情報科学特殊講義'2000 ＃4

久野　靖*

2000.1.24-27

0　はじめに

アプレット作りはいかがでしたか? 課題内容が自由だったので今回は「前回の問題の解説」が不要ですから、以前ちょっと触れた「Java の構文」をまとめてから手続きの復習をして、その後で手続きをいろいろ書く練習をして頂きます。そしておしまいの方で、再帰手続きについて学ぶことにしましょう。その後は時間があれば再度、自由にお絵描きをしてもらいます

なお、今回からしばらくの間、題材そのものはアプレットによるグラフィクスを利用して行きます。というのは、その方が「楽しい」と思って頂ける方が多いようですので。ただし、あくまでも講義の目的は Java によるプログラミングの各種側面を学んで行くことであり、出てくるグラフィクスはその「副作用」であって「主たる目的」ではないことを忘れないように。(そうでないと、せっかく Java を学んだのにアプレットしか作れないプログラマになってしまいますから。)

1　Java の構文

プログラムの正確な書き方は、その言語の「文法」によって定められている。一応、今回はクラスからはじめてひととおり書き方を並べておく (まだ説明していないものもある)。なお「[...]」は「あってもなくてもよい」、「|」は「または」を表す。

```
クラス    ::= [public] class クラス名 継承指定 { 変数… メソッド… }
継承指定 ::= [extends クラス指定] [implements クラス指定,…]
変数      ::= [public] [static] 型指定 変数名 [= 式];
メソッド ::= [public] [static] 型指定 メソッド名 (宣言,…) [例外] { 文… }
```

ここまでのところは「クラスの中には変数やメソッドが入る」「メソッドの中には文が入る」という構造を表している。ここまでで出てくる宣言や型の規則は次の通り。

```
宣言      ::= 型指定 変数名
型指定    ::= 型名 | 型名 [] | void
型名      ::= クラス名 | boolean | byte | char | int | long | float | double
例外      ::= throws クラス名 [, クラス名…]
```

さて、文はこれまでにやったように、宣言、式 (代入文も式のうち)、if、while、for などがある。また今回新しく return が登場するがあとで説明。

```
文      ::= 宣言 [= 式]; | 式; | if(式) 文 [else 文] | while(式) 文
          | for([宣言 =] 式; 式; 式) 文 | { 文…}
          | break; |  continue; | return [式];
```

*筑波大学大学院経営システム科学専攻

式は演算子、オブジェクト生成、メソッド呼び出し、リテラルなどからなる。なお、配列オブジェクトを生成するとき、その要素の値を直接書いて初期設定する書き方もある。

```
式      ::= 式 演算子 式 | 演算子 式 | 式 [式] | new クラス名 (式,…)
          | new 型名 [式] | new 型名 []{式,…}
          | クラス名. メソッド名 (式,…) | 式. メソッド名 (式,…)
          | リテラル | 変数名 | クラス名. 変数名 | 式. 変数名 | ( 式 )
リテラル::= 整定数 [l] | 実定数 [f] | "..." | '...'
```

整数のリテラルは「l」がついたのは long、そうでないのは int。実数のリテラルは「f」がついたのは float、そうでないのは double。なお、演算子については先に挙げた。「a = b;」の「=」はあくまでも代入演算子であることに注意。演算子の一覧も再掲しておく。

四則演算	+	-	*	/	%	←剰余
ビット演算	&	|	^	~		
シフト演算	<<	>>				
論理演算	&&	||	!			
比較演算	==	!=	<	>	<=	>=
代入演算	=	+=	-=	*=	…	
増減演算	++	--				

2 整数と実数の間の行き来

実数と整数の変換方法についてまとめておく。整数 (int) から実数 (double) に変換するときは次の方法があるのだった。

- (new Integer(整数値)).doubleValue() — やっぱりちょっと長いかな?
- (double) 整数値 — キャスト。やっぱりこれが簡単でいいかも。
- double の値と混ぜて演算する — ちょっとお行儀悪い。
- double の変数に入れ直す — 入れるついでがあるのならいいけど。

しかし逆に実数 (double) から整数 (int) にする場合は次の 2 つしか使えない。

- (new Double(実数値)).intValue() — やっぱりちょっと長いかな?
- (int) 実数値 — キャスト。やっぱりこれが簡単でいいかも。

