Java による UI プログラミング: MICS 実験 2-19-1c
＃03 アニメーション

久野　靖 *

2019.10.14

1　基本的なアニメーション

1.1　アニメーションの原理

アニメーションとは「時間とともに (人間が手を下さなくても) 変化する絵」を意味します。アニメーションの原理は「少しずつ違った絵を短い時間間隔で次々に見せると動いて見える」ことだというのはご存知と思います。今回はこれをプログラムで実現してみましょう。

なお、単にアニメーションと言った場合は、少しずつ違う画像を多数生成して、後からそれを順次再生する、というものも含まれます (図 1 上)。しかしここでは、「人間が見ているその時々に絵を生成する」方法を取ります。これを**実時間アニメーション**と言います。ゲームのように「人間が操作すると動きがそれに応答する」ためには、実時間アニメーションであることが前提になります (図 1 下)。

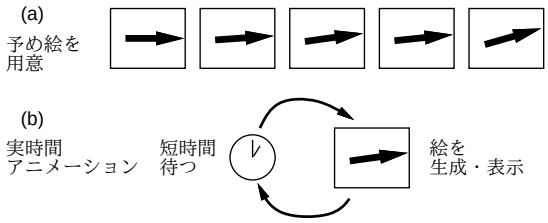

図 1: アニメーションの原理

1.2　タイマーオブジェクトの使用

実時間アニメーションのためには、短い時間間隔ごとにコードを動かし、「その時点に対応した絵」を生成して表示する必要があります。そのために `javax.swing.Timer` というクラスを次のように使います。[1]

```
new javax.swing.Timer(30, 動作).start(); // 30 ミリ秒間隔
```

`Timer` オブジェクトは生成時に「時間間隔」「動作」を指定し、`start()` が呼ばれると指定時間間隔ごとに「動作」を実行させます。「動作」としては、`ActlinListener` インタフェースを実装したオブジェクトを渡し、このインタフェースで定めている唯一のメソッドである `actionPerformed` が動作として呼び出される、という形になります。この動作のクラスも、前回のイベントの時と同様、1 箇所でしか使わないクラスになるので、無名内部クラスの形にして次のように定義することにします。

*電気通信大学

[1] Java の標準 API には `java.util.Timer` というクラスもあるため、ここではパッケージ名まで含めたフルクラス名で指定しています。

```
new javax.swing.Timer(30, new ActionListener() {
  public void actionPerformed(ActionEvent evt) {
    // 動作…
  }
}).start();
```

1.3 例題: 円をまるく動かす

では最初の例題として「円をまるく動かす」というのを見てみましょう。円のクラスは前回同様のものを使います。時間につれた動きを定めるには、「プログラム実行開始から何秒たっているか」(実数値) を求めて変数 time に入れ、それを使います。時間そのものは、System.currentTimeMillis() によって「1970 年 1 月 1 日 0 時 0 分 0 秒」からの経過ミリ秒数が取得できるのを利用します。このメソッドの返値は倍精度整数 long であり、プログラム実行開始時の値を変数 baseTime に格納しておきます。

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

public class Sam31 extends JPanel {
  Circle c1 = new Circle(Color.RED, 100, 50, 30);
  long baseTime = System.currentTimeMillis();
  double time = 0.0;

  public Sam31() {
    setOpaque(false);
    new javax.swing.Timer(30, new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        time = 0.001 * (System.currentTimeMillis()-baseTime);
        int x = (int)(200*Math.cos(time) + 250);
        int y = (int)(100*Math.sin(time) + 150);
        c1.moveTo(x, y); repaint();
      }
    }).start();
  }
  public void paintComponent(Graphics g) { c1.draw(g); }
```

コンストラクタ中で上記のようにタイマーを起動しますが、一定時間間隔ごとの動作の先頭で現在時刻から baseTime をさし引いて秒単位に直し time に入れます。そのあと、その time をもとに円の中心位置を cos、sin 関数で求め、その位置に円を移動してから再表示します。repaint() を呼ぶとやがて paintComponent() が呼び出され、円が所定位置に描かれるわけです。main や Circle についてはこれまでと変わっていません。

