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 `JPanel` 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、本来はもっと大きな例を取り上げるべきだが、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（継承を使わず）定義する（実は、`java.awt.Point` という、ほとんど同じようなクラスが Java の標準ライブラリーにあるが、それは今回は使わない。）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` と書くのと同じことになる。

ファイル Point.java（バージョン 0）

```
1 public class Point {
2   // フィールド （インスタンス変数）
3   public int x;
4   public int y;
5 }
```

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。（メンバー変数, 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 `int` 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。

ファイル PointTest0.java

```
1 void main() {
2   Point p = new Point();
3   p.x = 1; p.y = 2;
4   System.out.printf("(%d, %d)", p.x, p.y);
5 }
```

オブジェクトのフィールドには「 __ 」（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。

> **詳細:** PointTest0.java と Point.java を同じディレクトリーに置いておくと、PointTest0.java をコンパイル（javac PointTest0.java）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。

なお、次の例を実行してみるとわかるように、別々のインスタンスのフィールドは別々の領域に割り当てられている。

ファイル PointTest0b.java

```
1 void main() {
2   Point p = new Point(), q = new Point();
3   p.x = 1; p.y = 2;
4   q.x = 9; q.y = 8;
5   System.out.println("(" + p.x + ", " + p.y + ")");
6   System.out.println("(" + q.x + ", " + q.y + ")");
7 }
```

問 5.2.1 上の PointTest0b クラス + Point クラス（バージョン 0）を実行したときの出力を書け。

_____

_____

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（＿＿＿＿, メンバー関数）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 1）

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

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

  public double distance() {
    return Math.sqrt(x * x + y * y);
  }

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

  public void info() {
    System.out.printf("x: %d, y: %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 `distance`, `print`, `info`, `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` や `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:** プログラマー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（例えば `int` 型の場合は `0`）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。つまり、`Point` の場合、 `Point() { x = 0; y = 0 }` とコンストラクターを定義するのと同等である。

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 new というキーワードを使って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 PointTest1 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 PointTest1.java

```
1 void main() {
2   Point p = new Point(10, 20);
3   p.move(1, -1);
4   p.print();
5 }
```

Q 5.3.1 上記の PointTest1 クラス + Point クラス（バージョン 1）を実行したときの出力を書け。

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＿＿（インヘリット）する（名詞形は＿＿＿＿）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、クラス名の後に「extends」の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ーに置く。

ファイル ColorPoint.java（バージョン 1）

```
1  public class ColorPoint extends Point {
2    public String color;
3    public ColorPoint(int x, int y, String c) {
4      super(x, y);  /* 1 */
5      color = c;
6    }
7
8    @Override
9    public void print() {
10     System.out.printf("<span style='color: %s;'>",
```

```
                        color);
    // 次の行は super.print(); でも可
    System.out.printf("(%d, %d)", x, y);    /* 2 */
    System.out.print("</span>");
  }

  @Override
  public void info() {
    super.info();
    System.out.printf(", color: %s", color);
  }
}
```

詳細: Java 以外の言語では、スーパークラスを二つ以上指定することができるものもある。これを多重継承という。Java は多重継承を許さないので、`extends` のあとに書けるクラス名は一つだけである。

`ColorPoint` では、新しいフィールド `color` と再定義するメソッド `print()` と `info()`、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 GUI アプリケーションを簡単に作成できたのはスーパークラスの `JPanel` 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 ブラウザーで表示すると、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 ColorPointTest.java

```
void main() {
```

```
  ColorPoint cp = new ColorPoint(10, 20, "green"); /*
3 */
  cp.move(1, -1); /* 4 */
  cp.print();
}
```

Q 5.4.1 上記の ColorPointTest クラス + ColorPoint（バージョン 1）+ Point（バージョン 1）を実行したときの出力を書け。