なぜかというと、実数から整数に変換するということは、小数点以下の情報が失われるのだから、キャストなど「何をやっているか明確に指定する」書き方だけが使えるようになっているのであった。

3 復習: メソッド

次は前回ちょっと喋ったメソッドの復習。メソッドというのは、「一連の動作」(これまで散々やった計算とか代入とか枝分かれとか繰り返しとか) をひとまとめにしたものである、ということでしたね。そして、従来は `main()` とか `paint()` というメソッドで「自分のやりたい動作」を書くことだけでプログラミングしてきたが、これからもっと動作が込み入って来ると、自分の書く動作も部分部分ごとにメソッドに分けて作った方がよくなる。その理由:

- 込み入った一連の動作を何回もやるとき、メソッドは 1 回定義するだけで何カ所からでも利用できる。
- 一連の動作を延々と書き綴るよりも、要所要所ごとにまとめて書き表した方が考えやすく (間違えにくく)、後で読みやすい。

たとえば、用事があって函館市内の知人のところへ行くとしたら、その手順をおおまかに

- 青森へ行く
- 青森から JR かフェリーで函館へ行く
- 函館駅/ターミナルからバスや市電で友人宅へ行く

と考えて、それぞれの内容は (たとえば最初のだったら、弘前駅まで行って JR の青森行きに乗るか、またはバスターミナルから直通の青森行きに乗るなど) また別に考えた方が分かりやすいですね?

このように、「まず大まかな計画を立て、その各ステップをそれぞれまた大まかな手順に分解して、…」という風に段階的に細かいところを考えるようにするのを「段階的詳細化」とか「トップダウン (アプローチ)」などと呼び、プログラミングで効果的な (知っておいて損のない) やり方の 1 つである。そして、その各段階をメソッドという形で別けておくと、自分が考えた手順がプログラムにそのまま反映できて分かりやすくなる。

たとえば、「山が 3 つ連なった絵を描くアプレット」を作ることにする。その疑似コードを次のように考える。

- 山その 1 を奥の方に濃い色で描く。
- 山その 2 をその手前にやや薄い色で描く。
- 山その 3 をそのまた手前にもっと薄い色で描く。

そして、「山を描く」部分は別にメソッドとして考えるわけである。

- 点 (x0,y0)(x1,y1)(x2,y2) を頂点として指定された色で三角を描く:
- 大きさ 3 の配列 xpos を用意し、値 x0、x1、x2 を格納。
- 大きさ 3 の配列 ypos を用意し、値 y0、y1、y2 を格納。
- 塗り潰しの色を指定された色に設定。
- 配列 xpos、ypos で指定された座標の三角形を塗り潰す。

これを Java アプレットにしたもののソースコードを示す。なお、前回は「大きさ 3 の配列を用意して 3 回値を入れる」書き方をしていたが、今回は「初期値指定で配列を作成する」書き方を使ってみた。

```
import java.applet.Applet;
import java.awt.*;

public class R4Sample1 extends Applet {
  public void paint(Graphics g) {
    sankaku(g, new Color(200, 110, 255), 120,140,280,140,200,20);
    sankaku(g, new Color(180, 150, 255), 80,160,240,160,160,30);
    sankaku(g, new Color(160, 190, 255), 40,180,200,180,120,40);
  }
  void sankaku(Graphics g, Color c,
      int x0, int y0, int x1, int y1, int x2, int y2) {
    int[] xpos = new int[]{x0, x1, x2};
    int[] ypos = new int[]{y0, y1, y2};
    g.setColor(c); g.fillPolygon(xpos, ypos, 3);
  }
}
```

疑似コードも Java コードも見通しがよいでしょう? なお、メソッドのに対しては「三角形の座標」「色」などをそのつど変えて渡す必要があり、これを「引数」と呼ぶのでしたね。その説明の図を再掲しておきます。

