# 13–1 抽象クラス

第８章から前章までで、オブジェクト指向プログラミングの基本を学習しました。本章と次章では、やや応用的なことを学習します。本節で学習するのは、抽象クラスです。

## 抽象クラス

前章ではクラスの派生について学習しました。本章では、派生を応用して、図形を表すクラス群を作っていくことにします。

最初に考える図形は、《点》と《長方形》です。両方のクラスに、描画のためのメソッド *draw* をもたせることにします。二つのクラスは、以下のように設計します。

### ▪ 点クラス Point

点を表すクラスです。フィールドはもちません。メソッド *draw* は、以下のように実現して、記号文字 '+' を 1 個だけ表示します。

```
// クラスPointのメソッドdraw
void draw() {
   System.out.println('+');
}
```

```
+
```

### ▪ 長方形クラス Rectangle

長方形を表すクラスです。幅と高さを表す **int** 型のフィールド *width* と *height* をもたせます。メソッド *draw* は、以下のように実現して表示を行います。

```
// クラスRectangleのメソッドdraw
void draw() {
   for (int i = 1; i <= height; i++) {
      for (int j = 1; j <= width; j++)
         System.out.print('*');
      System.out.println();
   }
}
```

```
****
****
****
```

▶ ここに示す実行例は、幅（*width*）が 4 で高さ（*height*）が 3 の長方形の場合です。

個別に定義されたクラスでメソッド *draw* を作っても、それらは無関係なものとなってしまいます。前章で学習した《多相性》を有効に活用できるようにしましょう。そこで、以下のようにします。

**《図形》クラスから《点》と《長方形》を派生する。**

それでは、図形クラスを設計しましょう。

▪ 図形クラス Shape

図形を表すクラスです。点や長方形や直線などのクラスは、このクラスから直接的あるいは間接的に派生するものとします。

▪ **メソッド *draw* は何を行えばよいのでしょう？**

何を表示すべきでしょう。適切なものは見当たりません。

▪ **どのようにインスタンスを生成すればよいのでしょう？**

まだ各クラスのコンストラクタを設計していませんが、おそらく以下のように呼び出してインスタンスを生成するはずです。

クラス *Point* … `new Point()` ※引数はなし
クラス *Rectangle* … `new Rectangle(4, 3)` ※幅と高さを与える

クラス *Shape* のインスタンスは、このような形では生成できません。どのような引数を与えればよいのか、見当がつきません。

クラス *Shape* は、図形の設計図というよりも、図形という《**概念**》を表す抽象的な設計図です。*Shape* のように、

- インスタンスを生成できない、または生成すべきでない。
- メソッドの本体が定義できない。その内容はサブクラスで具体化すべきである。

といった性質をもったクラスを表すのが**抽象クラス**（*abstract class*）です。

クラス *Shape* を抽象クラスとして定義すると、以下のようになります。

```
// 図形クラス（抽象クラス）
abstract class Shape {
    abstract void draw();       // 描画（抽象メソッド）
}
```

クラス *Shape* とメソッド *draw* の両方に、`abstract` というキーワードが与えられている点に着目しましょう（詳細は次ページで学習します）。

なお、このクラスから派生するクラス *Point* とクラス *Rectangle* は、抽象クラスではなく、普通の（非抽象）クラスです。

三つのクラスのクラス階層図を **Fig.13-1** に示します。本書のクラス階層図では、抽象クラスの名前を*斜体*で表現します。

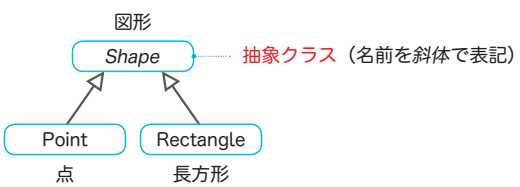


**Fig.13-1**　図形クラス群のクラス階層図

クラス *Shape* を抽象クラスとし、そこからクラス *Point* とクラス *Rectangle* を派生するように実現したプログラムを **List 13-1** に示します。