_____

Q 5.4.2 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(")");
    }
  }

  _____
  public void info() {
    super.info();
    System.out.printf(", depth: %d", depth);
  }
}
```

さらに次の DeepPointTest クラス + 定義した DeepPoint + Point（バージョン 1）を実行したときの出力を予想せよ。

ファイル DeepPointTest.java

```
void main() {
  DeepPoint dp = new DeepPoint(10, 20, 4);
```

```
  dp.move(1, -1);
  dp.print();
}
```

5.5 動的束縛

次のような例を考える。

TestMove というクラスに testMove 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 TestMove.java

```
void testMove(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、testMove メソッドに渡す。

ファイル TestMove.java（続き）

```
void main() {
  Point p = new Point(1, 2);
  ColorPoint cp = new ColorPoint(3, 4, "green");
  DeepPoint dp = new DeepPoint(5, 6, 5);

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

testMove 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。サブクラスのほうが、フィールドやメソッドをより多く持っているので、スーパークラスの代わりが務まるからである。
ファイル CastTest.java

```
void main() {
  ColorPoint cp = new ColorPoint(1, 2, "red");
  Point p = cp; // ワイドニング
  p.print();

```

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 q = (ColorPoint) p; // ナローイング
  q.color = "blue";
  q.print();
}
```

ここで 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。

なお、キャスト時の例外の発生を防ぐためには、instanceof という演算子を使うことができる。 instanceof 演算子は、左オペランドのオブジェクトが右オペランドのクラスのインスタンスであるときに真となる。

CastTest.java の書き換え案 1

```
    if (p instanceof ColorPoint) {
      ColorPoint q = (ColorPoint)p; // ナローイング
      q.color = "blue";
      q.print();
    } else {
      System.out.println("p is not a ColorPoint.");
    }
```

さらに Java 16 からは、右オペランドのクラスのあとにフレッシュな変数を書くことによって、キャストを書かなくても良くなった。つまり、同等の処理を次のように書くことができる。

CastTest.java の書き換え案 2

```
    if (p instanceof ColorPoint q) {
      q.color = "blue";
      q.print();
    } else {
      System.out.println("p is not a ColorPoint.");
    }
```

testMove メソッドの中では何が起こるだろうか? testMove 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 TestMove（Point, ColorPoint, DeepPoint は、ここまでの最新版）の出力を予想せよ。

1. (11, 12)(13, 14)(15, 16)
2. (11, 12)(13, 14)(((((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 _____ (dynamic binding) という。

- 静的 (static) — プログラムを実行する前（コンパイル時）にわかる性質、つまり、具体的な値がわからなくて、条件判断や繰り返しをしなくてもわかる性質
- 動的 (dynamic) — プログラムを実行してみないとわからない性質

（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*  dp = new DeepPoint(5, 6, 5);

    testMove(p);
    testMove(cp);
    testMove(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 _____ (virtual function) というものにする必要がある。仮想関数とは、ポインターの型ではなく、ポインター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++ のメンバー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。

```
1  public void moveAndPrint(int dx, int dy) {
2    print(); move(dx, dy); print();
3  }
```

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。

ファイル MoveAndPrintTest.java

```
1  void main() {
2    Point       p = new Point(1, 2);
3    ColorPoint cp = new ColorPoint(3, 4, "green");
4    DeepPoint  dp = new DeepPoint(5, 6, 3);
5
6    p.moveAndPrint(1, 1);
7    System.out.println();
8    cp.moveAndPrint(2, 2);
9    System.out.println();
10   dp.moveAndPrint(3, 3);
11   System.out.println();
12 }
```

Q 5.5.2 上記の MoveAndPrintTest クラス + Point（バージョン 1） + ColorPoint（バージョン 1） + DeepPoint（Q 5.4.2）を実行したときの出力を書け。