あともう 1 つついでですが、HTML ファイルも一応のせておきます (もう十分分かっていると思いますが)。

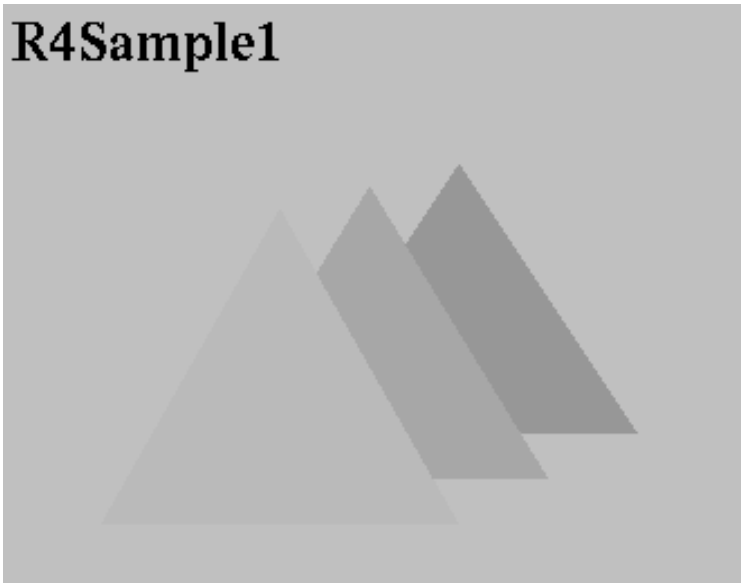

図 1: 例題 1 の表示

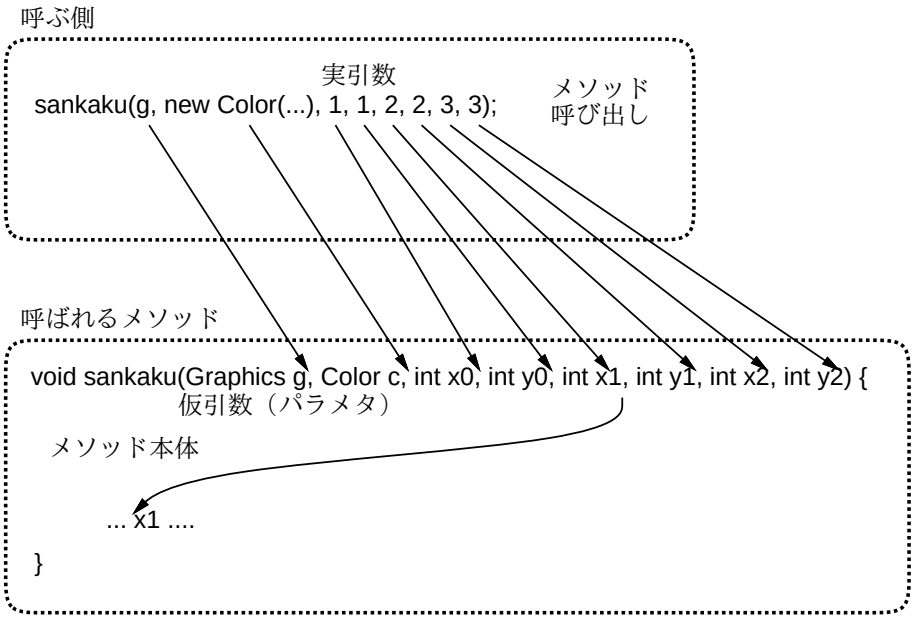

図 2: メソッドとパラメタ

```
<html>
<head><title>sample</title></head>
<body>
<h1>r4sample1</h1>
<applet code="R4Sample1.class" width=300 height=200></applet>
</body>
</html>
```

演習 1 次のようなアプレットを作り表示せよ。

a. 中心の座標 (x0,y0) を受取り、その位置に日の丸を描く手続きを作って、それを呼び出して日の丸を適当な場所に 3 つ描くアプレット。ただし全体の大きさは横 80 ピクセル、縦 60 ピクセル、赤い丸の直径が 40 ピクセルとする。(ヒント: Graphics クラスの fillRect() と fillOval() を使えばできます。矩形を描いてその後で丸を描くこと。)

b. 中心の座標 (x0,y0) と赤い丸の直径 r0 を受け取り、その位置に日の丸を描く手続きを作って、それを呼び出して 3 つ違う大きさの日の丸を描くアプレット。ただし全体の大きさは縦が r0 の 1.5 倍、横が r0 の 2 倍とする。

c. 上記 b と同じだが、加えて丸い部分の色と地の色も引数で指定できる手続きを作って、それを呼び出して 3 つ違う大きさで色違いの日の丸を描くアプレット。

演習 2 次のようなアプレットを作り表示せよ。

a. (x0,y0) を起点とし、角度 t 度、長さ l の極座標ベクトルの終点までの色 c の直線を引くメソッドを作り、中心から多数の放射状の点が描かれるアプレット。(ヒント: sin/cos の計算は Math.sin()、Math.cos() を使えばできます。ただし引数がラジアンなのにも注意)。

b. (x0,y0) に重心がある、1 辺の長さ l の正方形を色 c で描くメソッドを作り、これを利用して多数の色ちがいの正方形がかさなったものを描くアプレット。

c. 上記 b の正方形のうち 3 辺を色 c で描くメソッド (描かない辺を引数 d で渡し、d==0,1,2,3 のときそれぞれ右、上、左、下の辺を描かない) を作り、これを利用して十字型を描くアプレット。

