再帰

2004 年 1 月 9 日 増原 英彦

ある定義の中で、自分自身の定義を使うことを**再帰的 (recursive)** という。メソッドであれば、あるメソッドの定義の中で、そのメソッド自身を呼び出しているようなものを**再帰的 (recursive)** メソッド定義という。データ構造であれば、あるクラスの定義の中で、そのクラスのオブジェクトをインスタンス変数に持つようなものを再帰的なデータ構造という。

再帰的なメソッド定義を使うと、帰納的な考え方をそのままプログラムにできる。そのため、プログラムの働きや正しさを考えるのにも適した強力なプログラム方法になっている。再帰的なデータ構造を使うと、データ (オブジェクト) の個数がプログラムの実行中に増減するようなデータを扱うことができる。

1 帰納的定義

無限の場合について厳密な定義をするためには、**帰納的**(inductive) 定義が用いられる。帰納的定義とは「基底の場合」と「前の定義から 1 つ進めた場合」について定義を与えることで、あらゆる正の n について「基底の場合から n 個進めた場合」を定義する方法である。

例として x の y 乗の定義を考えてみよう。「x の y 乗」を $p(x,y)$ と書くことにする。p は y に関する帰納的な定義ができる。基底の場合は $y=0$ のときで、 任意の x について $p(x,0)=1$ である。「1 つ進めた場合」は y が 1 増えた場合になる。ある y について (任意の x に対して) $p(x,y)$ が定義済みであったとする[1]このときに、(任意の x について)$p(x,y+1)$ を定義できればよい。$p(x,y+1)$ は「x の $y+1$ 乗」、$p(x,y)$ は「x の y 乗」を表わしていることから、$p(x,y+1)=x\times p(x,y)$ である。y の値を 1 ずらして 1 以上の y に対して $p(x,y)=x\times p(x,y-1)$ と書いてもよい。これをまとめると、次のように書ける:

$$p(x,y)=\begin{cases}1 & (y=0\text{ の場合})\\ x\times p(x,y-1) & (y>0\text{ の場合})\end{cases}$$

冪乗 ($p(x,y)$) を定義するのに、冪乗自身 ($p(x,y-1)$) を使っていることに注意せよ。

この定義を用いれば、正の y について冪乗を計算することができる。例えば、2 の 4 乗は次のように定義を用いることで計算できる (下線部は次に展開される項を示している):

$$\begin{aligned}
p(2,4) &= 2\times p(2,\underline{4-1})\\
&= 2\times \underline{p(2,3)}\\
&= 2\times(2\times \underline{p(2,3-1)})\\
&= 2\times(2\times(2\times \underline{p(2,2-1)}))\\
&= 2\times(2\times(2\times(2\times p(2,\underline{1-1}))))\\
&= 2\times(2\times(2\times(2\times \underline{p(2,0)})))\\
&= 2\times(2\times(2\times(2\times 1)))\\
&= 16
\end{aligned}$$

また、この定義が正しいこと、つまり $p(x,y)$ が「x を y 回掛けた値」と等しくなっていることの証明は難しくない。まず、$y=0$ の場合 $p(x,y)=p(x,0)=1$ であり正しい。次に $y=0,1,\ldots,k$ について $p(x,y)$ が「x を y 回掛けた値」と等しいことが証明されていたと仮定する。このとき、$p(x,k+1)=x\times p(x,k)$ であるが、帰

[1] $p(x,y)$ の値は帰納法による証明の、帰納法の仮定に相当する。

納法の仮定より $p(x,k)$ は「x を k 回掛けた値」と等しいので $p(x,k+1) = x \times p(x,k)$ は「x を $k+1$ 回掛けた値」と等しい。帰納法により全ての正の y について $p(x,y)$ は「x を y 回掛けた値」と等しい。

帰納的な定義は、様々な計算について可能である。

- x の y 倍を $m(x,y)$ と書くことにすると:

$$m(x,y) = \begin{cases} 1 & (y = 0 \text{ の場合}) \\ x + m(x, y-1) & (y > 0 \text{ の場合}) \\ -m(x,-y) & (y < 0 \text{ の場合}) \end{cases}$$

