

第5章 オブジェクト指向

これまで定義したクラスは、すべて JApplet クラスを継承したものだった。この章ではオブジェクト指向の概念をより良く理解するために、簡単なクラスを一から設計することにする。この章の例は規模が小さ過ぎて、再利用などオブジェクト指向のありがたみがわかりにくいかもしれない。オブジェクト指向は規模の大きなソフトウェアでこそ生きる技術であり、この章の例はおもちゃの例 (toy example) に過ぎないことを心に留めておいて欲しい。

5.1 クラス

まず、もっとも簡単な 2 次元座標を表すためのクラスから始める。クラスの定義は、今までも行ってきたが、今回は一から定義するので extends 以下がない。

詳細: 正確にいうとすべてのクラス型の暗黙のスーパークラスとなる Object というクラスがあり、extends 以下がない場合は、... extends Object と書くのと同じになる。

ファイル Point.java (バージョン 1)

```
public class Point {  
    // フィールド (インスタンス変数)  
    public int x;  
    public int y;  
}
```

クラスは基本的には、いくつかのデータ (変数) をひとつのまとまりとして扱えるように部品化したものである。配列は同種のデータをまとめたものであるが、クラスは異種のデータをまとめることができる。

上の例では Point という名前のクラスを定義している。x と y は、このクラスの _____ である。(_____, _____ という呼び方も用いる。) この例では、たまたまフィールドの型がすべて同じであるが、もちろんフィールドの型はバラバラで構わない。

5.2 クラスの使用

Point などのクラスの名前は、int などの Java にもともとある型名と同じように使うことができる。例えば p という変数が Point クラスに属することを宣言するためには、

```
Point p;
```

のようにすれば良い。このような変数を初期化するためには ____ というキーワードと、クラス名を用いて、

```
p = new Point();
```

と書く。このとき、新しい Point クラスの _____ (instance, 具体例という意味) が生成されて、p という変数に代入される。Point クラスのインスタンスは、今の定義の場合、int を 2 つ持ち、自分が Point クラスに属するという情報も持つデータである。

実際の使用例は次のような形になる。

```
Point p = new Point();
p.x = 1; p.y = 2;
System.out.println("(" + p.x + ", " + p.y + ")");
```

オブジェクトのフィールドには「_」(_____) 演算子を用いてアクセスする。 . の前にオブジェクト、後にフィールド名を書く。

5.3 メソッド

これまでのクラスの使用法は C の構造体にほぼ相当する。このままではオブジェクト指向の一步手前である。実際にはクラスはもっとパワフルな概念であり、オブジェクト指向を使いこなすには、その差の部分を知る必要がある。

まず大事なことは、クラスの中には、関数 (_____ , _____) を定義することができるということである。

ファイル Point.java (バージョン 2)

```
public class Point {
// フィールド (メンバ変数)
public int x;
public int y;

// メソッド (メンバ関数)
public void move(int dx, int dy) {
    x += dx;
    y += dy;
}

public void print() {
    System.out.printf("(%d, %d)", x, y);
}

public void moveAndPrint(int dx, int dy) {
    print(); move(dx, dy); print();
}

// コンストラクタ
public Point(int x0, int y0) {
    x = x0; y = y0;
}
}
```

move と print、moveAndPrint はこのクラスのメソッドである。メソッドの中では、同じオブジェクトの他のフィールド (例えば x, y) やメソッド (例えば move や print) を . なしで参照することができる。

さらに各クラスはクラスと同じ名前の特別なメソッド(_____)を持つことができる。上の例では Point クラスに int 型の引数 2 つを取るコンストラクタを定義している。

詳細: プログラマがコンストラクタを 1 つも明示的に定義しないときは、すべてのフィールドに既定値を割り当て、他に何もしない引数なしのコンストラクタが自動的に用意される。

他のメソッドの場合と異なり、コンストラクタの定義のとき上の例のように戻り値の型は指定しない。コンストラクタを使うと、Point 型の変数を次のように初期化することができる。

```
p = new Point(1, 2);
```