動的束縛がなければ（静的束縛ならば） moveAndPrint をコンパイルする時点で、既知のクラスは Point クラスだけだから、move と print は Point クラスのものになる。そうすると、ほとんど同じようなメソッドをクラス毎に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、moveAndPrint のような、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。 また GUI アプリケーションの場合 paintComponent メソッドをオーバーライドするたびに repaint メソッドを再定義する必要があるだろう。

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。

ポリモルフィズム (polymorphism・多相) — 関数などが様々な型の引数に対して適用できること (本来は、ポリモルフィズムという言葉の意味は、 これだけだ

が、 オブジェクト指向言語の文脈では「ポリモルフィズム」を「動的束縛」 の意味も含めて「関数などが様々な型の引数に対して適用でき、実行時の 型によって振舞いが異なること」として用いることがある。》

"Poly"は"多くの"という意味 (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)、"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。

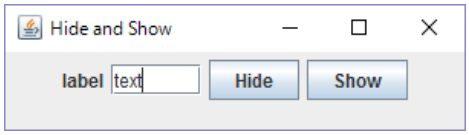
今までても継承を用いてサブクラスを定義するときに、スーパークラスに対して定義されていたメソッドを、そのまま何気なくサブクラスにも適用していた。このようなことが可能なのも、ポリモルフィズムがサポートされているからである。

例えば、`JButton`, `JLabel`, `JTextField`, `JTextArea`などの GUI 部品はすべて `JComponent` （正確には `javax.swing.JComponent`）のサブクラスである。だから、どの部品も `JComponent` 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。

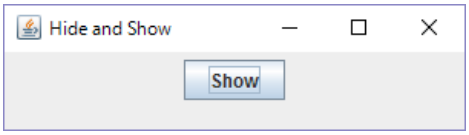
例題 5.5.3 ファイル HideShow.java

```
 1 import javax.swing.*;
 2 import java.awt.*;
 3 import java.awt.event.*;
 4
 5 class HideShow extends JPanel
 6                 implements ActionListener {
 7   private JTextField input;
 8   private JLabel lbl;
 9   private JButton b1, b2;
10
11   public HideShow() {
12     setPreferredSize(new Dimension(300, 50));
13     lbl   = new JLabel("label");
14     input = new JTextField("text", 5);
15     b1 = new JButton("Hide");
16     b2 = new JButton("Show");
17     b1.addActionListener(this);
18     b2.addActionListener(this);
19     setLayout(new FlowLayout());
20     add(lbl); add(input); add(b1); add(b2);
21   }
22
23   public void actionPerformed(ActionEvent e) {
24     if (e.getSource() == b1) {
25       lbl.setVisible(false);
26       input.setVisible(false);
27       b1.setVisible(false);
28     } else if (e.getSource() == b2) {
29       lbl.setVisible(true);
30       input.setVisible(true);
31       b1.setVisible(true);
32     }
33   }
34 }
35
```

```
36 void main() {/* 省略 */}
```



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 `JComponent` 型の変数に代入できるし、`JComponent` 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（つまり、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 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。もし動的束縛がなければ（静的束縛ならば）コンパイル時にクラスが固定されてしまうため、ほとんど同じようなメソッドをクラス毎に定義する必要がある。

5.6 多重定義（オーバーローディング）

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

```
 1 class OverloadTest {
 2   double x, y;
 3
 4   public OverloadTest(double x0, double y0) {
 5     x = x0; y = y0;
 6   }
 7
 8   // foo-1
 9   public void foo(double dx, double dy) {
10     x += dx; y += dy;
11   }
12
13   // foo-2
14   public void foo(int dx, int dy) {
15     x *= dx; y *= dy;
16   }
```

```
32   /* 1 */
33   public void print() {
34     System.out.printf("(%g, %g)", x, y);
35     System.out.println();
36   }
37 }
38
39 void main(String[] args)  {
```

```
40    OverloadTest o = new OverloadTest(1.1, 2.2);
41    o.foo(3.3, 4.4); // foo-1 が呼ばれる
42    o.print();
43    o.foo(2, 3);     // foo-2 が呼ばれる
44    o.print();
45    /* 2 */
```

```
51  }
```