  (つまり、掛け算は足し算と引き算で帰納的に定義できる。)

- y の階乗 $(y \times (y-1) \times (y-2) \times \cdots \times 3 \times 2 \times 1)$ を $f(y)$ と書くことにすると:

$$f(y) = \begin{cases} 1 & (y = 0 \text{ の場合}) \\ y \times f(y-1) & (y > 0 \text{ の場合}) \end{cases}$$

1.1 より複雑な帰納的定義

帰納的な定義は、自分自身を 2 回以上使ってもよい。フィボナッチ数列 (Fibonacci numbers) は、x 番目の数が $x-1$ 番目と $x-2$ 番目の数の和になっているような数列であり、$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$ のようになっている。$b(x)$ をフィボナッチ数列の x 番目の数だとすると、$b(x)$ は $b(x-1)$ と $b(x-2)$ を使って次のように定義できる:

$$b(x) = \begin{cases} 1 & (x = 0, x = 1 \text{ の場合}) \\ b(x-1) + b(x-2) & (x > 1 \text{ の場合}) \end{cases}$$

例えば 6 番目のフィボナッチ数は 13 であるが、この定義でも以下のように展開して確かに求められる:

$$\begin{aligned}
b(6) &= b(5) + b(4) \\
&= (b(4) + b(3)) + (b(3) + b(2)) \\
&= ((b(3) + b(2)) + (b(2) + b(1))) + ((b(2) + b(1)) + (b(1) + b(0))) \\
&= (((b(2) + b(1)) + (b(1) + b(0))) + ((b(1) + b(0)) + b(1))) + (((b(1) + b(0)) + b(1)) + (b(1) + b(0))) \\
&= ((((b(1) + b(0)) + b(1)) + (b(1) + b(0))) + ((b(1) + b(0)) + b(1))) + (((b(1) + b(0)) + b(1)) + (b(1) + b(0))) \\
&= ((((1 + 1) + 1) + (1 + 1)) + ((1 + 1) + 1)) + (((1 + 1) + 1) + (1 + 1)) \\
&= 13
\end{aligned}$$

また、2 つ以上のパラメータに関する帰納的な定義も可能である。例えば、m 個の中から n 個を選ぶ組み合わせ数 $c(m,n)$ は、

- k 個の中から k 個を選ぶ組み合わせは 1 つしかない、
- k 個の中から 0 個を選ぶ組み合わせも 1 つしかない、
- k 個の中から j 個を選ぶには、「先頭を選んで、残りの $k-1$ 個から $j-1$ 個を選ぶ組み合わせ」と「先頭を選ばずに、残りの $k-1$ 個から j 個を選ぶ組み合わせ」の和になる

ことから、

$$c(m,n) = \begin{cases} 1 & n = 0, m \text{ の場合} \\ c(m-1, n-1) + c(m-1, n) & \end{cases}$$

と定義できる。

練習 11-1: (複雑な帰納的定義) $a > b$ なる整数 a, b の最大公約数 $GCD(a,b)$ を帰納的に定義せよ。(ヒント: ユークリッドの互除法から、a, b の最大公約数は $b, (a\%b)$ の最大公約数と等しい。ただし、$a\%b$ は a を b で割った余りだとする。基底の場合は $b = 0$ になる。)

2　再帰的なメソッド定義

帰納的定義は、**再帰的なメソッド呼出**を使うことで、メソッド定義にすることができる。再帰的なメソッド呼出とは、あるメソッド定義の中で、そのメソッド自身を呼び出すことである。例えば、x の y 乗を計算するメソッドは次のように定義できる:

```
1 public static int power(int x, int y) {
2   if (y==0) {
3     return 1;                // 基底の場合
4   } else {
5     return x*power(x,y-1);   // y>0 の場合
6   }
7 }
```

