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活きる技術であり、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、今までも行なってきたが、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（より正確には java.lang.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＿＿＿＿＿＿（空欄 5.1.1）である。（メンバ変数, インスタンス変数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 ____（空欄 5.2.1）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_____（空欄 5.2.2）（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「_（空欄 5.2.3）」（ドット）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（_____（空欄 5.3.1）, メンバ関数）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();
  }
```

```
 // コンストラクター
  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（コンストラクター）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ーを定義している。

> **詳細:** プログラマがコンストラクターを 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ーが自動的に用意される。

他のメソッドの場合と異なり、コンストラクターの定義のときは戻り値の型は指定しない。

コンストラクターを使うと、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）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＿＿＿＿(空欄 5.4.1)（インヘリット）する（名詞形は＿＿＿＿＿＿＿＿＿＿(空欄 5.4.2)）という。Point クラスは ColorPoint クラスの＿＿＿＿＿＿＿＿＿＿(空欄 5.4.3)である、という。逆に ColorPoint クラスは Point クラスの＿＿＿＿＿＿(空欄 5.4.4)である。

継承するときは、クラス名の後に「extends」後に続けてスーパークラスの名前を一つだけ書く。以下のファイルを Point.java と同じディレクトリに置く。

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ー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。）コンストラクターの中の super(x, y) という式（/* 1 */）はスーパークラス（Point）のコンストラクターを呼び出す。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ーでは、最初の文でスーパークラスのコンストラクター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ーを持っていて、スー

パークラスのコンストラクター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。

Q 5.4.1 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。コンストラクターは 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 クラスの定義を完成させよ。

ファイル DeepPoint.java

```
public class DeepPoint ____________________ {

    ____________________     // フィールドの定義

    public DeepPoint(int x, int y, int d) {

        ________________
        depth = d;
    }


    ____________
    public void print() {
        int i;
        for (i=0; i<depth; i++) {
```

```
            System.out.print("(");
        }
        System.out.printf("%d,␣%d", x, y);
        for (i=0; i<depth; i++) {
            System.out.print(")");
        }
    }
}
```

5.5　動的束縛

次のような例を考える。

PointTest クラスに testPoint という Point を引数として受け取る静的メソッドを用意し、

ファイル PointTest.java（バージョン 3）

```
public static void testPoint(Point p) {
    p.move(10, 10);
    p.print();
}
```

main メソッドでは、Point, ColorPoint, DeepPoint の 3 つのクラスのインスタンスを生成し、testPoint メソッドに渡す。

ファイル PointTest.java（バージョン 3、続き）

```
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);

  testPoint(p);
  testPoint(cp);
  testPoint(dp);
}
```

testPoint メソッドを呼び出すときに、ColorPoint, 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、一般的にサブクラスの方がスーパークラスよりメソッドが多いので可能である。

> **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.move(1, -1);
> ```

一方、スーパークラスの型を持つ式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）

```
// 次の行をコメントアウトすると実行時エラー
// p = new Point(3, 4);
ColorPoint cp2 = (ColorPoint)p;  // 明示的なキャスト
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例外 ClassCastException が発生する。

testPoint メソッドの中では何が起こるだろうか? testPoint メソッドの中で呼び出される move メソッドは各クラスで共通なので、同じメソッドが起動される。しかし、print メソッドは ColorPoint 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?

Q 5.5.1 上記の PointText.java（バージョン 3）の出力を予想せよ。

(1). `(11, 12)(13, 14)(15, 16)`

(2). `(11, 12)<font color='green'>(13, 14)</font>(((((15, 16)))))`

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、＿＿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＿＿＿＿＿（空欄 5.5.1）(dynamic binding) 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* p = new Point(1, 2);
ColorPoint* cp = new ColorPoint(3, 4, "green");
DeepPoint* cp  = new DeepPoint(5, 6, 5);

testPoint(p);
testPoint(cp);
testPoint(dp);
// ...
```

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＿＿＿＿＿＿（空欄 5.5.2） (virtual function) というものにする必要がある。仮想関数とは、ポインタの型ではなく、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 (polymorphism) — 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[1]）