**List 13-1** shape1/Shape.java

抽象クラス

```
// 図形クラス群【第１版】

//===== 図形 =====//
abstract class Shape {
   abstract void draw();                    // 描画（抽象メソッド）
}

//===== 点 =====//
class Point extends Shape {
   Point() { }                              // コンストラクタ

   void draw() {                            // 描画
      System.out.println('+');
   }
}

//===== 長方形 =====//
class Rectangle extends Shape {
   private int width;       // 幅
   private int height;      // 高さ

   Rectangle(int width, int height) {  // コンストラクタ
      this.width = width;
      this.height = height;
   }

   void draw() {                            // 描画
      for (int i = 1; i <= height; i++) {
         for (int j = 1; j <= width; j++)
            System.out.print('*');
         System.out.println();
      }
   }
}
```

▶ 本来は、個々のクラスを独立したソースファイルで実現すべきですが、スペース節約のために、一つにまとめています。また、メソッドの **public** も省略しています。
なお、第２版では、個々のクラスを独立したソースファイルとして実現します。

## 抽象メソッド

前ページで簡単に学習したように、クラス *Shape* のメソッド *draw* の先頭に **abstract** が付いています。このように宣言されたメソッドは、**抽象メソッド**（*abstract method*）となります。メソッドの前に付けられた **abstract** は、

> **ここ（私のクラス）ではメソッドの実体を定義できませんから、私から派生したクラスで定義してください !!**

といったニュアンスです。

抽象メソッドには本体がありませんので、その宣言では、{ } の代わりに ; を置きます。たとえ空であってもブロック { } を書いてはいけません。

```
abstract void draw() {  }      // エラー：定義はできない
```

クラス *Point* とクラス *Rectangle* では、メソッド *draw* をオーバライドして本体を定義しています（メソッドの内容は、最初に設計したとおりです）。

このように、抽象クラスから派生したクラスで、抽象メソッドをオーバライドして本体を定義することを、メソッドを**実装する**（*implement*）といいます。

**重要** スーパークラスの抽象メソッドをオーバライドして、メソッド本体の定義を宣言することを、『**メソッドを実装する**』という。

クラス *Point* とクラス *Rectangle* は、抽象クラス *Shape* のメソッド *draw* を“実装”しているのです。

## 抽象クラス

クラス *Shape* のように、**抽象メソッドを１個でも有するクラスは、必ず抽象クラスとして宣言しなければなりません。**クラスを抽象クラスとして宣言するために与えるのが、`class` の前に置く `abstract` です。

ただし、抽象メソッドが１個もないクラスを、抽象クラスとすることもできますので、**Fig.13-2** のように理解しましょう。

▶ 抽象クラスに対して、`final`，`static`，`private` を指定することはできません。

**P.java** ✕

```
class P {
    abstract void a();
    void b() { /*…*/ }
}
```

抽象メソッドを１個でももつクラスは、抽象クラスでなければならない。
class の宣言に abstract がないため、コンパイルエラーとなる。

**Q.java** ○

```
abstract class Q {
    abstract void a();
    void b() { /*…*/ }
}
```

１個でも抽象メソッドをもつクラスは、抽象クラスでなければならない。

**R.java** ○

```
abstract class R {
    void a() { /*…*/ }
    void b() { /*…*/ }
}
```

抽象クラスは、抽象メソッドをもたなくてもよい。

**Fig.13-2**　抽象クラスと抽象メソッド

図形クラス群を利用するプログラムを **List 13-2** に示します。

**List 13-2** shape1/ShapeTester.java

```
// 図形クラス群【第１版】の利用例

class ShapeTester {

   public static void main(String[] args) {
//    以下の宣言はエラー：抽象クラスはインスタンス化できない
//    Shape s = new Shape();

      Shape[] a = new Shape[2];
      a[0] = new Point();            // 点
      a[1] = new Rectangle(4, 3);    // 長方形

      for (Shape s : a) {
         s.draw();       // 描画
         System.out.println();
      }
   }
}
```