- 1 行目:以下がクラスメソッドの宣言であること (`static`)、返り値の型 (`int`)、メソッドの名前 (`power`)、引数の型 (`int` と`int`) と、引数の値がセットされる変数名 (`x` と`y`) を示している。
- 2 行目の `if` 文は、`y` が 0 であれば 3 行目を実行し、そうでない場合 (つまり 0 以上) は 5 行目を実行させる。
- 3 行目は帰納的定義の基底の場合に相当し、1 を返している。これは $x^0 = 1$ という定義に対応する。
- 5 行目は帰納的な場合に相当し、`power(x,y-1)` という式で`x` の`y-1` 乗を計算し、それと`x` を掛けた値を返している。これは $p(x, y) = x \times p(x, y-1)$ という定義に対応する。この`power(x,y-1)` という式は自分自身を呼出している[2]、再帰的な呼出しである。

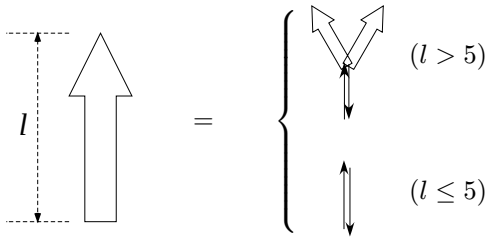
練習 11-2: (再帰的な冪乗メソッド)*　上のメソッド定義を用いて、2 の 0 乗から 30 乗までを計算し表示させるプログラムを作れ。余力があれば繰り返しを用いて冪乗を計算するメソッドと計算結果を比べ、同じ結果を返すことを確認せよ。

練習 11-3: (その他の再帰的メソッド)　1 節および練習 11-1 で書いた帰納的な定義に従って、(a)2 数の積、(b) フィボナッチ数、(c) 組み合わせ数、(d) 最大公約数、を計算するプログラムをそれぞれ作成せよ。

2.1　再帰的なメソッドとフラクタル図形

再帰的なメソッドは数学的な計算に限らない。木の枝は、中心となる太い枝に小さな枝が数本ついた形をしている。この小さな枝はまた、中心となる太い枝にさらに小さな枝が数本ついているという形をしている。つまり、木の枝は、太い枝に「木全体に相似な小さな枝」がついているといえる。

このような図形は、次のように帰納的に定義できる:



左辺の白い矢印を木だとすると、その長さ l が 5 より大きい場合は右辺上のように、幹に相当する線があり (実線の矢印)、その上端から左上と右上に向かって一まわり小さな木が伸びている (白い矢印)。幹の長さは $l/3$、

[2]クラスメソッドの呼出は クラス名.メソッド名(式, ...) のように書くが、同じクラスのメソッドの場合は単に メソッド名(式, ...) と書ける。

一まわり小さな木はそれぞれ左右に 30 度傾いて、長さが $2l/3$ だとする[3]。長さ l が 5 以下の場合は、右辺下の場合のように、単にその長さの線があるだけである。

この図形の描き方を知っているようなタートルは、Turtle クラスを拡張して次のように定義できる:

```
1  public class Tree extends Turtle {
2    public Tree() { super(); }      //コンストラクタは、
3    public Tree(int x, int y, int a) { //それぞれ親クラスの
4      super(x,y,a);                 //コンストラクタを呼び出すだけ
5    }
6    public void tree(int length){ //高さおよそ length の木を描く
7      if (length>5) {           //長さが 5 より大きい場合は枝分かれする
8        fd(length/3);           //幹を描く
9        lt(30);                 //左を向く
10       tree(length*2/3);       //再帰的に 2/3 の長さの木 (枝) を描く
11       rt(60);                 //右を向く
12       tree(length*2/3);       //再帰的に 2/3 の長さの木 (枝) を描く
13       lt(30);                 //正面を向き直す
14       bk(length/3);           //元の位置に戻る
15     } else {                  //長さが 5 以下の場合は線を一本描くだけ
16       fd(length);             //幹を描く
17       bk(length);             //元の位置に戻る
18     }
19   }
20   public static void main(String[] args){
21     TurtleFrame f = new TurtleFrame();
22     Tree m = new Tree(200,300,0);
23     f.add(m);
24     m.tree(200);                                  //木を描く
25   }
26 }
```