4 アプレットの虫取りと例外について

4.1 アプレットの虫取り

アプレットを作るようになったので、その虫取り (間違いさがし) の方法をちょっと説明しておこう。まず、アプレットの中でも `System.out.println(...)` は使えるので、図形の形などがおかしい場合は座標の値などを表示させてチェックできる。ただし、その出力はブラウザでは「Java コンソール」と呼ばれるウィンドウに出るので、Netscape Navigator では「Communicator」メニューの「ツール」サブメニューで「Java Console」を選んでコンソールを表示させる。または、アプレットを appletviewer で動かせば出力は appletviewer を起動した窓に普通に表示される。

たとえば、次のアプレットを見てみよう。

```
import java.applet.Applet;
import java.awt.*;

public class R4Sample2 extends Applet {
  int []x = new int[101];
  int []y = new int[101];
  public void init() {
    for(int i = 0; i <= 100; ++i) {
```

```
        double theta = 0.01 * 2 * Math.PI * i;
        x[i] = (int)(100*Math.cos(theta)) + 100;
        y[i] = (int)(100*Math.sin(2*theta)) + 100;
      }
    }
    public void paint(Graphics g) {
      for(int i = 0; i <= 100; ++i) {
        g.drawLine(x[i-1], y[i-1], x[i], y[i]);  // ※
      }
    }
  }
```

このアプレットを Netscape Navigator で動かすと、絵がちゃんと表示されません。そこで、Java コンソールを開いてみると図 3 のような表示が出ています。

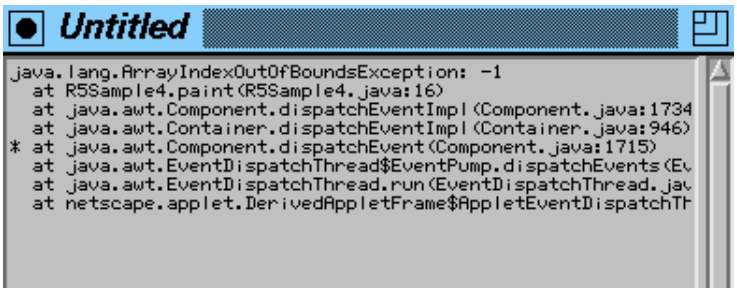

図 3: Java コンソールの表示

これを見ると、最初はごちゃごちゃに見えますが、冒頭に「ArrayIndexOutOfBoundsException: -1」とありますから、どうも「配列 (array) の添字 (index) が範囲 (bounds) を超えている (out) らしいと読る。その値は-1 みたいですね。次の行を見ると R5Sample3 というクラスの paint というメソッドの中で、ファイルでいうと R5Sample3.java というファイルの 16 行目だと出ている。つまり※の行ですね。そこでその近辺をよーく見ると、i が 0 のとき x[i-1] は確かに配列 x の-1 番目を参照してしまい、これが間違っているのだと分かる。正しくは「`for(int i = 1; …`」とすべきだったのですね。まあ、こう教科書通りには行かないにしても、エラーメッセージの見方に慣れておくとどこに間違いがあるか探す場所を絞ることができる。

4.2 例外

ところで、上にも出て来たように、さまざまな「エラー」が起きるとそのことを表すメッセージがコンソールなどの出てくる。実は、これらのエラーは Java では「例外」と呼ばれる機能によって統一的に扱われている。そして、特に指定しなければ例外が起きるとプログラムは終了し、その状況が表示されるので上のようなメッセージを目にするわけだ。

ここで、例外が発生したとき、それを受け止めて処理するようにプログラムを書いておけば、エラーが起きたときにプログラムがそこで止まってしまわないようにもできるので、その方法も学ぼう。

そのためには、例外を受け止めるための構文である try 文というものを使う。具体的には、try 文は

```
try {
  文 ... (1)
} catch(例外クラス名 変数) {
  文 ... (2)
}
```