注意: 通常、多重定義は同じような動作をする関数に使用する。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 5.6.1 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 `/* 1 */` の部分に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）

```
18    // bar-1
19    public void bar(Point p) {
20      System.out.print("Point class: ");
21      p.print();
22      System.out.println();
23    }
24
25    // bar-2
26    public void bar(ColorPoint p) {
27      System.out.print("ColorPoint class: ");
28      p.print();
29      System.out.println();
30    }
```

ファイル OverloadTest.java（`/* 2 */` の部分に追加）

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

Q 5.6.2 OverloadTest.java （`/* 1 */`, `/* 2 */` の部分に追加後）の出力の追加部分を予測せよ。

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。

多重定義も広い意味では多相（ポリモルフィズム）の一種だが、オブジェクト指向言語に特有のものではない。例えば C 言語でも「+」「*」などの算術演算子は多重定義されている。

5.7 カプセル化

ところで、ColorPoint クラスの 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。このようにフィールドの値を制限したいというのは、他のクラスでもよく起こりうる状況である。

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 _____ 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）

```
 1 public class ColorPoint extends Point {
 2   public String[] cs = {
 3     "black", "red", "green", "yellow",
 4     "blue", "magenta", "cyan", "white"};
 5   public String color;
 6
 7   @Override
 8   public void print() {
 9     System.out.printf("<span style='color: %s;'>",
10                       getColor());
11     // 次の行は super.print(); でも可
12     System.out.printf("(%d, %d)", x, y);
13     System.out.print("</span>");
14   }
15
16   public void setColor(String c) {
17     int i;
18     for (i = 0; i < cs.length; i++) {
19       if (c.equals(cs[i])) {
20         color = c; return;
21       }
```

```
    }
    // 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＿＿＿＿＿＿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＿＿＿＿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（ColorPoint.java（バージョン 3））。

```
…
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 static あるいは final（後述）にしておくべきだろう。）他のクラスやインスタンスのメソッド（例えば PointTest クラスの main 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。

ファイル ColorPointTest2.java

```
void main() {
  ColorPoint cp = new ColorPoint(10, 20, "green");
  cp.move(1, -1);
  cp.color = "noSuchColor"; /* エラーになる */
  cp.print();
}
```

Q 5.7.1 実際にどのようなエラーメッセージが出力されるか確かめよ。

＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿

＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿

＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿

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: この他に `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。これをパッケージプライベート (package private) ということもある。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。これらの修飾子はパッケージを紹介したあとに説明する。

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同じクラスのメソッド以外に非公開にすることを＿＿＿＿あるいは＿＿＿＿(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ことに注意する。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。

詳細: `public`, `private`, `protected` などの修飾子によるアクセス制限は、Java の一番最初からあった仕組みだが、これだけでは仕組みとして十分でないことがだんだんわかってきた。このため Java 9 より Jigsaw の名前で知られるモジュールシステムが導入された。ここでは Jigsaw の説明は割愛する。

問 5.7.2 `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 (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。)

問 5.7.3 `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。（`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それぞれ 1 または 10 になるようにせよ。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も 1 または 10 になるようにせよ。）そして `depth`

フィールドの値は他のクラス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せよ。

問 5.7.4 `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 $a \cdot x + b \cdot y = 1$ 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他のクラスのオブジェクトから値が見えないようにせよ。

5.8 総称クラスの定義

総称クラス（型パラメーターを持つクラス）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ーを書く。この型パラメーター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` が型パラメーターである。

ファイル Pair.java

```
1 public class Pair<E1, E2> {
2   public E1 fst;
3   public E2 snd;
4   public Pair(E1 f, E2 s) {
5     fst = f; snd = s;
6   }
7 }
```