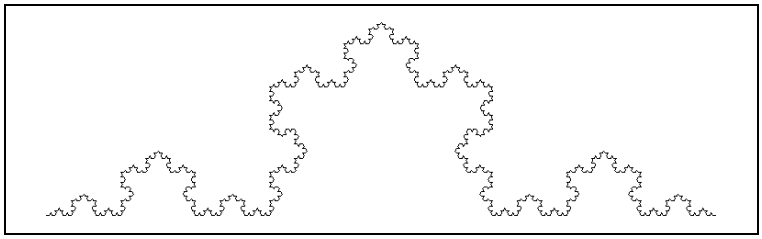
メソッド`tree(l)` は、Tree オブジェクトを使って (オブジェクトの現在位置から、現在向いている方向に向かって) 長さ`l` の木を描いた後に、オブジェクトを元の場所・向きに戻す。

幹の部分は 8 行目で描かれる。その先の小枝は 9, 11 行目で自身を適切な方向へ向け、10, 12 行目の`tree` メソッドの呼出しでそれぞれ描く。(`tree` メソッドは小枝を描いた後、元の場所・向きに戻るように作られていることに注意。) 最後に、13, 14 行目で、最初の向きと位置に戻っている。

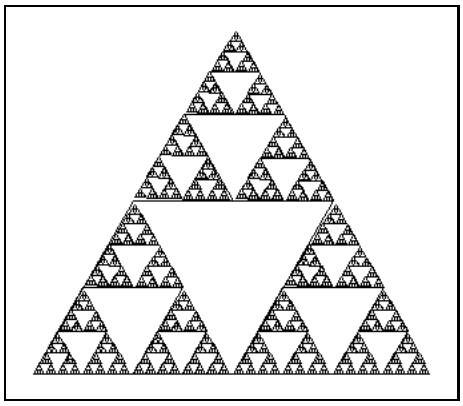
このような図形は**自己相似形**あるいは**フラクタル図形**といわれる。

2.2 他のフラクタル図形

以下に示すのは有名なフラクタル図形の一例である。



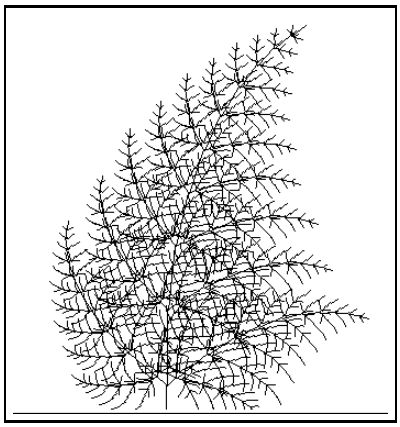
コッホ曲線



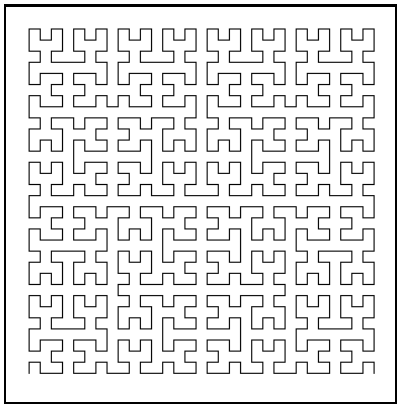
シェルピンスキーの三角形

これらの図形も木と同様に再帰的なメソッド呼び出しを利用して描くことができる。例えば一辺の長さが l のシェルピンスキーの三角形は、長さ $l/2$ の三角形 3 つで作られている。コッホ曲線は長さ l の直線が長さ $l/3$ の直線 4 本からなる山形によって作られている。