これで、Point クラスのインスタンスが生成され、フィールド x が 1、y が 2 に初期化される。

オブジェクトのメソッドにもやはり「.」演算子を用いてアクセスする。次に示す PointTest は Point クラスをテストするための別のクラスであり、main メソッドのみからなる。

ファイル PointTest.java

```
public class PointTest {
    public static void main(String args[]) {
        Point p = new Point(10, 20);
        p.move(1, -1);
        p.print();
        System.out.println("<br/>");
    }
}
```

static はメソッドがクラスメソッドであること(他のスタティックでないフィールドに依存しないこと)を表す修飾子である。クラスメソッドは、C や C++ の通常の(メソッドではない)関数と同じ感覚で使うことができる。

PointTest はフィールドが一つもない、変なクラスであるが、Java ではすべてのメソッドはクラスの中に宣言しなければならないため、このようなクラスも必要になる。

詳細: PointTest.java と Point.java を同じディレクトリに置いておくと、PointTest.java をコンパイルすれば、javac が自動的に依存関係を見つけ出して、Point.java もコンパイルする。

5.4 継承

Point にさらに色の属性を持たせて ColorPoint というクラスを定義する。このとき既存の Point クラスを利用して、増えたフィールドやメソッドだけを定義する。このことを Point クラスを _____ (_____) するという。Point クラスは ColorPoint クラスの _____ である、という。逆に ColorPoint クラスは Point クラスの _____ である。

継承するときは、クラスを定義するときに「extends」の後にスーパークラスの名前を書く。

ファイル ColorPoint.java (バージョン1)

```
public class ColorPoint extends Point {
    public String color;

    public ColorPoint(int x, int y, String c) {
        super(x, y); /* 1 */
        color = c;
    }

    @Override
    public void print() {
        System.out.printf("<font color='%s'>", color); // 色の指定
        System.out.printf("(%d, %d)", x, y); // 2 /*
        // super.print(); でも可
        System.out.print("</font>"); // 色を戻す
    }
}
```

ColorPoint では、新しいフィールド color と再定義するメソッド print()、それとコンストラクタのみを定義している。(このように継承を用いると既存のクラスを利用して差だけを記述すれば良い。これまでアプレットを簡単に作成できたのはスーパークラスの JApplet に必要な処理がほとんどすべて記述されていたからである。) コンストラクタの中の super(x, y) という式 (/* 1 */) はスーパークラス (Point) のコンストラクタを呼び出す。super はスーパークラスを表すキーワードである。

詳細: 継承したクラスのコンストラクタでは、最初の文でスーパークラスのコンストラクタを呼び出さなければいけない。(ただし、スーパークラスが引数なしのコンストラクタを持っていて、スーパークラスのコンストラクタ呼び出しがない場合は、自動的に追加される。)

色は、文字列で表すことにする。print() の中では、HTML のタグを用いて色を変更している。このプログラムの出力結果を HTML ブラウザで表示すると、実際にその色で文字が表示される。

また、ColorPoint の print() の 2 行目 (/* 2 */) は Point の print() と同じなので、単に super.print(); と書くこともできる。この場合、super はスーパークラスを指す。

下のプログラムの main メソッドの 1 行目 (/* 3 */) で ColorPoint クラスのインスタンスが生成される。フィールド x が 10、y が 20、color が "green" にそれぞれ初期化される。また、インスタンスは自分が ColorPoint クラスに属するという情報も持つ

Point からフィールド x と y とメソッド move は継承されるので、引き続き利用することができる (/* 4 */)。

ファイル PointTest.java (バージョン2)

```
public static void main(String args[]) {
    ColorPoint cp = new ColorPoint(10, 20, "green"); /* 3 */
    cp.move(1, -1); // 4 /*
    cp.print();
    System.out.println("<br/>");
}
```

