計算機プログラミング I (9)

クラスの設計

2002 年 12 月 12 日　増原 英彦

Java 言語のクラスを定義には、

- 値を覚えるインスタンス変数を定義する
- 処理や計算を行うインスタンスメソッドを定義する
- 初期化を行うコンストラクタを定義する
- 他のクラスを拡張する

といった要素がある。実際のプログラムを作る場合に、それぞれの要素をどう定義してクラスを設計するかを、実例を通して見てゆこう。

1　複素数

複素数は $x + iy$ のように、実部と虚部の 2 つの数の組み合わせによって表わすことができる。複素数をオブジェクトとして表わせば、複素数という「1 つの数」がプログラムの中でも「1 つの値」として扱うことができる。

ここでは、複素数をオブジェクトとして表わす際に、どのようなクラスを設計し、コンストラクタやメソッドを定義するかを見てゆく。

1.1　クラスの定義

まず「複素数オブジェクトのクラス」を定義する。ここでは Complex という名前にする:

```
1 /** 複素数 */
2 class Complex {
3   インスタンス変数・インスタンスメソッド・コンストラクタの定義
4 }
```

1.2　インスタンス変数

複素数のベクトル表現は、実部 (real part)・虚部 (imaginary part) の 2 つの実数からなる。そこで、Complex クラスには、2 つの実数 (double) 型のインスタンス変数を定義する。

```
1 /** 複素数 */
2 class Complex {
3   public double real; // ベクトル表現の実部
4   public double imag; // ベクトル表現の虚部
5   インスタンスメソッド・コンストラクタの定義
6 }
```

これで $1 + 2i$ のような複素数を

```
Complex c = new Complex();
c.real = 1;
c.imag = 2;
```

のようにして作ることができるようになった。作られた複素数は 1 つのオブジェクトなので、`someMethod(c)` のような形で、他のメソッドの引数に渡すことができるようになる。(`c.real` は変数 c にしまわれている複素数オブジェクトの中にある、 real というインスタンス変数を表わす。)

1.3 コンストラクタ

コンストラクタを定義すると、 `new Complex(1,2)` のような式で $1+2i$ を表わす複素数オブジェクトを作ることができる。上の例では複素数オブジェクトを作った後、2 つのインスタンス変数に代入していた。コンストラクタがあれば

```
Complex c = new Complex(1,2);
```

のように、1 文で済む。さらに、`someMethod(new Complex(2,3))` のように、複素数を引数として渡す場合などに、一度変数にしまう必要もなくなる。

コンストラクタは下の 6–10 行目のように定義できる。`new Complex(1,2)` という式が計算されると、まず Complex クラスのオブジェクトが作られ、次に `r=1, i=2` として 7 行目のコンストラクタが呼び出される。コンストラクタでは、作られたオブジェクトの real (および imag) というインスタンス変数に r (および i) の値を代入している:

```
1  /** 複素数 */
2  class Complex {
3    public double real; // ベクトル表現の実部
4    public double imag; // ベクトル表現の虚部
5
6   /** 実部 (r), 虚部 (i) の成分から複素数を作るコンストラクタ */
7    public Complex(double r, double i) {
8      real = r;
9      imag = i;
10   }
11 }
```

1.4 インスタンスメソッド

オブジェクト指向言語では、複素数の計算 (足し算、掛け算など) をインスタンスメソッドとして定義するのが普通である。これは「計算方法の詳細はオブジェクト自身が知っており、その計算を使う側 (監督者) は気にしなくてよい」という考え方によるものである。

2 つの複素数 x と y を足すという計算をインスタンスメソッドとして定義する場合、一方の複素数オブジェクト (例えば x) に対して、「(もう一つの) 複素数 (y) を加えた値はいくらか?」と問い合わせる形式をとる。

```
1  /** 複素数 */
2  class Complex {
3    インスタンス変数・コンストラクタの定義 (略)
4    //インスタンスメソッドの定義
5    /** 自分自身に複素数 y を加えた複素数を作って返す */
6    public Complex add(Complex y) {
```

```
7     double r = real + y.real; //実部の計算
8     double i = imag + y.imag; //虚部の計算
9     return new Complex(r,i); //新しい複素数を作って返す
10   }
11 }
```