```
  public static void main(String[] args) {
    JFrame app = new JFrame();
    app.add(new Sam31());
    app.setSize(600, 600);
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
    app.setVisible(true);
  }
  static class Circle {
    Color col;
    int xpos, ypos, rad;
    public Circle(Color c, int x, int y, int r) {
      col = c; xpos = x; ypos = y; rad = r;
    }
    public void moveTo(int x, int y) { xpos = x; ypos = y; }
    public void setColor(Color c) { col = c; }
    public void draw(Graphics g) {
      g.setColor(col);
      g.fillOval(xpos-rad, ypos-rad, rad*2, rad*2);
    }
  }
}
```

演習 1 例題をそのまま動かせ。円の色や動き方 (とくに周回の速さ) を変更してみよ。納得したら以下のことをやってみよ。

a. 円でなく「長方形の周囲に沿って」動くようにする。
b. 時間とともに色が変化していくようにする。
c. 円以外の図形を動かす。
d. 2 つ以上の図形が動くようにする。
e. 2 つ以上の図形がそれぞれ違う動きをするようにする。

2 動きの部品化と組合せ

前節の方法で確かに図形を動かすことはできますが、「さまざまな図形をさまざまな動き方で」動かすことはかなり面倒そうです。そこで、図形をいくつでも `fset` に入れて統一的に操作するようにしたのと同様のやりかたで、「動き」もそれぞれをオブジェクトとして扱うように変更してみます。

具体的には、「動かす」機能は次のインタフェースに従うようにします。たったこれだけ? 動かす図形はどこに? と思うかも知れませんが、図形そのものは `Mover` オブジェクトを作るときに渡して内部で保持させればよいので、必要なのは「どの時点の位置/形/色にしなさい」と指示することだけなのです (`getFigure()` は内部に保持する図形オブジェクトを参照する必要があるときだけ必要で、動かすだけなら無くても大丈夫ですが、後の例題のために入れてあります)。

```
interface Mover {
  public void setTime(double t);
  public Figure getFigure();
}
```

これだけではイメージが湧かないと思いますので、例題に進みましょう。これまでは変数として図形の集まり `fset` を持っていましたが、それに加えて図形を動かすための `Mover` オブジェクトを格納した `mset` も用意します。時間を進めていくやりかたは最初の例題と々なので、`baseTime` と `time` もあります

```
import java.awt.*;
```

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

public class Sam32 extends JPanel {
  ArrayList<Figure> fset = new ArrayList<Figure>();
  ArrayList<Mover> mset = new ArrayList<Mover>();
  long baseTime = System.currentTimeMillis();
  double time = 0.0;
  public Sam32() {
    setOpaque(false);
    Figure f1 = new Circle(Color.blue, 0, 0, 30); fset.add(f1);
    Mover m1 = new CircleMover(f1, 200, 300, 150, 50, 0.3); mset.add(m1);
    Figure f2 = new Rect(Color.green, 0, 0, 40, 30); fset.add(f2);
    Mover m2 = new CircleMover(f2, 250, 400, 40, 80, 2.0); mset.add(m2);
    new javax.swing.Timer(30, new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        time = 0.001 * (System.currentTimeMillis()-baseTime);
        for(Mover m: mset) { m.setTime(time); }
        repaint();
      }
    }).start();
  }
```

コンストラクタの中では 2 つの円をこれまで通り作りますが、それに加えてこれらの円をそれぞれ異なる楕円軌道で動かす Mover である CircleMover を作り、これらは mset に入れます。そして、タイマーで反復実行される部分では、mset に格納されているすべての Mover についてそれぞれ時間を進めるようにしています。paintComponent と main についてはこれまでと変わりません。

```
  public void paintComponent(Graphics g) {
    for(Figure f: fset) { f.draw(g); }
  }
  public static void main(String[] args) {
    JFrame app = new JFrame();
    app.add(new Sam32());
    app.setSize(600, 600);
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.setVisible(true);
  }
```

ではいよいよ Mover を見てみましょう。コンストラクタで渡された値を覚えるとか getFigure で内部で覚えておいた図形を返すとかは普通です。重要なのは setTime ですが、これは時刻 t に対して円の中心を $(c_x + r_x \cos t, c_x + r_y \sin t)$ に設定することで楕円軌道に沿って動かします。

```
  interface Mover {
    public void setTime(double t);
    public Figure getFigure();
```

```
}
static class CircleMover implements Mover {
  Figure fig;
  double cx, cy, rx, ry, a;
  public CircleMover(Figure f, double cx1, double cy1,
                     double rx1, double ry1, double a1) {
    fig = f; cx = cx1; cy = cy1; rx = rx1; ry = ry1; a = a1;
  }
  public void setTime(double time) {
    int x = (int)(rx*Math.cos(a*time) + cx);
    int y = (int)(ry*Math.sin(a*time) + cy);
    fig.moveTo(x, y);
  }
  public Figure getFigure() { return fig; }
}
```

