計算機プログラミング I (8)

# 継承を使ったプログラムの設計

2003 年 12 月 5 日 増原 英彦

## 1 描画エディタ

1 のような画面を持ち、マウスを使って図形を描くことができる描画エディタを作ることを考える。

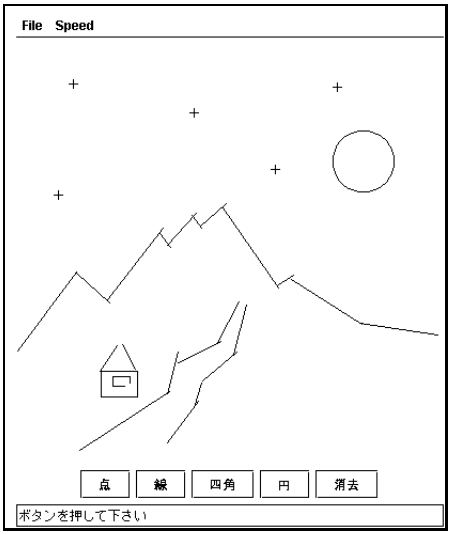


図 1: 描画エディタの画面例

### 1.1 機能

このエディタは次のような機能を持っている。

- 点・線・円などの種類の**図形要素**から成る図をマウスを使って対話的に描くことができる
- 作図は、(1) まず図形の種類をボタンによって選び、(2) 次に画面上の位置を 1〜2 回クリックして図形の形状を指定すると、画面上にその図形が描かれる。この手順を繰り返して図を完成させる。
- 消去ボタンを選ぶと全ての図形を消去することができる。

さらに、拡張機能として次のようなことができる:

**色の変更:** まず、「赤」「青」などのボタンを選んで、変更したい色を決め、次に画面上をクリックして色を変更したい図形要素を選ぶことで、選んだ図形の色を変更する。

**取り消し:** 「取消」ボタンを選ぶと、一番最後に描いた図形を消すことができる。

**削除:** まず、「削除」ボタンを選び、次に画面上をクリックすると、クリックした位置にある図形要素が消される。

**その他:** 図形要素の種類を増やす (e.g., 星形)、図形要素を後から移動できるようにする、形状を変更できるようにする、図形を回転する、等々の拡張も考えられる。

## 1.2 クラスの設計

描画エディタは、マウスからの入力を読みとり画面表示を行う**エディタ**部分と、画面に描かれる円や線を表わす**図形要素**に分けて考える。

エディタ部分は、図を描くこと、ボタンを選ぶこと、画面上のクリックされた位置を調べることができる必要がある。これらの機能は ClickableTurtleFrame オブジェクト (cf. 資料 web ページ) を使うことにする。また、作図は Turtle オブェクトによって実現できる。

円や線のような 1 つ 1 つの図形要素は、新たに定義したクラスのオブジェクトとして表わす。ここで、全ての図形要素に共通の性質・機能と、個々の図形要素に特有の性質・機能を考えてみる。すると

- 「全ての種類の図形要素は画面に表示できる」が、「どのような図形を描くか」は図形要素の種類ごとに異なる。(e.g., 円も線も「作図できる」点は共通である。しかし、Turtle オブジェクトを使って作図する手順を考えると、円は正多角形を描くし、線は二点間を移動することになるので、全く違うものとなる。)
- 図形要素はそれぞれ「図形の位置や形」の情報を持つが、どのような種類の情報を持つかは図形要素の種類ごとに異なる。(e.g., 円であれば中心の座標と半径、線であれば両端点の座標など)

そこで、次のようにクラスを設計する:

- 図形要素に共通した性質・操作を表わすクラス FigureElement を 1 つ定義する。
- 図形要素に共通した操作として、「タートル m を使ってその図形を画面に表示する」というメソッド`void draw(Turtle m)` を FigureElement クラスに定義する。
- 円や線などの各図形要素に応じて FigureElement の子クラスをそれぞれ定義する。
- それぞれの子クラスで`void draw(Turtle m)` を再定義して、m を使って円や線などの適切な図形を描く。

共通の親クラスとして定義する FigureElement クラスは、次のように、何も描かない「何もない図形」になる[1]。

```
1  public class FigureElement {    // 何もない図形要素
2    public void draw(Turtle m) {  // m を使って画面に図形を描く
3        // 何もないので何も描かない
4    }
5  }
```

