずオーバーロードしなければならない。

3　以下のプログラムを実行し、ポリモーフィズムの働きを確認しましょう。

```
// キャラクタークラス
abstract class Character {
    abstract void move();
    void draw() { System.out.println("キャラクター描画"); }
}

// プレイヤークラス
class Player extends Character {
    void move() { System.out.println("プレイヤー移動"); }
    void draw() { System.out.println("プレイヤー描画"); }
}

// 敵クラス
class Enemy extends Character {
    void move() { System.out.println("敵移動"); }
}

// ボスクラス
class Boss extends Enemy {
    void move() { System.out.println("ボス移動"); }
    void draw() {
        System.out.println("ボス描画");
        super.draw();
        super.draw();
    }
}

public class AppMain {

    // メインメソッド
    public static void main(String[] args) {
        Character[]   chara = new Character[8];

        // キャラクター生成
        chara[0] = new Player();
        chara[1] = new Enemy();
        chara[2] = new Enemy();
        chara[3] = new Enemy();
        chara[4] = new Enemy();
```

```
        chara[5] = new Enemy();
        chara[6] = new Enemy();
        chara[7] = new Boss();

        // 移動
        System.out.println("--- 移動処理開始 ---");
        for(i = 0; i < chara.length; i++)
            chara[i].move();

        System.out.println("");

        // 描画
        System.out.println("*** 描画処理開始 ***");
        for(i = 0; i < chara.length; i++)
            chara[i].draw();
    }

}
```