このプログラムでは、"(11, 19)
" と表示されるはずである。

5.5 カプセル化

ところで、color フィールドは、"red", "green" など、色を表す文字列専用で、それ以外が設定されると困るので、専用の設定関数を設けて、正当な色を表しているかをチェックしたい。このため2つのメソッド setColor と getColor を ColorPoint に追加する。具体的には、色は"black", "red", "green", "yellow", "blue", "magenta", "cyan", "white"のいずれかの文字で指定することにする。また、color フィールドは、各色に対応する整数値 (int 型) で表すことにする¹。

ファイル ColorPoint.java (バージョン2)

```
public class ColorPoint extends Point {
    public String[] cs = {"black", "red", "green", "yellow",
                        "blue", "magenta", "cyan", "white"};
    public int color;    // 0-黒 1-赤 2-緑 3-黄 4-青 5-紫 6-水 7-白

    @Override
    public void print() {
        System.out.print("<font color='"+getColor()+"'>");    // 色の指定
        System.out.printf("%d, %d", x, y);    // super.print();でも可
        System.out.print("</font>");    // 色を戻す
    }

    public void setColor(String c) {
        int i;
        for (i=0; i<cs.length; i++) {
            if (c.equals(cs[i])) {
                color = i; return;
            }
        }
        // 対応する色がなかったら何もしない。
    }

    public ColorPoint(int x, int y, String c) {
        super(x, y);
        setColor(c);
    }

    public String getColor() {
        return cs[color];
    }
}
```

ところで、せっかく setColor と getColor を定義したのだから、フィールドの color は直接、他のオブジェクトのメソッドやクラスメソッドからは見えないようにして、0~7以外の値を設定できないようにしたい。(つまり、cp.color = 100; のような操作ができないようにしたい。) 同じオブジェクトのメソッドからは見えるが、他のオブジェクトのメソッドやクラスメソッドからは見えないフィールドやメソッドを _____ であるという。逆に他のオブジェクトのメソッド(あるいはクラスメソッド)からでも見えるフィールドやメソッドを _____ であるという。プライベートなフィールドやメソッドを定義するためには、public の代わりに _____ という修飾子を使う。color フィールドをプライベートにするために ColorPoint の定義を次のように書き換える。

¹実際のプログラムでは、このように記憶領域をケチる必要がある場合はほとんどない。ここで、color フィールドを int 型に変えるのは、単なる説明のための方便である。

```
...
private int color;    // ...
...
```

これで `color` はプライベートなフィールドになる。(ついでに `cs` もプライベート (かつスタティック) にしておく。) 他のインスタンスのメソッドで、例えば `cp.color = 100;` のように、このフィールドへの直接操作を行なおうとするとコンパイル時にエラーになる。その他のフィールドやメソッドは `public` という修飾子があるのでパブリックである。

詳細: また、`protected` という修飾子がつく場合も、`private`, `public`, `protected` の、どの指定もない場合もある。後者の場合の意味は `public` に近いが、プログラムをいくつかのファイルに分割した場合には意味が変わってくる。このプリントでは分割コンパイルは扱わないので、これらの場合の説明は割愛する。

このように、クラスを構成するフィールドやメソッドの一部をメソッド以外に非公開にすることを _____ あるいは _____ という。カプセル化を行なっておくと、メソッド以外のプログラムがクラスの実装の詳細に依存していないことが保証できるので、クラスの実装の変更が容易に行なえるようになる。(例えば `ColorPoint` クラスの場合、`color` フィールドは `"black"`, `"red"` などの文字列をそのまま記憶するように変更することも可能である。)

関数・サブルーチンを利用する場合、外部から見た振舞いが同じである限り、内部でどのように実現されていても構わない。例えば、配列の要素を大きさの順に並び替える (ソーティング) 方法はいくつもあり、(性能に違いはあるかもしれないが) 自由に入れ換えることができる。これと同じように、クラスを利用する場合でも、2つのクラスの内部の実現方法が少々異なっても、外部から見た振舞いが同じであれば、それらを入れ換えることができる。カプセル化は、そのためにクラスの内部の実現方法を外部から隠すことを意味する。

問 5.5.1 `ColorPoint` の実装を「`color` フィールドは `"black"`, `"red"` などの文字列をそのまま記憶する」ように変更せよ。