このクラスを親クラスとして、点や直線は次のように定義される:

```
1  public class Point extends FigureElement { // 点図形要素
2    int x, y;                                // 点の座標値
3    int size = 4;                            // 表示サイズ
4    Point(int x0, int y0) { x = x0; y = y0; }// コンストラクタ
5    public void draw(Turtle m) {             // 点を表示する (十字を描く)
6      m.up();   m.moveTo(x+size,y,0);        // 亀を点の少し下に移動
7      m.down(); m.fd(size*2);                // 上方向へ線を描く
8      m.up();   m.moveTo(x,y-size,90);       // 亀を点の少し左に移動
9      m.down(); m.fd(size*2);                // 右方向へ線を描く
10   }
11 }
```

```
1  public class Line extends FigureElement { // 線図形要素
2    int startx, starty, endx, endy;         // 始点・終点の座標値
3    Line(int x1, int y1, int x2, int y2) {  // コンストラクタ
4        startx = x1; starty = y1;
5        endx   = x2; endy   = y2;
6    }
7    public void draw(Turtle m) {            // 線を表示する
8      m.up();   m.moveTo(startx,starty,0);  // 始点に移動し
9      m.down(); m.moveTo(endx,  endy,  0);  // ペンを下ろして終点に移動
10   }
11 }
```

[1] このようなクラスは実際にオブジェクトを作ることがないので、抽象 (abstract) クラスとして定義するのが一般的であるが、この授業では簡単のために、「何もしない」クラスとして定義している。

エディタ部分は、押されたマウスボタンに対応して図形要素オブジェクトを作り、その draw メソッドを呼び出して表示をする。概略は次のようになる:

```
public class Sketch {
    public static void main(String[] args) {
        ウィンドウを作る (略);
        Turtle m = new Turtle(); // 描画用のタートルを作る
        f.add(m);
        while (true) {
            if (「消去」ボタンが押された) {
                画面を消去する
            } else {
                FigureElement e = null;          // 作られる図形要素をしまう変数
                if (「点」ボタンが押された) {
                    int x = (略), y = (略);        // クリックされた場所を調べる
                    e = new Point(x,y);          // 新しく点を作る
                } else if (「線」ボタンが押された) {
                    int x1 = (略), y1 = (略);    // クリックされた場所を調べる
                    int x2 = (略), y2 = (略);    // クリックされた場所を調べる
                    e = new Line(x1,y1,x2,y2);   // 線を作る
                }
                e.draw(m);                       // 作った図形要素を表示する
            }
        }
    }
}
```

同様にして四角と円を表わす図形要素クラス Box, Circle などをそれぞれ定義し、Sketch クラスと組み合わせてゆけば基本的な図形エディタが完成する。

## 1.3 拡張機能: 取消

図形要素をオブジェクトとして表わしているので、「これまで描いた図形要素を全て記録する」ということも可能である。記録ができれば「一度画面を消去して、これまで描いた図形要素を全て描き直す」こともできる。さらに、「一度画面を消去して、これまで描いた図形要素のうち最後に追加されたもの以外をもう一度描き直」せば、「最後に描いた図を取り消」したのと同じ結果が得られる。

この拡張は、エディタ部分の変更だけで実現できる。

- エディタに描かれた全ての図形要素を覚えておく FigureElement 型の配列 figures を用意する。配列の大きさは適当に大きな値としておく。
- エディタに「今まで何個の図形が描かれたか」を覚えておく変数 numFigures を用意する。
- 新たに図形要素が追加されたときには、画面に表示すると同時に、図形要素オブジェクトを figures の numFigures 番目にも代入する。そして numFigures を 1 増やす。
- 「消去」ボタンが押されたときには、まず numFigures を 1 減らす。そして一度画面を消去し、figures の 0 から (numFigures − 1) 番目の図形要素を描き直す。

## 1.4 拡張機能: 削除

マウスで画面をクリックして図形を削除するには、エディタと図形要素の両方の変更が必要である。

まず、エディタの変更は、上の取消機能を次のように変更することになる:

- 削除ボタンが選ばれたら、画面上の位置をクリックしてもらう。
- figures の 0 から (numFigures − 1) 番目の図形要素の中から、クリックされた位置に一番近い図形要素を探す。その番号を i とする。

- figures の (numFigures − 1) 番目の図形要素を figures の i 番目に代入[2]し、 numFigures を 1 減らす。
- 画面を消去し、figures の 0 から (numFigures − 1) 番目の図形要素を描き直す。

上で、「クリックされた位置に一番近い図形要素を探す」ために、図形要素クラスを次のように拡張する:

- 座標値から図形要素までの距離を求めるメソッド `double distance(int x, int y)` を定義する。このメソッドは全ての図形要素が共通して持っている。しかし、どのようにして距離を求めるかは図形要素の種類ごとに異なる。
- そこで、FigureElement クラスに distance メソッドを定義する。このメソッドは便宜上のものなので、常に 0 を返すとしておけばよい。
- 各子クラスで distance メソッドを再定義する。距離をどう決めるかは色々であるが、図形の輪郭線からの距離が直感的だろう。例えば中心が $(u, v)$ で半径 $r$ の円であれば、円周からの距離、つまり、$\left|r - \sqrt{(x-u)^2 + (y-v)^2}\right|$ を距離とするのが一つの定義である。もっと単純には、図形要素の中心からの距離としてしまうのでもよい。

### 1.5 拡張機能: 色の変更

図形要素の色を変更できるようにするためには、エディタと図形要素の両方の定義を変更する必要がある。

まず全ての種類の図形要素は、共通して「色」という性質を持つことになる。これは、Color クラスのインスタンス変数 color を FigureElement に追加すればよい。このとき、はじめに図形要素が作られたときは黒色になるように適切な初期化式を書いておく必要がある。

図形要素を描くメソッド draw は、図形要素が持つ色で図を描かなければいけない。これは、各子クラスの draw メソッドの先頭で、Turtle オブジェクトの setColor メソッドを呼び出すように変更すればよい。

次に、マウスでクリックした図形要素の色を変更する必要がある。これは、まず、「削除」の場合と同様に、クリックした位置に最も近い図形要素を選び、選ばれた図形要素のインスタンス変数 color に新しい色を代入すればよい。

## 2 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 $f(c, z) = z^2 + c$ のような複素関数が定義されているとする。このとき、

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

のように、$z$ に $f$ を $k$ 回適用した値を $f^k(c, z)$ 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 $|f^k(c, 0)| \to \infty$ とならないような $c$ の集合である。実際に $f^\infty(c, 0)$ を求めることはできないが、$|f^k(c, 0)| > 2$ となるときには、$|f^k(c, 0)| \to \infty$ となることが知られているので、次のような近似をする:

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

[2] このようにすると、その後の「取消」が「最後に描いた図形の消去」でなくなってしまう。より正確には、$(i+1)$ 番目以降の要素を全て、1 つ若い番号にずらしてやるべきである。

- $f^0(c,0),\ldots,f^{30}(c,0)$ 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。

複素関数 $f$ としては $f(c,z)=z^2+c$ という型のものが有名だが、その他にも以下のようなものが面白い図形になることが知られている:

$$
\begin{aligned}
f(c,z) &= z^k+c \qquad (k \text{ は適当な整数}) \\
f(c,z) &= z^2+1/c \\
f(c,z) &= z^2+1/(c+0.25) \\
f(c,z) &= z^2+1/(c-1.40115) \\
f(c,z) &= z(1-z)(1+\sqrt{1+4c}) \\
f(c,z) &= z(1-z)/(1+\sqrt{1+4c}) \\
f(c,z) &= z(1-z)/(\sqrt{1+4c})
\end{aligned}
$$

また、マンデルブロ集合は「自己相似形」あるいは「フラクタル図形」である。マンデルブロ集合の一部を拡大すると、そこに集合全体に相似した形が見られ、その一部をさらに拡大してもまた、そこに集合全体に相似した形が見られる……といった性質がある。

## 2.1 プログラムの構成

マンデルブロ集合表示プログラムは、以下のクラスから成る。

**作図クラス DrawMandelbrot:** 複素関数を使って $f^k(c,0)$ が発散する回数 $k$ を求め、$c$ に対応する画面上の点に $k$ に対応した色を塗る。複素関数の計算は ComplexFunction オブジェクトを使う。

**複素関数クラス ComplexFunction:** 様々な複素関数 $f(c,z)$ をオブジェクトとして表わすためのクラス。継承によって異なる複素関数を定義する。複素数の計算は、Complex オブジェクトを使う。

**複素数クラス Complex:** 複素数の計算を容易にするために、1 つの複素数を 1 つのオブジェクトとして表わすためのクラス。

以下、逆順にクラスの定義を見てゆく。

## 2.2 複素数クラスの設計

**機能:** 複素数をオブジェクトとして表現する場合、その機能は次のようになる。

- 2 つの実数 $x,y$ から $x+iy$ を表わす複素数オブジェクトを作る (以下、$c,c',c''$ は複素数オブジェクトだとする)
- $c$ を文字列にする (あるいは画面に表示する)
- $c$ と $c'$ を足した複素数オブジェクトを求める
- $c$ と $c'$ を掛けた複素数オブジェクトを求める
- $c$ の絶対値を求める
- $c$ の実部、虚部をそれぞれ求める