ここから先は前回学んだ図形オブジェクトそのままなのでとくに説明は不要と思います。hit() は後の例題で使うのでそのまま入れてあります。

```
interface Figure {
  public boolean hit(int x, int y);
  public void moveTo(int x, int y);
  public void setColor(Color c);
  public void draw(Graphics g);
}
static abstract class BasicFigure implements Figure {
  Color col;
  int xpos, ypos;
  public BasicFigure(Color c, int x, int y) {
    col = c; xpos = x; ypos = y;
  }
  public void moveTo(int x, int y) { xpos = x; ypos = y; }
  public void setColor(Color c) { col = c; }
}
static class Circle extends BasicFigure {
  int rad;
  public Circle(Color c, int x, int y, int r) {
    super(c, x, y); rad = r;
  }
  public boolean hit(int x, int y) {
    return (x-xpos)*(x-xpos) + (y-ypos)*(y-ypos) <= rad*rad;
  }
  public void draw(Graphics g) {
    g.setColor(col);
    g.fillOval(xpos-rad, ypos-rad, rad*2, rad*2);
  }
}
```

```
  static class Rect extends BasicFigure {
    int width, height;
    public Rect(Color c, int x, int y, int w, int h) {
      super(c, x, y); width = w; height = h;
    }
    public boolean hit(int x, int y) {
      return xpos-width/2 <= x && x <= xpos+width/2 &&
             ypos-height/2 <= y && y <= ypos+height/2;
    }
    public void draw(Graphics g) {
      g.setColor(col);
      g.fillRect(xpos-width/2, ypos-height/2, width, height);
    }
  }
} // 外側クラスの「}」
```

演習 2 例題をそのまま動かせ。図形の数をさらに増やしてみよ。動き方や色も変えること。納得したら以下のことをやってみよ。

a. 時刻 t_1 に (x_1, y_1) にいて、それから時刻 t_2 までの間に一定速度で (x_2, y_2) に直線的に移動させる `LinearMover` を作る。

b. 時刻 t まではこれまでの位置にとどまり、t になったら (x, y) に瞬時に移動させる `WarpMover` を作る。

c. $[0, 1)$ の一様乱数を得るメソッド `Math.random()` を利用して、ランダムウォークを行う `Mover` クラスを作る。

d. 色が時間とともに連続的に変化したり、またはある時刻に瞬時に新しい色に変化するような `Mover` クラスを作る。

e. これまでは図形は「常に表示」されていたが、`Figure` インタフェースにメソッド `void setVisible(bool v)` を追加して、「見える」「見えない」を切替え可能にする。その上で、「ある時刻になったら見えるようになる/見えないようになる」`Mover` クラスを作る。

f. 円の半径を (連続的に、ないし瞬時に) 変化させる円専用の `Mover` クラスを作る。

g. その他好きな `Mover` クラスを作る。

h. これらを組み合わせて、ストーリー性のある動画を表示させる。(例: 緑の円がふらふらしていると、赤い円が画面外からやってくる。2 人はひかれ合い、より沿って去っていく。)

演習 3 これまで、位置を動かす `Mover` クラスは複数同時には使用できなかった (最後に設定した位置だけが効果を持つから)。これを改め、複数の動きが (円軌道を描きながらふらふら動くなど) 合成されるようにしてみよ。

演習 3 は図形や `Mover` の設計をかなり変える必要があります。たとえば、図形にメソッド `getX()`、`getY()` を追加し、`Mover` は取得した XY 座標を元にそこから相対移動する形にするなどです。さらに、一連の `Mover` を適用すると位置が変わってしまうので、最初の位置を別に覚えておいて、描画が終ったらまた最初に位置に戻すことが必要になると思われます。

演習 4 複合図形オブジェクトに対して時間とともにその全体の形 (個々の部品の相対位置や大きさや形) が変化すると楽しそうである。そのようなことが可能になるためにはどうしたらよいか検討し実現してみなさい。