- 複素数オブェジェクトは「加えた値を計算せよ」という指示を受け取る。これが 6 行目の add という名前のメソッドである。
- 加えた値は、メソッドの返り値として返される。ここでは複素数なので add メソッドの返り値の型は Complex である (6 行目の 2 番目の単語)。
- 「加えた値を計算せよ」という指示には「どれだけ加えるか」という引数が必要である。ここでは加える数も複素数であるので、add メソッドの引数も Complex 型であり、変数名は y としている (6 行目の括弧内)。
- 複素数の足し算は、$(x_R + ix_I) + (y_R + iy_I) = (x_R + y_R) + i(x_I + y_I)$ である[1]。自身の実部 (`real`) と y の実部 (`y.real`) を足した値を求める (7 行目)。単に real と書けば、自分自身のインスタンス変数の値をとり出す。y.real のように書けば、引数として与えられたオブェクトの real というインスタンス変数の値をとり出すことになる。同様に虚部どうしを足した値も求める (8 行目)。それらの値を使って、新しい Complex オブジェクトを作り、return 文によって返している (9 行目)。

ここで定義された add メソッドは次のような形で使うことができる:

```
Complex c1 = new Complex(1,2);
Complex c2 = new Complex(3,4);
Complex c3 = c1.add(c2);
```

ここでは、$c_1 = 1 + 2i, c_2 = 3 + 4i$ として $c_3 = c_1 + c_2$ としている。式の書き方は数学的な書き方と違うが、ほぼそのまま書き直せることに注意せよ。

足し算の他にも、いくつかの演算が必要となる:

- 掛け算 multiply : $(x_R + ix_I)(y_R + iy_I) = (x_R y_R - x_I y_I) + i(x_R y_I + x_I y_R)$
- 絶対値 magnitude : $|x_R + ix_I| = \sqrt{x_R^2 + x_I^2}$
- などなど

これらは、add メソッドと同様の形で定義できる。

計算以外に、オブジェクトを文字列変換するインスタンスメソッドを定義しておくと、計算結果を表示する際に便利である。Java 言語では、toString という名前のメソッドを定義しておくと、オブジェクトを文字列に自動的な変換する際にこのメソッドを呼び出すことになっている。そこで、$1 + 2i$ のような複素数だったら「`1+2i`」のような文字列を返すようにこのメソッドを定義しておく。

```
1  /** 複素数 */
2  class Complex {
3    インスタンス変数・コンストラクタの定義 (略)
4    計算をするインスタンスメソッドの定義 (略)
5    /** 文字列に変換する */
6    public String toString() {
7      return real + "+" + imag + "i";
```

[1] x_R, x_I を複素数 x の実部、虚部とする。

```
  }
}
```

このように定義しておけば、

```
Complex c1 = new Complex(1,2);
System.out.println("c1 の値は" + c1 + "です。");
```

と書くだけで、「c1 の値は 1+2i です。」のように表示させることができる。

練習 9-1: (複素数)* 上で示した Complex クラスに、掛け算をする multiply、絶対値を求める maginitude を追加して複素数クラスを完成させよ。(足し算や掛け算の結果は複素数であり、返り値の型も Complex 型であるのに対し、複素数の絶対値は実数値であるので、返り値の型は double 型になるべきことに気をつけよ。)

さらに、$a = 1 + 2i, b = 3 + 4i, c = 5 + 6i$ としたときに、以下の値を計算し、表示させるようなプログラムを作り、手で計算した結果と比べよ:

(ア) $b + c$　(イ) ab　(ウ) ac　(エ) $ab + ac$　(オ) $a(b + c)$

(参考プログラム Complex.java, ComplexTest.java)

2 複素数オブジェクトの利用: マンデルブロ集合

マンデルブロ (Mandelbrot) 集合は、複素平面上に定義される関数を繰り返し適用したときに値が発散しない領域である。この領域の形は自己相似形になっていることが知られている。