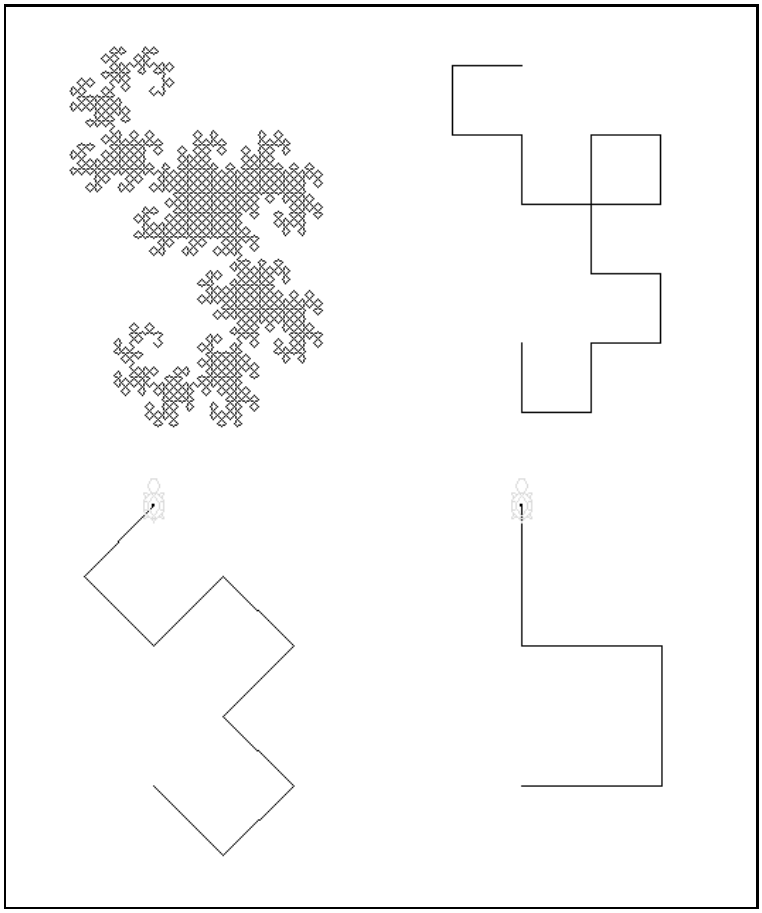
[3] この長さでは正確に長さ l の木を描くことにはならない。



木の変形



ヒルベルト曲線



ドラゴン曲線 (左上:8 段階, 右上:3 段階, 左下:2 段階, 右下:1 段階)

練習 11-4: (木のバリエーション)　上のプログラムをもとに、(a) 枝分かれの本数を 2 本から 3 本にした木、(b) 枝分かれの本数・枝の開く角度・幹と枝の長さの割合、といった要素を乱数によって変更するプログラムを作り、より自然に見える木、をそれぞれ描くプログラムを作れ。

3　再帰的メソッドの実行

`m.tree(200)` のようなメソッド呼出があると、その実行中に同じ名前のメソッドが呼出される。このとき、呼出す側のメソッドの実行と呼出される側のメソッドの実行は**全く別のもの**として扱われる。

より詳しく説明すると、メソッド呼出式が実行されるときはには、次のような処理が行われている。以下では区別のためにメソッド m1 がメソッド m2 を呼出すとして説明するが、同じ名前のメソッドを呼出したときでも同じである。

1. 実行しようとしているメソッド呼出式が m1 の中どの場所にあるかを、また m1 のメソッドの中で宣言されている変数の値を覚えておく
2. メソッド m2 を実行 (し終了) する。
3. 覚えておいた変数の値を元に戻し、覚えておいた場所から m1 の実行を再開する

3 で覚えおいた場所と変数の値に戻るので、2 で同じ名前のメソッドを実行したとしても、呼出す側と呼出される側は全く別のものとして扱われる。

4　再帰的メソッドの応用

練習 11-5: (冪乗)　練習 6-8 で紹介した冪乗を計算する「別のアルゴリズム」は、再帰的メソッドを使って簡潔に定義できる。まず、x の y は次のように帰納的に定義できる:

$$p(x,y) = \begin{cases} 1 & (y=0) \\ p(x^2, y/2) & (y \text{ は } 0 \text{ より大きい偶数}) \\ x \times p(x, y-1) & (y \text{ は奇数}) \end{cases}$$