問 5.5.2 `DeepPoint` クラスは、このプリントで定義された `Point` クラスを継承し、新しいフィールド `int depth` を持っている。コンストラクタは `x`, `y`, `depth` フィールドの初期値を引数とする。`print` も再定義されていて、`depth` が 5 の `DeepPoint` は `"((((((11, 19))))))"` のように括弧が 5 重になって出力される。

`DeepPoint` クラスを定義せよ。特に `depth` が 1 ~ 10 の値に制限されるように `setDepth` (および `getDepth`) を定義せよ。`depth` フィールドの値は `setDepth` メソッドのみが変更できるようにすること。(`setDepth` メソッドに 0 以下または 11 以上の値が引数として渡されたときは無視するようにせよ。)

問 5.5.3 `SecretPoint` クラスは、このプリントで定義された `Point` クラスを継承し、2つの新しいフィールド `int a`, `b` を持っている。この2つのフィールドはコンストラクタ内で乱数により初期化される。`print` メソッドも再定義されていて、方程式 $a \cdot x + b \cdot y = 1$ を満たすときだけ、普通に `(1, 2)` のように出力し、方程式を満たさないときは、`(?, ?)` とクエスチョンマークを出力する。`SecretPoint` クラスを定義せよ。ただし、フィールド `a`, `b` は `print` メソッド以外の方法で外部から値が見えないようにせよ。

5.6 動的束縛

次のようなコードを考える。

```
public static void main(String args[]) {
    Point p = new Point(1, 2);
    ColorPoint cp = new ColorPoint(3, 4, "green");
    DeepPoint dp = new DeepPoint(5, 6, 5);
    ...
}
```

Point, ColorPoint, DeepPoint の 3 つのクラスのインスタンスを生成している。

つぎに Point の配列を用意し、3 つのインスタンスのアドレスを代入する。

```
...
Point[] pts = new Point[3];
pts[0] = p; pts[1] = cp; pts[2] = dp;
...
```

ColorPoint と DeepPoint から Point への型変換 (キャスト) が暗黙に行なわれているわけであるが、これはサブクラスからスーパークラスへの型変換 (ワイドニング, widening という) であり、一般的に可能である。

詳細: 一般にサブクラスのオブジェクトをスーパークラスの変数に代入することは無条件に可能である。

ファイル CastTest.java

```
ColorPoint cp = new ColorPoint( ... );
Point p = cp;
p.print();
```

一方、スーパークラスの型を持っている値をサブクラスを期待するコンテキストで使用するためにはキャスト (明示的型変換) が必要である。

ファイル CastTest.java (続き)

```
// p = new Point(3, 4); // これをコメントアウトすると実行時エラー
ColorPoint cp2 = (ColorPoint)p;
cp2.setColor("red");
cp2.print()
```

p が指しているオブジェクトが ColorPoint クラス (あるいはそのサブクラス) のインスタンスでないときは実行時にエラーとなる。

ここで、この配列の各要素に一斉に move メッセージを送る。

```
...
int i;
for (i=0; i<3; i++) {
    pts[i].move(10, 10);
}
...
```

これも当然可能である。move は各クラスで共通なので、同じメソッドが起動される。さらに、一斉に print メッセージを送る。

```

...
for (i=0; i<3; i++) {
    pts[i].print();
    System.out.println("<br/>");
}
...

```

print メソッドは ColorPoint, DeepPoint では上書きされているので、各クラスで異なるメソッドである。この場合、どのメソッドが起動されるのだろうか？

実は、Java では、各オブジェクトの生成時のクラスの print メソッドが起動されて、“ (11, 12) (13, 14) (((((15, 16))))))” のように表示される。

このように、字面（変数の型）によって実行されるコードが決まらずに、変数が参照しているオブジェクトの型によって、呼び出されるメソッドが定まる。通常、実際に変数が参照するオブジェクトの型は実行時までわからないので、このようなメソッドの振舞いを _____ という。

（参考） C++で、上のような Java プログラムを真似て Point, ColorPoint, DeepPoint の各クラスを定義し、

```

...
Point* pts[3];
Point* p = new Point(1, 2);
ColorPoint* cp = new ColorPoint(3, 4, "green");
DeepPoint* dp = new DeepPoint(5, 6, 5);
pts[0] = p; pts[1] = cp; pts[2] = dp;
for (i=0; i<3; i++) {
    pts[i]->print();
    cout << "<br>¥n";
}
...

