第5章　オブジェクト指向

これまで定義したクラスは、全て 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活きる技術であり、この章の例は 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、今までも行なってきたが、今回は一から定義するので extends 以下がない。

```
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。この時、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 つ含

むようなデータ構造である。

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歩手前である。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、そのプラスα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。

```
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.print("(" + x + ", " + y + ")");
  }

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

move と print はこのクラスのメソッドである。メソッドの中では、他のフィールド（例えば x, y）やメソッドを . 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。他のメソッドと異なり、コンストラクタの戻り値の型は、上の例のように指定しない。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、x が 1、y が 2 に初期化される。

下の 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。

```
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（他のフィールドに依存しないこと）表す修飾子である。特にmain関数はstaticでなければならない。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。

以上のコードを同一のファイルにまとめて、PointTest.javaという名前をつける。（Pointクラスにはpublicがついていないことに注意する。原則として、一つのファイルにpublicなクラスは一つしか定義できない。Pointクラスと PointTestクラスを別のファイルに定義する場合は、Pointクラスもpublicにしてもよい。）コンパイルはアプレットと同じようにjavac PointTest.javaで構わない。実行するには、

```
java PointTest
```

である。

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。

```
class ColorPoint extends Point {
  public String color;
  public ColorPoint(int x, int y, String c) {
    super(x, y);  /* 1 */
    color = c;
  }
  public void print() {
    System.out.print("<font color='"+color+'>");     // 色の指定
    System.out.print("(" + x + ", " + y + ")");
              // 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Applet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。

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。

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。

(/* 2 */)

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

緑色で “(11, 19)” と表示されるはずである。

5.5　情報隠蔽

ところで、color フィールドは、"red", "green" など、色を表す文字列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]。

```
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
  public void print() {
    System.out.print("<font color='"+getColor()+"'>");     // 色の指定
    // System.out.print("(" + 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; !cs[i].equals(""); 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];
  }
}
```

[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。

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、プログラムの他の部分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。(p.color = 100; のような操作ができないようにしたい。) 同じクラスのメソッドからは見えるが、プログラムの他の部分からは見えないフィールドを _____ であるという。逆に他のインスタンスのメソッドからでも見えるフィールドを _____ 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 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。

```
...
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 もプライベートにしておいても良い。) 他のインスタンスのメソッドで、例えば p.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[2]。また、private, public, protected の、どの指定もない場合もある。この場合の意味は 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、これ以上立ち入らないことにする。

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ことも可能である。)

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 *Point* クラスを継承し、新しいフィールド *int depth* を持っている。*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。

1. *DeepPoint* クラスを定義せよ。
2. *depth* が *1* 〜 *10* の値に制限されるように *setDepth* (及び *getDepth*) を定義せよ。

問 5.5.3 *SecretPoint* クラスは、*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。*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。

[2] なお、この他に protected という修飾子もあるが、ここでは紹介しない。

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。
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、おな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 のメソッドは全て仮想関数である。
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、次のような関数を考える。

```
static void moveInSquare(Point p) {
  p.move(1, 0);
  p.print();
  System.out.print(" ");
  p.move(0, 1);
  p.print();
  System.out.print(" ");
  p.move(-1, 0);
  p.print();
  System.out.print(" ");
  p.move(0, -1);
  p.print();
  System.out.println("<br>");
}
```

moveInSquare 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を呼び出してくれる。動的束縛がなければ、ほとんど同じような関数を 3 種類定義しなければならない。

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[3]）

“Poly” は “多くの” という意味[4]、“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。

今までにでも継承を用いてサブクラスを定義する時に、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が、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ーフェース (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。

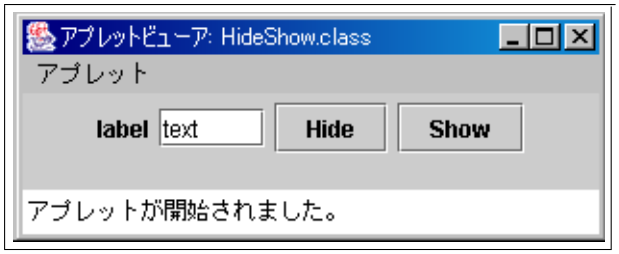
例えば、Button, Label, TextField, TextArea などの GUI 部品はすべて Component （正確には java.awt.Component）のサブクラスである。だから、どの部品も Component のメソッドである setVisible, setEnabled, setLocation などを持っている。次のような例を試してみよう。

例題 5.6.1 ファイル *HideShow.java*

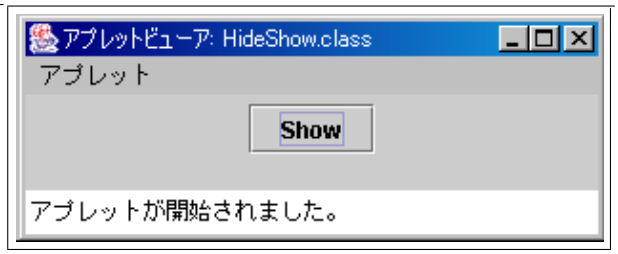
```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

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

  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);
    getContentPane().setLayout(new FlowLayout());
    getContentPane().add(l1); getContentPane().add(input);
    getContentPane().add(b1); getContentPane().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();
  }
}
```

[3]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。

[4]ポリエチレン、ポリゴン（=多角形）などの “ポリ” と同じ語源



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、それにともなって、`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。これもポリモルフィズムの一例である。

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