これをもとにメソッドを定義すると、以下のようになる:

```
public static int power(int x, int n) {
    if (n==0) {
        return 1;                // n が 0 のとき
    } else if (n%2 == 0) {
        return (x^2)^{n/2}を計算;   // n が 0 より大きい偶数のとき
    } else {
        return x × x^{n-1} を計算; // n が奇数のとき
    }
}
```

上のプログラムを完成させ、2 の 30 乗を計算させよ。また、メソッドの最初に`System.out.println(n);`という行を挿入し、メソッドが何回呼出されているかを調べよ。

練習 11-6: (ハノイの塔)　3 本の柱 (p_0, p_1, p_2 とする) と、n 枚の中央に穴の開いた円盤 ($d_0, d_1, \ldots, d_{n-1}$) があり、$d_i$ は d_{i+1} より小さいとする。初め全ての円盤は p_0 に下から $d_{n-1}, \ldots, d_0$ の順に差してあったとする。このとき、全ての円盤を p_0 から p_1 に移す手順を考えよ。ただし、1 つの柱に差してある円盤は 1 番上のものしか移動できず、より大きな円盤の上にしか重ねられないものとする。

p_f にある $d_i, \ldots, d_0$ を p_t へ移動する (ただし、残りの柱は p_e とする) には、次のような手順をふめばよい:

- p_f にある $d_{i-1}, \ldots, d_0$ を p_e へ移動し (ただし残りの柱は、p_t)、
- d_i を p_f から p_t へ移動し、
- p_e に移した $d_{i-1}, \ldots, d_0$ を p_t へ移動する (ただし残りの柱は、p_f)

これを再帰的なメソッドで定義すると、次のようになる。このプログラムを完成させよ。(参考 Hanoi.java)

```
/** 円盤 di〜d0 を柱 pf から柱 pt に (柱 pe を使って) 移す */
public static void hanoi(int di, int pf, int pt, int pe) {
  if (di == 0) {                          // ベースケース
    di を pf から pt に移す
  } else {                                // di の上に他の円盤があるとき
    di-1〜d0 を pf から pe に (pt を使って) 移す
    di を pf から pt に移す
    di-1〜d0 を pe から pt に (pf を使って) 移す
  }
}
```

5　再帰的なデータ構造

これまでに学んだデータの種類 (データ型) には

- 整数 (int) や実数 (double) のような基本型、
- 同じ型のデータを一列に並べた配列型と、
- 複素数のようにいくつかのデータ型を組み合わせたオブジェクト型

があった。実用的なプログラムでは、複数の値をまとめて一つの値として扱うことができる配列型やオブジェクト型が必須となる。

オブジェクト型は、基本型だけでなくオブジェクト型の値をも組み合わせることができる。つまり、オブジェクトのインスタンス変数には、基本型の値に限らず、他のオブジェクトをしまうことができる。(例えばタートルオブジェクトのインスタンス変数には、「X 座標」や「方向」といった数値だけでなく、「色」や「表示されている TurtleFrame」といったオブジェクト (の参照) がしまわれている。)

多くのプログラムには、値を一列に並べたようなデータ (cf. カラオケの予約リスト) や、値をグループ化し、それをさらにグループ化したようなデータ (cf. 組織構造や階層ファイル構造) のようなデータが現われる。このようなデータの設計を**データ構造**という。

データ構造の中には、構造の定義に自分自身の定義を再帰的に使っているものがある。つまり、あるクラスのあるインスタンス変数が、そのクラス自身の型になるような設計をしている。このようなデータ構造を**再帰的なデータ構造**という。再帰的なデータ構造を使うと、無限の大きさを持つデータを簡単に表現できる。

5.1 リスト構造

リスト構造は、データを鎖のようにつなぎ、一列に並んだデータ表わすための再帰的データ構造である。配列と異なり、列の長さを増減させることや、列の途中にデータを挿入したり、取り除くことが簡単にできる特徴がある。