という形をしていて、最初の (1) の部分にある文のならびの中で発生した例外を受け止めることができる。具体的には、例外が起きると実行は (2) の文のならびに「ジャンプ」して、この部分の文を最後まで実行したら try 文全体の処理が終わって次の文に進む。例外が起きなければ (1) の中の文は全部実行されるが、(2) の文は実行されないことになる。つまり (2) の部分はエラー処理専用の動作ということになる。

4.3 例外クラス

ここで「例外クラス」はエラーの種類を表すもので複数あるが、次のように階層構造に分類されている。

```
Throwable --- すべての例外やエラーの全体
  Error --- 回復不可能なエラー
    ClassFromatError --- .class ファイルが変である
    NoClassDefFoundError --- .class ファイルが見つからない
    ... その他いろいろ ...
  Exception --- 例外的なできごと全般
    RuntimeException --- 実行時エラー全般
      NumberFormatException --- 文字列が数値の形式になってない
      NullPointerException --- null 値に対してメソッドを呼ぼうとした
      IndexOutOfBoundsException --- 配列の添字範囲外をアクセスした
      ... その他いろいろ ...
    IOException --- 入出力エラー
    IntrruptedException --- 一時停止中に割り込みが起きた
    IllegalAccessException --- クラスの内容を不正にアクセスしようとした
    ... その他いろいろ ...
```

通常は受け止めて処理するのは Exception 以下なので、「例外クラス名」として Exception を指定することが多い。ただし、ある特定の例外だけ別に扱いたければ次のように try 文を入れ子にして使うことができる。

```
try {
  ... ※A
  try {
    ... ※B
    ...
  } catch(NumberFormatException e) {
    (1) 数値の形式が間違っていた場合の処理
  }
  ... ※C
} catch(Exception e) {
  (2) その他すべての例外の処理
}
```

ここで、※ B で NumberFormatException 例外が起きた場合には (1) の位置にある処理を実行するが、それ以外の例外は (2) の位置で処理する。※ A や※ C は内側の try 文の範囲外なので、すべての例外を (2) で処理する。

4.4 例外を受け止めたところの処理

「例外を処理する」具体的な内容としては、次のようなものが考えられる。

- 何も動作を書かない — 単に例外が起きても無視して先の処理へ進みたい場合は何も動作を書かなくてもよい。

- 簡単なエラーメッセージを表示する。System.err.println(...) でメッセージを出力すればよい。なお、System.err.println() は System.out.println() とほぼ同様だが、エラー表示専用の出力ストリームになっている。
- 例外クラスすべてが共通に持っているメソッド printStackTrace() を呼び出して詳しいエラー状況を表示する。

実はこれまでエラーが自動的に表示される場合は 3 番目の pirntStackTrace() による表示が行われていた。
では、エラーを処理する簡単な例題を見てみよう。

```
import java.io.*;

public class R4Sample3 {
  public static void main(String[] args) {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    while(true) {
      try {
        System.out.print("x> "); System.out.flush();
        String s = in.readLine();
        if(s.equals("")) break;
        int x = (new Integer(s)).intValue();
        System.out.print("y> "); System.out.flush();
        int y = (new Integer(in.readLine())).intValue();
        System.out.println("x + y = " + (x+y));
      } catch(Exception e) {
        System.err.println("Oops! some error occured...");
        e.printStackTrace();
        System.err.println("Continue...");
      }
    }
  }
}
```

この場合、たとえば入力に数字でないものを入れると、NumberFormatException が発生するが、それをループの内側で catch で受け止めているのでプログラムはエラーで終了せず、次のデータを入れることができる。このように、例外を自前で処理することで何かエラーがあっても終らずに続行することができる。

4.5 例外を投げる

ここまでは例外を受け止める話ばかりだったが、例外は自分で投げることもできる。たとえば処理していて正しくない事態に陥ったと思った時、そこで何かじたばたするより例外を投げてどこか別の場所でまとめて処理した方がきれいにできるのが普通なので、そのような時は自分で例外を投げれば良い。例外を投げるには、throw 文を使う。

```
throw 式;
```

ただし「式」は例外クラス (Throwable 以下) のオブジェクトを返すものでなければならない。とりあえずは次のような形で Exception オブジェクトを投げればよいだろう (自分でもっと適切なクラスを選んだり新規に作ってもよいが)。

```
throw new Exception("ほにゃららがまずい。");
```