演習 4 はいくつかやりかたがあると思います。複合図形の 1 つずつに Mover をつけることも考えられますが、設定が繁雑そうです。たとえば「どのように動かす」というパラメタをいくつか用意して、そのパラメタを設定する Mover をくっつける、などの方法はどうでしょうか (これ以外の方法でももちろん構いません)。

3 イベント・ドラグとアニメーションの併用

このように絵がいろいろ動くようになりましたが、そのかわり前回扱った「選択してドラグする」機能はなくなっていました。それは寂しいので、両方とも使えるようにする、というのが次のお題です。その場合の問題は、「ドラグと Mover による動きが干渉する」ことです。それに対処する方針として、次のものが考えられます。

(1) ドラグ対象となっている図形だけ Mover を適用しない

(2) 図形に「動かす/動かさない」というフラグを設け、Mover 側でそのスイッチが ON のものだけ動かす

ここでは図形側の改修が必要ない (1) で実現してみます。前回やったのと同様、ドラグ対象の図形を保持する変数 target を追加しています。動かす部分で、Mover が動かそうとする図形を getFigure() で取得し、それが target と同じものであれば setTime() を実行しません (したがって動きません)。

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

public class Sam33 extends JPanel {
  ArrayList<Figure> fset = new ArrayList<Figure>();
  ArrayList<Mover> mset = new ArrayList<Mover>();
  long baseTime = System.currentTimeMillis();
  double time = 0.0;
  Figure target = null;

  public Sam33() {
    setOpaque(false);
    Figure f1 = new Circle(Color.blue, 0, 0, 30); fset.add(f1);
    Mover m1 = new CircleMover(f1, 200, 300, 150, 50, 0.3); mset.add(m1);
    Figure f2 = new Rect(Color.green, 0, 0, 40, 30); fset.add(f2);
    Mover m2 = new CircleMover(f2, 250, 400, 40, 80, 2.0); mset.add(m2);
    new javax.swing.Timer(50, new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        time = 0.001 * (System.currentTimeMillis()-baseTime);
        for(Mover m: mset) {
          if(m.getFigure() != target) { m.setTime(time); }
        }
        repaint();
      }
    }).start();
    addMouseListener(new MouseAdapter() {
```

```
    @Override
    public void mousePressed(MouseEvent evt) {
      target = null;
      for(Figure f: fset) {
        if(f.hit(evt.getX(), evt.getY())) { target = f; }
      }
      if(target == null) { return; }
      fset.remove(target); fset.add(target); repaint();
    }
    public void mouseReleased(MouseEvent evt) {
      target = null;
    }
  });
  addMouseMotionListener(new MouseMotionAdapter() {
    @Override
    public void mouseDragged(MouseEvent evt) {
      if(target == null) { return; }
      target.moveTo(evt.getX(), evt.getY()); repaint();
    }
  });
}
// 以下はまったく同じ
```

マウスボタン押し下げ時は前回同様 `hit()` で対象を探して `target` に入れます。そして、ボタンを離した時は `target` に `null` を入れます (これをやらないとどうなると思いますか?)。ドラグ時は前回とまったく同じです。わりと簡単でしたね。

演習 5 例題は方針 (1) で作ってあったが、方針 (2) だとどうか。実装してみて、(1) との優劣を検討しなさい。

演習 6 このやり方だと確かに「ドラグできて任意の位置に動かせる」が、離したとたんに `Mover` がもともとのやり方で動かすためドラグした効果が残らない。ドラグした効果が残るように改良してみよ。

演習 6 はどうしたらできるでしょうか。ドラグしたということは結果だけ見れば (dx, dy) だけ位置を相対移動したわけですから、当該図形を動かしている `Mover` の中で持つ座標も同じように修正すればよいことになります。または別案として、演習 3 に関して説明したように、図形には本来の位置があって Mover は相対移動させる、と言う形に全体ができてれば、その本来の位置を修正することで対応できますね。

演習 7 前節の内容と組み合わせ、「ドラグすることで部品の相対位置が変化させられるような (もしかしたらそれ自体形が時間とともに変わる、そして全体としてもドラグできる) 複合図形」を実現してみなさい。

演習 7 はけっこう大変そうですが、できたら楽しいと思います。やりかたは各自考えてみてください。