以下では、マウスボタンをクリックによって、画面上のタートルを増減させることのできるプログラム ListDemo を例にリスト構造の定義と使い方を紹介する。ListDemo では画面上にあるタートルを管理するために Turtle を拡張した LinkedTurtle を用いる。LinkedTurtle は、複数のタートルを一列に並べたリスト構造であり、マウスクリックによってタートルが追加されたり、取り除かれたりする。また、リストに登録されたタートルを全て前進・回転させる機能もある。

5.1.1 LinkedTurtle の定義

LinkedTurtle は通常のタートルに加え「次のタートル」を覚えることができる。次のタートルが、その「次のタートル」を覚え、その次のタートルがそのまた次のタートルを覚え……とすることで、無限に長いタートルの列を表わすことができる。

```
1  public class LinkedTurtle extends Turtle {      //数珠つなぎタートル
2    LinkedTurtle next;                            //次のタートル
3    public LinkedTurtle(int x, int y, int angle, LinkedTurtle next) {
4      super(x, y, angle);
5      this.next = next;
6    }
7  }
```

3–6 行目はコンストラクタである。つまり、

`new LinkedTurtle(`x 座標`,`y 座標`,` 角度`,` 次のタートル`)`

という式で、新しいタートルを作ることができる。もし「次のタートル」がいない場合には、

`new LinkedTurtle(`x 座標`,`y 座標`,` 角度`,null)`

として、「末尾のタートル」を作ることができる。インスタンス変数 next は LinkedTurtle のオブジェクトをしまうことができる。つまり、 LinkedTurtle クラスは、定義中に LinkedTurtle クラス自身が使われている**再帰的なデータ構造**になっている。

5.1.2 リストを作る

タートルのリストを使うプログラム (ListDemo) は、ボタンが押されるのを待ち、押されたボタンに応じた処理をする。まず「追加」というボタンが押されると、タートルを 1 つ作り、それをリストに追加する部分を見る:

```
1  public class ListDemo {
2    public static void main(String[] args) {
3      ClickableTurtleFrame f = new ClickableTurtleFrame();//ウィンドウを作る
4      f.addButton("追加");                                //「追加」ボタンを作る
5      LinkedTurtle l = null;                              // タートルのリスト; 始めは空
6
7      while (true) {              //ボタンが押されるのを待ち、それに応じた動作をする
8        String command = f.getPressedButton();          //ボタンが押されるのを待つ。
9        if (command.equals("追加")) {                   //「追加」ボタンの場合
10         int x = f.getClickedX(), y = f.getClickedY(); //クリックされた座標を得る
11         l = new LinkedTurtle(x, y, 0, l);             //その場所にタートルを作り
12         f.add(l); l.fd(0);                            //画面に追加する
13       } else
14         他のボタンが押されたときの処理 (略)
15     }
16   }
17 }
```

このプログラムでは変数 l に先頭タートルをしまっている。4 行目にあるように、最初、リストは「空」の状態である。`null` は「どのオブジェクトも参照していない」ことを示す特別な値である。

リストを作る典型的な方法は、すでにあるリストの先頭に要素を 1 つずつ追加してゆくものである。11 行目では、新しくタートルを作る。このタートルの「次」が今まで「先頭」だったタートルになる。新しく作ったタートルを l に代入するので、結果としてすでにあったリストの先頭に要素を 1 つ追加したことになる。

例として、「追加」ボタンが 3 回押され、タートルが 3 匹追加される場合を考えてみよう。

1. 最初、l は null, つまりリストは「空」である。
2. 「追加」が押されると、タートルが 1 つ作られ (l1 とする)、その next が l, つまり null になる。この時点で l は l1 だけからなる長さ 1 のリストである。
3. もう 1 度「追加」が押されると、タートルがもう 1 つ作られ (l2 とする)、その next が l, つまり l1 になる。この時点で l は l2, l1 からなる長さ 2 のリストになる。
4. さらに「追加」が押されると、タートルがもう 1 つ作られ (l3 とする)、その next が l, つまり l2 になる。この時点で l は l3, l2, l1 からなる長さ 3 のリストになる。