**実行結果**

```
+

****
****
****
```

**別解** shape1/ShapeTester2.java

```
for (int i = 0; i < a.length; i++) {
   a[i].draw();
   System.out.println();
}
```

### ▪ 抽象クラスのインスタンスは生成できない

*s* の宣言がエラーとなることに注目しましょう（コメントアウトしています）。

抽象クラスは、具体的定義のないメソッドをもっているため、**new** *Shape*() によってインスタンスを生成することはできません。

**重要** 抽象クラスのインスタンスを生成することはできない。

▶ もし抽象クラスのインスタンスを生成できるとしたら、実体のないメソッド *draw* を *s.draw*() として呼び出せることになってしまいます。

### ▪ 抽象クラスと多相性

*a* は、*Shape* 型の配列です。各要素 a[0] と a[1] は *Shape* 型のクラス型変数であって、*Shape* から派生したクラスのインスタンスを参照しています。

▶ クラス型変数が、下位クラスのインスタンスを参照できる（p.403）ことを利用しています。

**Fig.13-3** に示すように、a[0] はクラス *Point* 型のインスタンスを参照し、a[1] はクラス *Rectangle* 型のインスタンスを参照します。

▶ 図が煩雑になるため書いていませんが、a[0] と a[1] と a.length をセットにした、配列本体用オブジェクトを参照するための、*Point*[] 型の配列変数 a も存在します。

拡張 **for** 文では、配列 a 中の要素に対してメソッド *draw* を呼び出します。先頭の要素に対しては、クラス *Point* のメソッド *draw* が呼び出され、２番目の要素に対しては、クラス *Rectangle* のメソッド *draw* が呼び出されることが、実行結果からも確認できます。

▶ 別解に示しているのは、基本 **for** 文によって実現したプログラムです。

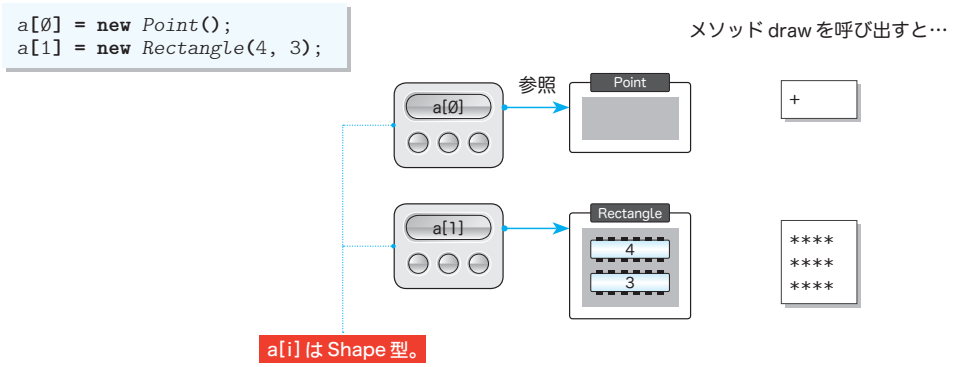


**Fig.13-3**　Shape 型の配列と多相性

抽象クラス *Shape* は、具体的な図形ではなく、図形の**概念**を表すクラスです。実体を生成できない**不完全なクラス**ではあるものの、**自分自身を含め、自分から派生したクラスに対して、《血縁関係》をもたせる**役目を果たしていることが分かりました。

**重 要**　下位クラスをグループ化して多相性を有効活用するためのクラスに具体的な実体がなければ、抽象クラスとして定義するとよい。

＊

**抽象クラスから派生したサブクラスで抽象メソッドを実装しなければ、抽象メソッドのまま継承されます。**このことを **Fig.13-4** で理解しましょう。

抽象クラス *A* の二つのメソッド a と b はいずれも抽象メソッドです。クラス *A* から派生して、メソッド b を実装していないクラス *B* も抽象クラスです。もしクラス *B* の宣言から **abstract** を省略するとコンパイルエラーとなります。なお、クラス *B* から派生したクラス *C* はメソッド b を実装していますので、抽象クラスではなくなります。

▶　抽象メソッドをもたないクラスを抽象クラスとすることもできる（p.429）ため、*C* を抽象クラスとして定義することもできます。

**C.java**

```
abstract class A {
    abstract void a();
    abstract void b();
}

abstract class B extends A {
    void a() { /*…*/ }
}

class C extends B {
    void b() { /*…*/ }
}
```

メソッド a を実装。
抽象メソッド b を継承しているので、
このクラスも抽象クラス。

メソッド b を実装。
抽象メソッドはないため、抽象クラスとして宣言しなくてよい。

**Fig.13-4**　抽象クラスとメソッドの実装