また、先の catch 節での処理に次のものも追加しておこう。

- 一旦受け止めてエラーメッセージを出すが、さらに外側で処理してもらうため同じ例外を再度投げ直す。

この場合は次のような書き方になるだろう。

```
try {
  ...
} catch(Exception e) {
  System.err.println("....");
  throw e; // 投げなおす
}
```

4.6 throws 宣言

自分で投げた例外はこれまで通り catch で受け止めればよいが、自分のメソッド内では受け止めず、外側 (そのメソッドを呼び出した側) で受け止める場合にはメソッド定義の冒頭部分に

```
... メソッド名 (パラメタ…) throws 例外クラス名,... {
```

の形で宣言しておかなければならない。つまり「私はこれこれの例外を投げますよ」と予め明らかにしておかないと、そのメソッドを呼ぶ側で「心の準備」ができないので呼んでみたら知らない例外が投げられてびっくり、といったことになるからである。

実は、throws 節による宣言は、そのメソッドの中で throw で例外を投げる可能性がある場合だけでなく、そのメソッドの中で例外が発生するような操作を行って、なおかつその例外を catch しない場合にも必要である。というのは、catch しなかった例外はそのままメソッド呼び出し側に伝わって行くことになるから。

これでようやく、第 1 回のおまじないの謎がちょっとだけ解けることになる。つまり、

```
public static void main(String[] args) throws Exception { ...
```

というのは、この中で例外 (実は in.readLine() を呼ぶと IOException が投げられることがある) が発生してそれを捕まえないので外側にもそれが伝わりますよ、ということを断っているのだった。

ただし、この規則にも「例外」があって、Error と RuntimeException およびそれの子分の例外については throws 節で断らなくてもよい。というのは、これらの例外はあらゆる場所で発生し得るため、いちいち断っていると大変すぎるからである。

…大変長い説明ですいませんでしたが、一応きちんと説明するとこういうことになってしまうので。まあ、これからも try...catch が出たところでは簡単に復習して行きますから。

5 再帰的なメソッド呼出し

途中でだいぶ脱線したが、再度メソッドの話題に戻ろう。メソッドを使えば「一連の動作をまとめる」ことができ、またあるメソッドの中で「別のメソッドを呼び出して利用する」ことができるのは分かった。ところで、実はあるメソッドの中で「自分自身を呼び出す」こともできる。これを「再帰的なメソッド呼び出し」と呼ぶ。

たとえば、「渦巻き」を描くことを考えてみよう。「渦巻き」を描くには、まずある方向に線を引き、ちょっとだけ方向を変えて、さっきより短い線を引き、またちょっとだけ方向を変えて、またさらに短い線を引き、…を繰り返せばよい。そこで、次のような疑似コードを考えてみる。

- (x0,y0) から辺の長さ l0、角度 d0 で d1 ずつ曲りながら螺旋を描く:
- $x1 \leftarrow x0 + l0\ cos\ d0$
- $y1 \leftarrow y0 + l0\ sin\ d0$

- $(x0, y0)$ から $(x1, y1)$ まで直線を引く
- $(x1, y1)$ から辺の長さ $r * l0$、角度 $d0 + d1$ で $d1$ ずつ曲りながら螺旋を描く。

つまり、辺を 1 つだけ描いたらあとは「残りの螺旋を描く」作業は誰かに頼めばいいのだけど、それは実は自分自身と同じもの、というわけ。

しかしこれではいつまでも終わりにならないので「無限の堂々めぐり」になってしまう。そこで、辺の長さがだんだん短くなってある値以下になったら、それ以上描いてもしかたないのでやめることにしよう。つまり次のようになる。

- (x0,y0) から辺の長さ l0、角度 d0 で d1 ずつ曲りながら螺旋を描く:
- もし $l0 \leq l1$ ならば戻る。
- $x1 \leftarrow x0 + l0cosd0$
- $y1 \leftarrow y0 + l0sind0$
- $(x0, y0)$ から $(x1, y1)$ まで直線を引く
- $(x1, y1)$ から辺の長さ $r * l0$、角度 $d0 + d1$ で $d1$ ずつ曲りながら螺旋を描く。