```

のように書くと、すべて Point クラスの print メソッドが起動されて、“(11, 12) (13, 14) (15, 16)” のように表示される。

この C++ のプログラムを Java のような振舞いにするためには、print メソッドを _____ というものにする必要がある。仮想関数とは、ポインタ（上の例では pts[i]）の型ではなく、ポインタが参照している実際のオブジェクト（上の例では p, cp, dp）の型によって実際に呼び出されるコードが決まるメソッドのことである。Java のメソッドはすべて仮想関数である。

一方、C++ のメンバ関数を仮想関数にするためには virtual というキーワードを宣言の前につける。

```

class Point { // 注: これは C++ のプログラム
public:
    int x, y;
    void move(int dx, int dy);
    virtual void print(void);
};

```

C++ では効率を重視するので、非仮想関数をデフォルトにしているのである。

動的束縛はコードの再利用の可能性を高める。例えば、Point クラスに定義された moveAndPrint メソッドを考える。

```

public void moveAndPrint(int dx, int dy) {
    print(); move(dx, dy); print();
}

```

`moveAndPrint` は `ColorPoint` にも `DeepPoint` にも適用できて、`print` メソッドは、それぞれのクラスのもの呼び出してくれる。動的束縛がなければ、ほとんど同じようなメソッドを何種類も定義しなければならない。例えば、`print` メソッドをオーバーライドすれば、`print` を間接的に呼び出すすべてのメソッドをオーバーライドしなければいけない。

ポリモルフィズム— 関数などが様々な型の引数に対して適用できること（しかも実行時の型によって振舞いが異なること²）

“Poly” は“多くの”という意味³、“Morph” は“形”という意味で、1つの関数がいろいろな型（形）に対して適用可能であることを表す。

今まででも継承を用いてサブクラスを定義するとき、スーパークラスに対して定義されていたメソッドを、そのまま何気なくサブクラスにも適用していた。このようなことが可能なのも、ポリモルフィズムがサポートされているからである。

グラフィカルユーザインタフェース (GUI) を用いるアプリケーションでは、ボタン・ラベル・テキストフィールドなどのように、ある面ではほとんど同じだが微妙に異なるというデータ型を扱うことが多い。Java ではこれらの部品に対して移動・拡大/縮小・削除などの操作を同じような方法で行なうことができる。このようなプログラムで、一つのメソッドを多くのデータ型に対して再利用するために、動的束縛は欠かせない機能である。

例えば、`JButton`, `JLabel`, `JTextField`, `JTextArea` などの GUI 部品はすべて `Component`（正確には `java.awt.Component`）のサブクラスである。だから、どの部品も `Component` のメソッドである `setVisible`, `setEnabled`, `setLocation` などを持っている。次のような例を試してみよう。

²本来は、ポリモルフィズムという言葉の中にこの意味は含まれていないが、人によってはポリモルフィズムをこのかっこの中の意味で用いることもある。

³ポリエチレン、ポリゴン（=多角形）、ポリネシアなどの“ポリ”と同じ語源

例題 5.6.1 ファイル *HideShow.java*

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class HideShow extends JApplet implements ActionListener {
    JTextField input;
    JLabel l1;
    JButton b1, b2;