複素数オブジェクトを使って、マンデルブロ集合を描くプログラムを作成してみよう。複素平面上の点 c に対して $f(z) = z^2 + c$ のような複素関数が定義されているとする。このとき、

$$\begin{aligned} f^0(z) &= z \\ f^1(z) &= f(z) \\ f^2(z) &= f(f(z)) \\ &\vdots \end{aligned}$$

のように、z に f を k 回適用した値を $f^k(z)$ と書くことにする。

マンデルブロ集合とは、$k \to \infty$ としたときに $|f^k(0)| \to \infty$ とならないような c の集合である。実際に $f^\infty(0)$ を求めることはできないが、$|f^k(0)| > 2$ となるときには、$|f^k(0)| \to \infty$ となることが知られているので、次のような近似をする:

- $f^0(0), \ldots, f^{k-1}(0)$ の絶対値が 2 以下で、かつ $|f^k(0)| > 2$ であったら、「k 回で発散」とする
- $f^0(0), \ldots, f^{30}(0)$ の絶対値が全て 2 以下だったら「発散しない」とする

このように近似したときに「発散しない」c の集合をマンデルブロ集合とする。

2.1 複素関数の追加

まず $f(z)$ を計算するメソッドを Complex クラスに追加する。実際の f は c に対して定義されているので、c.f(z) のようにメソッドを呼び出すと $z^2 + c$ を計算するようなメソッドを定義すればよいので、次のようになる:

```
public Complex f(Complex z) {
  return [z*z に自身を足した値を計算];
```

```
}
```

この定義では z^2 に c 自身を足す必要があるが、c は「そのオブジェクト自身」である。Java 言語では、オブジェクト自身は `this` という変数で表わすことができる。

さらに、c において $|f^k(0)| > 2$ となる回数 k を求めるインスタンスメソッドを追加する。このメソッドでは、$z = 0$ から出発し、$|z| \leq 2$ の間、z を $f(z)$ に変化させてゆくので、次のようになる:

```
 1  public int divergeNumber() {
 2      Complex z = new Complex(0,0);  // z を 0 にする
 3      int k = 0;
 4      while ( z の絶対値が 2 以下か? ) {
 5          if (k > 30) {              // f^30(0) まで計算しても発散しなかった
 6              return -1;             // 場合、繰り返しを中断し -1 を返す
 7          }
 8          次回の z を f(z) にする
 9          k++;
10      }
11      return k;                      // 発散に要した回数 k を返す
12  }
```

5–7 行目は、一定回数 (ここでは 30 回) 以上繰り返しても発散しないときは繰り返しを打ち切るための部分である。この定義において c は、このオブジェクト自身であるので、`f(z)` のように書くと、自分自身 (つまり c) における f を z に適用することができる。

2.2 集合を描く

マンデルブロ集合を描くには、まず画面上の点 (x, y) に対応した複素数 c を作る。そして c おいて $f^k(0)$ が発散するのに要した回数を求め、その回数に応じて点 (x, y) の色を決めればよい。

例えば 400×400 の画面を、X 方向が実部で $[-2, 0.5)$, Y 方向が虚部の $[-1.25, 1.25)$ に対応させるなら、次のようになる:

```
 1  public class Mandel {
 2    public static void main(String[] args) {
 3      int size = 400;
 4      double rmin = -2, rmax = 0.5, imin = -1.25, imax = 1.25;
 5      for (int x = 0; x < size; x++) {
 6        for (int y = 0; y < size; y++) {
 7          // x,y に対応した複素数を作る
 8          Complex c = new Complex(rmin+x*(rmax-rmin)/size,
 9                                  imin+y*(imax-imin)/size);
10          int d = c.divergeNumber(); //発散に要した回数を求める
11          if (d == -1)
12              点 (x,y) を黒で塗る
13          else
14              点 (x,y) を d に応じた色に塗る
15        }
16      }
17    }
18  }
```

練習 9-2: (マンデルブロ集合) Complex クラスにインスタンスメソッド f, divergeNumber を追加せよ。さらに、Mandel クラスを完成させ、マンデルブロ集合を描いてみよ。(参考: Mandel.java)

練習 9-3: (マンデルブロ集合の自己相似性) 実部・虚部の範囲を変えて、集合の一部を拡大した図を描いてみよ。上の練習で表示した全体の形の相似形が見られることを確認せよ。