ファイル Triple.java

```
1 public class Triple<E1, E2, E3> extends Pair<E1, E2> {
2   public E3 thd;
3   public Triple(E1 f, E2 s, E3 t) {
4     super(f, s);
5     thd = t;
6   }
7 }
```

ファイル TripleTest.java

```
1 void main() {
2   Triple<Integer, String, Double> test = new Triple<>
  (1, "abc", 1.4);
3   System.out.printf("(%d, %s, %g)%n",
4       test.fst, test.snd, test.thd);
5 }
```

5.9 章末問題

問 5.9.1 ゲームのクラス階層を設計するために、テスト用のいくつかのクラスを作成した。まず、`Player` クラスは味方キャラクターを表すクラスである。

さらに、敵キャラクターのためにいくつかのクラスがある。Enemy クラスはすべての敵キャラクターのスーパークラスとなるクラスである。 GenericEnemy クラスと Boss クラスは Enemy クラスを継承し、さらに、Alice クラスと Grace クラスは、 GenericEnemy クラスを継承する。なお、これらのクラス名には特に意味はない。

Enemy クラスは attack メソッドを持ち、 GenericEnemy クラスは updateDamage メソッドを持つ。

ファイル Player.java

```
1  public class Player {
2    private int hp;
3
4    public Player(int initHp) {
5      hp = initHp;
6    }
7
8    public void damage(int damage) {
9      hp -= damage;
10     System.out.printf("%5d のダメージ、残り HP =
   %5d%n",
11                       damage, hp);
12   }
13 }
```

ファイル Enemy.java

```
1  public class Enemy {
2      public void attack(Player p) {}
3  }
```

ファイル GenericEnemy.java

```
1  public class GenericEnemy ____________________________ {
2    private String name;
3    public int damage;
4
5    public GenericEnemy(String n, int d) {
6      name = n;
7      damage = d;
8    }
9
10   @Override
11   public void attack(Player p) {
12     System.out.print(name + " の攻撃: ");
13     p.damage(damage);
14     updateDamage();
15   }
16
17   public void updateDamage() {}
18 }
```

ファイル Alice.java

```
1  public class Alice ____________________________ {
```

```
  private int init;

  public Alice(int d) {
    super("Alice", d);
    init = d;
  }

  @Override
  public void updateDamage() {
    damage += init;    // エラーにならない
    // name += "!";    // エラーになるのでコメントアウト
  }
}
```

ファイル Grace.java

```
public class Grace _____ {
  private int ratio;

  public Grace(int d, int r) {
    super("Grace", d);
    ratio = r;
  }

  @Override
  public void updateDamage() {
    damage *= ratio;  // エラーにならない
    // name += "?";    // エラーになるのでコメントアウト
  }
}
```

ファイル Boss.java

```
public class Boss _____ {
  public int count;

  public Boss() {
      count = 0;
  }

  @Override
  public void attack(Player p) {
    System.out.print("Boss の攻撃: ");
    if (++count >= 3) {
      p.damage(1000);
    } else {
      p.damage(0);
    }
  }
}
```

さらに、次のような `main` メソッドを持つテスト用プログラム Main.java を定義する。

ファイル Main.java

```
import java.util.ArrayList;

void main() {
```

```
    // 空の ArrayList
    ArrayList<Enemy> enemies = new ArrayList<>();
    Player p = new Player(3000);

    Alice a = new Alice(100);
    enemies.add(a);
    Grace g = new Grace(100, 2);
    enemies.add(g);
    Boss b  = new Boss();
    enemies.add(b);

    for (int i = 0; i < 3; i++) {
      for (int j = 0; j < enemies.size(); j++) {
        Enemy e = enemies.get(j);
        e.attack(p);
      }
    }
}
```

空欄を埋めてこれらのクラスの定義を完成させよ。さらに、Main クラスの main メソッドを実行するとき、出力はどうなるか？

キーワード

オブジェクト指向、クラス、フィールド（メンバー変数）、メソッド（メンバー関数）、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ー、パブリックメンバー、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