    @Override
    public void init() {
        l1 = new JLabel("label");
        input = new JTextField("text", 5);
        b1 = new JButton("Hide"); b1.addActionListener(this);
        b2 = new JButton("Show"); b2.addActionListener(this);
        setLayout(new FlowLayout());
        add(l1); add(input); add(b1); add(b2);
    }

    public void actionPerformed(ActionEvent e) {
        if (e.getSource()==b1) {
            l1.setVisible(false); input.setVisible(false); b1.setVisible(false);
        } else if (e.getSource()==b2) {
            l1.setVisible(true); input.setVisible(true); b1.setVisible(true);
        }
        repaint();
    }
}

```



最初の状態



“Hide” ボタンを押した状態

どの型の部品も `setVisible` メソッドに同じように反応している。これらはすべて `Component` 型の変数に代入できるし、`Component` 型の引数を取るメソッド（例えば `add` など）に同じように渡すことができる。また、配列などにこのクラスのサブクラスを詰め込んで、一斉にメッセージを送る（=メソッドを起動する）ことなどもできる。

しかし、これらのクラスはフィールドの種類や数も異なるし、それにとまって、`setVisible` などのメソッドのそれぞれのクラスでの実装も少しずつ異なるかもしれない。（`setVisible` メソッドの実装自体は同一かもしれないが、一般的にはそこから間接的に呼び出されるメソッド（`paint` など）の実装は異なる。）これもポリモルフィズム（動的束縛）の一例である。

詳細: 動的束縛と混同しやすい概念として多重定義（オーバーロード）というものがある。多重定義とは、引数の型や数の異なる同じ名前のメソッドを定義することである。

ファイル OverloadTest.java

```
public class OverloadTest {
    double x, y;
    // コンストラクタの定義省略
    public void foo(double dx, double dy) { // foo その 1
        x+=dx; y+=dy;
    }
    public void foo(int dx, int dy) {      // foo その 2
        x*=dx; y*=dy;
    }
}
```

ファイル OverloadTest.java (つづき)

```
public static void main(String[] args) {
    OverloadTest o = new OverloadTest(1.1, 2.2);
    o.foo(3.3, 4.4); // foo その 1 が呼ばれる
    o.print();
    o.foo(2, 3);    // foo その 2 が呼ばれる
    o.print();
}
}
```

実行結果:

.....
.....
.....

動的束縛と決定的に異なる点は、多重定義はコンパイル時に（つまり静的に）解決されてしまうことである。

これは、さらに次のようなメソッドを定義するとはっきりする。

ファイル OverloadTest.java (メソッド定義の追加)

```
public void bar(Point p) { // bar その 1
    System.out.print("Point class: ");
    p.print();
    System.out.println();
}

public void bar(ColorPoint p) { // bar その 2
    System.out.print("ColorPoint class: ");
    p.print();
    System.out.println();
}
```

ファイル OverloadTest.java (main メソッドに追加)

```
ColorPoint cp = new ColorPoint(0, 0, "red");
Point p = cp;
o.bar(cp); // bar その 2 が呼ばれる
o.bar(p); // bar その 1 が呼ばれる
```

実行結果:

.....

5.7 総称クラスの定義

総称クラス（型パラメータを持つクラス）を定義するときはクラス名の後に<と>で囲って型パラメータを書く。この型パラメータはフィールドやメソッドの型の中で使用することができる。

Pair クラスでは E1, E2 が型パラメータである。

ファイル Pair.java

```
public class Pair<E1, E2> {
    public E1 fst;
    public E2 snd;
    public Pair(E1 f, E2 s) {
        fst=f; snd=s;
    }
}
```

ファイル Triple.java

```
public class Triple<E1, E2, E3> extends Pair<E1, E2> {
    public E3 thd;
    public Triple(E1 f, E2 s, E3 t) {
        super(f, s);
        thd = t;
    }
}
```

ファイル TripleTest.java

```
public class TripleTest {
    public static void main(String[] args) {
        Triple<Integer, String, Double> test
            = new Triple<Integer, String, Double>(1, "abc", 1.4);
        System.out.printf("(%d, %s, %g)%n", test.fst, test.snd, test.thd);
    }
}
```

キーワード オブジェクト指向, クラス, フィールド (メンバ変数), メソッド (メンバ関数), インスタンス, 継承 (インヘリタンス), スーパークラス, サブクラス, プライベートメンバ, パブリックメンバ, 情報隠蔽, カプセル化, ポリモルフィズム, 動的束縛, 多重定義