ここで「戻る」というのは、メソッドの動作をここでやめて呼ばれたところへ戻る、という意味で、Java では「`return;`」という文になる。こうしておけば、堂々めぐりはある時点で止まるので、ちゃんと実行できる。ではこのコードを示そう。

```
import java.applet.Applet;
import java.awt.*;

public class R4Sample4 extends Applet {
  public void paint(Graphics g) {
    spline(g, Color.blue, 50, 100, -90, 30, 45, 5, 0.9);
  }
  void spline(Graphics g, Color c, int x0, int y0, int d0, int d1,
      int l0, int l1, double r) {
    if(l0 <= l1) return;
    int x1 = x0 + (int)(Math.cos(Math.PI*d0/180.0)*l0);
    int y1 = y0 + (int)(Math.sin(Math.PI*d0/180.0)*l0);
    g.setColor(c); g.drawLine(x0, y0, x1, y1);
    spline(g, c, x1, y1, d0+d1, d1, (int)(r*l0), l1, r);
  }
}
```

なお、「(int)(式)」というのは実数の計算結果を整数に丸める。また sin/cos のところでごちゃごちゃやっているのは、「度」をラジアンに換算している。ともあれ、ちょっと慣れないとまごつくけれど、「自分で自分を呼ぶ」のは覚えておくと役に立つ。

演習 3 上の例題を打ち込んでそのまま動かせ。

演習 4 上の例題は 1 方向にずっと曲っていくので「螺旋」になったが、1 つの「枝」から 2 つ以上別の方向に枝が延びると木の枝みたいな形ができる。

a. 木の枝みたいな形を描くように例題を直してみよ。

b. さらに、枝のはじっこに指定した色、大きさの円を描くことで「花の咲いた木」にしてみよ。

c. 枝や花の色を調整して、画面に沢山並べ、「桜の花盛り」を表現してみよ。

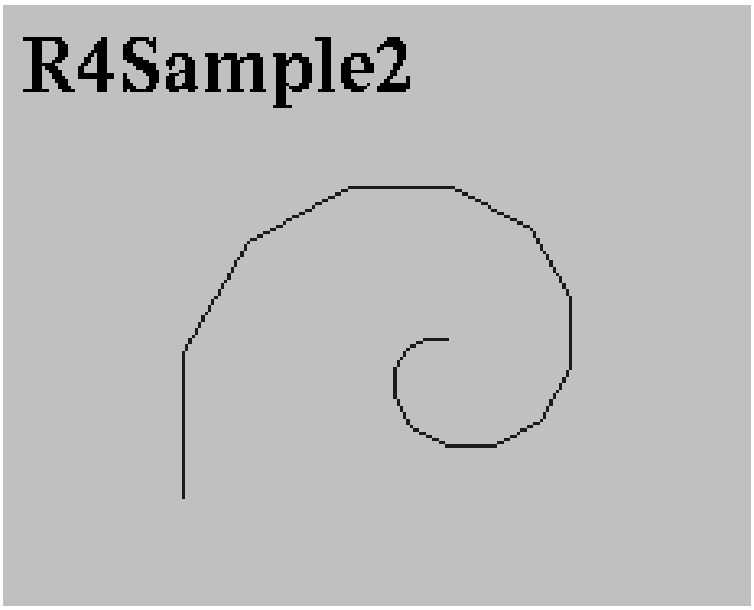

図 4: 例題 2 の表示

演習 5 次のメソッド dragon は $(x0, y0)$ から $(x1, y1)$ までをちょっと変わった形の折れ線で結ぶ働きがある。

- (x0,y0) から (x1,y1) までをパラメタ n に基づき折れ線で結ぶ:
- dx ← $\frac{x1-x0}{2}$
- dy ← $\frac{y1-y0}{2}$
- x2 ← $x0 + dx - dy$
- y2 ← $y0 + dx + dy$
- もし $n > 1$ ならば、
- (x0,y0) から (x2,y2) までをパラメタ $n - 1$ に基づき結ぶ。
- (x1,y1) から (x2,y2) までをパラメタ $n - 1$ に基づき結ぶ。
- そうでなければ、
- (x0,y0) から (x2,y2) までを直線で結ぶ。
- (x1,y1) から (x2,y2) までを直線で結ぶ。
- 枝分かれ終わり。

このメソッド dragon を Java に直し、アプレットに入れて動かせ。折れ線の規則がどうなっているかも分かる範囲で考えて見よ。

演習 6 メソッドや再帰的なメソッドを利用して、自分の好きな絵を描くアプレットを作成してみよ。