**設計:** 上のような機能を持った複素数をオブジェクトとして表現するには、「複素数オブジェクトのクラス」を作り、以下をコンストラクタやメソッドとして定義すればよい:

- 2 つの実数 $x, y$ から $x + iy$ を表わす複素数オブジェクトを作る—2 つの実数を受け取るコンストラクタ
- $c$ を文字列にする—`toString` メソッド (後述)
- $c$ と $c'$ を足した複素数オブジェクトを求める—$c'$ を受け取って自身との和を表わす複素数オブジェクトを新たに作り、それを返すメソッド
- $c$ と $c'$ を掛けた複素数オブジェクトを求める—(同上)
- $c$ の絶対値を求める—自身の絶対値を返すメソッド
- $c$ の実部、虚部をそれぞれ求める—メソッドあるいはフィールド

**クラス定義:** 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:

```
/** 複素数 */
class Complex {
  [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]
}
```

**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。

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

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。)

**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:

```
/** 複素数 */
class Complex {
  public double real; // ベクトル表現の実部
  public double imag; // ベクトル表現の虚部

 /** 実部 (r), 虚部 (i) の成分から複素数を作るコンストラクタ */
  public Complex(double r, double i) {
    real = r;
    imag = i;
  }
}
```

**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[3]。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:

[3] $x_R, x_I$ を複素数 $x$ の実部、虚部とする。

```
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。

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

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 8-1: (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) $a(b + c)$

## 2.3 複素関数クラス

**設計:** 複素関数は $f(c, z) =$ 式 という形をしている。つまり、2 つの複素数を受け取り、複素数を答えとして返す。

マンデルブロ集合では、様々な複素関数を使用して様々なマンデルブロ集合を描きたい。様々な複素関数に共通する性質・操作と、個有のものを考えてみる。

**共通する操作:** 全ての複素関数は 2 つの複素数を受け取り、複素数を答えとして返す操作 (計算) を持っている。

**個有の操作:** その操作 (計算) は複素関数によって内容が違う。

そこで、複素関数を使のように継承を使って扱うことにする。

- 全ての複素関数に共通する親クラス ComplexFunction を定義する。
- 全ての複素関数に共通して存在する操作を`Complex calc(Complex c, Complex z)` というメソッドとする。このメソッドを ComplexFunction に定義する。
- 個々の複素関数に応じて ComplexFunction の子クラスを定義する。
- それぞれの子クラスの中でメソッド calc を再定義して、その複素関数特有の操作 (計算) を定義する。

**クラス定義:** ComplexFunction クラスは、上の $f$ のように 2 つの複素数を使って計算をするメソッド`Complex calc(Complex c, Complex z)` を持っているだけである。

```
/** 複素関数 */
public class ComplexFunction {
  public Complex calc(Complex c, Complex z) { //f(c,z) の定義
    return null; //「答えが無い」ことを表わすために null を返す
  }
}
```

実際の複素関数はこの子クラスとして定義する。例えば $f(c,z) = z^2 + c$ を Mandelbrot という名前で定義すると次のようになる:

```
/** 複素関数 f(c,z) = z^2 + c */
public class Mandelbrot extends ComplexFunction {
  public Complex calc(Complex c, Complex z) { //f(c,z) の再定義
    return [z^2 + c の計算];
  }
}
```

この複素関数はオブジェクトとして代入することができるので、`ComplexFunction f = new Mandelbrot();` のようにして関数を作り、`f.calc(c,z)` のようにして計算をすることができる。

## 2.4 作図クラスの定義

**機能:** クラス DrawMandelbrot の基本的な機能は、表示用のフレームを作り、複素関数を用意し、各点について複素関数を用いた計算結果に応じた表示する。

**設計:** DrawMandelbrot は、継承を作ったりオブジェクトを作る必要はなく、単純に main メソッドの中で処理をすればよい。(もちろんオブジェクトとして定義した方がよい拡張も考えられる。)

表示の詳細については、資料 web ページに掲載した ColorFrame クラス[4]を使うことにする。

中心となる処理は次のようになる:

- 画面上の各点について、その点に対応する複素数 $c$ を作る。
- 各 $c$ について $f^k(c,0)$ が発散するのに要する回数を求める。
- 回数に応じて、画面上の点に色を塗る。

「$f^k(c,0)$ が発散するのに要する回数を求める」のは、少々複雑なので、これをクラスメソッドとして別に定義することにする。

[4] このクラスは TurtleFrame の子クラスで、 Turtle を使って画面上に点を打つことができるようにしただけのものである。

**定義:** 「発散に要する回数」を`int divergeNumber(ComplexFunction f, Complex c)`というクラスメソッドで求めることにすれば、以下のような定義になる:

```
1  /** マンデルブロ集合の描画 */
2  public class DrawMandelbrot {
3    public static void main(String[] args) {
4      int size = 400;                // 画面の大きさ
5      double rMin = -2, rMax = 0.5;  // 複素平面の範囲 (実部)
6      double yMin = -1.25, yMax = 1.25; //  〃 (虚部)
7
8      ColorFrame frame = new ColorFrame(size, size); //表示用の画面を作る
9      ComplexFunction f = new Mandelbrot(); //複素関数を用意する
10
11     for (int x = 0; x < size; x++) {
12       for (int y = 0; y < size; y++) {
13         Complex c =           // x,y に対応する複素数 c を作る
14           new Complex(rMin + x * (rMax - rMin) / size,
15                       yMin + y * (yMax - yMin) / size);
16
17         int k = divergeNumber(f,c);  //発散に要した回数を求める
18         frame.plot(x, y, k);         //回数に応じて x,y に色を塗る
19       }
20     }
21   }
22   divergeNumber の定義 (後述)
23 }
```

9 行目で関数オブジェクトを作っていることに注意せよ。発散に要する回数を求める divergeNumber は次のように定義できる:

```
1  /** |f^k(c,0)| > 2 なる最小の k を求める; f^k(c,0) が発散しない場合は-1 を返す */
2  public static int divergeNumber(ComplexFunction f, Complex c) {
3    Complex z = new Complex(0,0); // z を 0 にする
4    int k = 0;                // 発散に要した回数を 0 にする
5    while ( z の絶対値は 2 以下か? ) {
6      if (k > 30) {           // f^30(c,0) まで計算しても発散しなかった場合、
7        return -1;            // 繰り返しを中断し -1 を返す
8      }
9      次回の z を f(c,z) にする
10     k++;
11   }
12   return k;                 // 発散に要した回数 k を返す
13 }
```

6–8 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。複素関数がオブジェクトになっているので、$f(c,z)$ を求めるためには、オブジェクト`f` のメソッド`calc` を呼び出さなければいけないことに注意せよ。

## 2.5 拡張機能

- $f(c,z) = c + z^3$ や $f(c,z) = c + z^4$ のような複素関数を定義して、それによってどのような図形が描かれるかを見る。具体的には ComplexFunction の別の子クラスを定義し、DrawMandelbrot クラスの`new Mandelbrot()` の部分を変更すればよい。

- $f_n(c,z) = c + z^n$ という複素関数クラス MandelbrotN を定義する。この場合、$n$ は MandelbrotN クラスのインスタンス変数として表わすのがよい。

- 色々な複素関数の定義を用意しておき、キーボードからの入力によって計算に用いる関数を選べるようにする。例えば DrawMandelbrot の 9 行目を`ComplexFunction[] fs = { new Mandelbrot(), new Mandelbrot2(), ... };` のように、配列に置き換え、17 行目の f を `fs[i]` とすればよいだろう。(i は入力された番号。)

  (ColorFrame の親クラスを ClickableTurtleFrame に変更することで、ボタンで選ぶようにしてもよい。あるいは、全ての複素関数を順に表示してゆくものを作ってもよい。)

- 実部・虚部の範囲を変えて、集合の一部を拡大した図を描く。集合の表示範囲は DrawMandelbrot の 5–6 行目で指定されている。

  複素関数によって表示すべき範囲が違うので、これらの値は DrawMandelbrot の main メソッドの中の変数ではなく、 ComplexFunction のインスタンス変数として定義すべきものかも知れない。

  拡大部分を指定するには、ColorFrame の親クラスを ClickableTurtleFrame に変更することで、マウスによって拡大範囲を指定できるようにすることも考えられる。

- $f(z) = z^2 - 0.75$ ($a$ 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。ジュリア集合とマンデルブロ集合の両方を作図できるようなプログラムを作るにはどのようなクラスを作ればよいだろうか? そのようなプログラムを作って $a = 0.75$ として作図させてみよ。

  (ヒント: $z$ の初期値が 0 であるかか $c$ であるかが違うだけであるので、ComplexFunction クラスに「$z$ の初期値」を求めるようなメソッドを定義するのが一つの方法。)