[1]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。

“Poly” は “多くの” という意味[2]、“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。

今までででも継承を用いてサブクラスを定義するときに、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。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ーインタフェース (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。

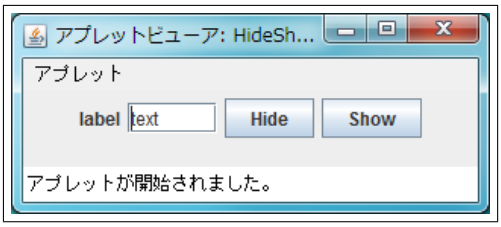
例題 5.5.2

ファイル HideShow.java

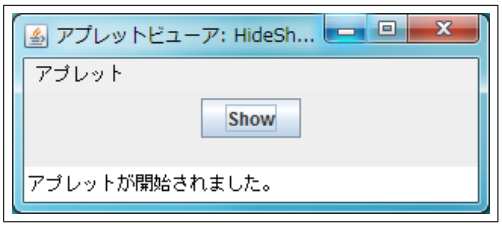
```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

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);
```

[2]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

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

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

```
    }
  }
```

注意: このプログラムは多重定義の使い方としては**悪い**例である。

Q 5.5.3 OverloadTest.java の出力を予測せよ。

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

動的束縛と決定的に異なる点は、多重定義は静的に（つまりコンパイル時に）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（bar メソッドの追加）

```
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
```

Q 5.5.4 OverloadTest.java（bar メソッド追加後）の出力を予測せよ。

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

つまり Java では動的束縛が起こるのは . 演算子の前のパラメーターに限られるのである。

5.6　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、専用の設定メソッドを設けて、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"のいずれかの文字列で指定することにする。

文字列同士が同じ文字列がどうかを判定するには String クラスの＿＿＿＿＿＿（空欄 5.6.1）というメソッドを用いる。String クラスに対する==演算子は物理的に同じオブジェクトかどうかを判定するので、==の結果が true にならなくても、equals の結果が true になることがある。

java.lang.String クラス

public boolean equals(Object s)

この文字列と指定されたオブジェクトを比較する。

public boolean equalsIgnoreCase(String s)

この文字列と指定された文字列を比較する。大文字小文字を区別しない。

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 String color;

  @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 = c; 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);
    if (color==null) color = "black";
  }
```

```
  public String getColor() {
    return 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、有効な色名以外の値を設定できないようにしたい。(つまり、cp.color = "NoSuchColor"; 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＿＿＿＿＿＿＿＿(空欄 5.6.2)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パブリックであるという。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＿＿＿＿＿＿(空欄 5.6.3)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。

```
  ...
  private String 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) にしておくとい。) 他のインスタンスのメソッドで、例えば cp.color = "NoSuchColor"; 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、これらの修飾子の説明は割愛する。

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＿＿＿＿＿＿(空欄 5.6.4)あるいは＿＿＿＿＿＿＿＿(空欄 5.6.5) (encapsulation) 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"などの文字列の配列 cs 中の添字を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いても、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。

クラスを設計するとき、外部から使用する必要のないメソッドやフィールドは private と宣言して隠蔽するべきである。ただし、何でもかんでも private とすれば良いというわけでもない。外部から必要な操作ができないクラスを設計してしまえば、ソースのコピーという最悪のかたちの再利用をせざるを得なくなってしまうからである。

問 5.6.1 color フィールドが、各色に対応する配列 cs 中の要素の添字（int 型）で表すように ColorPoint の実装を変更せよ[3]。

問 5.6.2 DeepPoint クラスに depth が 1 〜 10 の値に制限されるように設定するメソッド void setDepth(int d)（および depth を読み出すメソッド int getDepth()）を定義せよ。depth フィールドの値は他のオブジェクトからは setDepth メソッドを通じてのみ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。また、コンストラクターに depth の初期値として、 0 以下または 11 以上の値が引数として渡されたときは 1 になるようにせよ。）

問 5.6.3 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ー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·x+b·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。

5.7　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;
  }
```

[3]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、単なる説明のためである。

```
}
```

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