練習 9-4: (他のマンデルブロ集合) 複素数関数 f を $f(z) = z^3 + c$ に変えたときに、集合の形がどのように変わるか? プログラムを作って作図させてみよ。

練習 9-5: (ジュリア集合) $f(z) = z^2 - a$ (a は定数) のときに $f^k(c)$ が発散しないような c の集合をジュリア集合という。マンデルブロ集合のプログラムを変更して、$a = 0.75$ として作図させてみよ。

練習 9-6: (オブジェクトを使う意義) インスタンスメソッド f および divergeNumber を、Complex オブジェクトを使わずに、実部・虚部を表わす 2 つの実数を使って計算するように書き換えて、プログラムの読み易さはどの程度変わるかを考えよ。

3 Morphing 再び

Morphing の課題では、円と三角形を描く 2 匹のタートルを用意し、もう 1 匹のタートルにそれらの中間地点をたどらせることで中間の図形を描かせていた。

クラスを拡張することで、「指定された 2 匹のタートルの中間の位置を辿る」ようなタートルを定義することができる。このクラスの名前を Morph をとしよう。以下、Morph クラスをどのように定義するかを見てゆく。

3.1 クラス定義

Morph は「2 匹を追いかける」ことができる以外は、普通のタートルオブジェクトと同様の機能を持つので、Turtle クラスの子クラスとして定義する。

```
1 public class Morph extends Turtle {
2   インスタンス変数・コンストラクタ・インスタンスメソッドの定義
3 }
```

3.2 インスタンス変数

位置を追いかける 2 匹のタートルは、インスタンス変数によって覚える。そこで、Turtle 型の変数 m1, m2 を用意する。

```
1 public class Morph extends Turtle {
2   public Turtle m1, m2; // 追いかける対象のタートル
3   コンストラクタ・インスタンスメソッドの定義
4 }
```

3.3 コンストラクタ

Morph オブジェクトが作られるときに「どのタートルの中間を追いかけるか」を指定できるようにコンストラクタを定義する。つまり、`new Morph(m1,m2)` のようにしてオブジェクトを作ると、m1 と m2 を追いかけ

る Morph オブジェクトが作られるようにする。コンストラクタは、2 つの引数を受け取り、上で定義したインスタンス変数に代入して覚える。

```
1 public class Morph extends Turtle {
2   public Turtle m1, m2; // 追いかける対象のタートル
3   public Morph(Turtle m1, Turtle m2) { // コンストラクタ
4     this.m1 = m1;
5     this.m2 = m2;
6   }
7   インスタンスメソッドの定義
8 }
```

3.4 インスタンスメソッド

2 匹のタートルの位置の中間点を追いかけるためのインスタンスメソッドを定義する。ここでは track というメソッドが呼び出されると、その時点での m1, m2 の中間点に移動するようにする。

```
1 public class Morph extends Turtle {
2   public Turtle m1, m2; // 追いかける対象のタートル
3   // コンストラクタ (略)
4   public void track() {
5     自分自身を m1 と m2 の中間点に移動させる
6   }
7 }
```

これで、次のように m1 と m2 が移動する度に Morph オブジェクトの track メソッドを呼び出せば、Morph オブジェクトは自動的に m1 と m2 の中間をたどり、結果として中間の図形が描かれる:

```
Turtle m1 = new Turtle();
Turtle m2 = new Turtle();
Morph m = new Morph(m1,m2);
for (int i = 0; i < ...; i++) {
  m1 を移動
  m2 を移動
  m.track();
}
```

練習 9-7: (Morph オブジェクト)* 上で示した Morph クラスを完成させよ。課題で作ったプログラムを Morph オブジェクトを使うように書き換えてみよ。

練習 9-8: (段階的な Morphing) 円と三角形を描くタートルを m_C, m_T とすると、上の練習では m_C, m_T の間をたどる Morph オブジェクト m_0 を作った。さらに、m_C, m_0 の間をたどる Morph オブジェクト m_1 と m_T, m_0 の間をたどる Morph オブジェクト m_1 を作り、3 段階の Morphing を行ってみよ。