5.1.3 リストに対する繰り返し

「前進」ボタンが押されると、リストに含まれている全てのタートルが 10 だけ前進する。これを実現するには、先頭から全てのタートルに対して fd(10) というメソッドを呼び出せばよい。再帰的なデータ構造の全ての要素に対して処理をしたり計算をするには、再帰的メソッド呼出を使った定義が適している。この場合、つまり <u>先頭とそれに続く全てのタートルを前進させる</u> には

> まず、先頭のタートル (自分自身) を前進させ、
> 次に、<u>続くタートルとそれに続く全てのタートルを前進させる</u>

と考えることができる。下線部 (あるタートル ℓ とそれに続く全てのタートルを前進させる) を $f(\ell)$ と書くことにすると、次のような帰納的定義になる:

$$f(\ell) = \begin{cases} \ell\text{を前進} & \ell.\texttt{next}\text{ が空の場合} \\ \ell\text{を前進して } f(\ell.\texttt{next}) \text{ を実行} & \text{それ以外} \end{cases}$$

これをプログラムにすると、次のようなインスタンスメソッドを <u>LinkedTurtle クラス</u> に追加することになる:

```
1  public class LinkedTurtle extends Turtle {
2      インスタンス変数・コンストラクタの定義 (略)
3        public void forwardAll(int s) {   // 全てのタートルを s だけ前進
4          fd(s);                          // 自分を前進
5          if (next != null)               // もし次のタートルがいれば
6            next.forwardAll(s);           // 次のタートル (とそれに続く全て) を前進
7      }
8  }
```

6 行目は「続くリスト要素」(next) 中の全てのタートルを前進させる再帰的なメソッド呼び出しである。現在のリスト要素がリストの最後だった場合、next が null になっているので、5 行目の if 文を使って next が null でない場合のみメソッド呼び出しを行う。

練習 11-7: (タートルの回転) リストに含まれる全てのタートルを回転させるインスタンスメソッド void turnAll() を LinkedTurtle クラスに定義し、さらに ListDemo クラスを変更して「回転」ボタンがクリックされるとこのメソッドを呼び出すようにせよ。回転する角度は 0〜360 の乱数で決めることとする。(ヒント: Math.random() は 0 以上 1 未満の実数 (double) 型の乱数を発生させる。)

練習 11-8: (リストの長さ)* リストの長さを数えるインスタンスメソッド int length() を LinkedTurtle クラスに定義せよ。(ヒント: リストの長さは「続くリストの長さ」+1 である。) ListDemo を変更して、追加や削除がされる度に、現在のリストの長さを表示させせよ。(注意: リストが空 (null) のときは、メソッド呼び出しはエラーになるので、リストが空でない場合にのみ表示するようにしなければいけない。)

練習 11-9: (最上位置のタートル) リストに含まれるタートルのうち、Y 座標が最も小さい (つまり画面の一番上の位置にある) ものを探すインスタンスメソッド findTop() を LinkedTurtle クラスに定義せよ。さらに ListDemo を変更して「移動」ボタンによって全タートルが移動したら、最上位置にいるタートルの色を赤く変えるようにせよ。

ヒント: あるリスト要素 (から始まるリスト中) で「最上のタートル」は、「続くリスト要素中で最上のタートル」と、「先頭のタートル」のうち、より上方のものになるので、次のようになる:

```
1  public class LinkedTurtle extends Turtle {
2    インスタンス変数・コンストラクタ・他のメソッド (略)
3    public LinkedTurtle findTop() {
4      if (next== null)
5        return this;                      //次がいないなら自分が最上位置にいる
6      else {
7        LinkedTurtle topOfNext = 次のタートルから最上位置のタートルを探す;
8        自分 (this) と topOfNext の Y 座標を比較し、小さい方を return する
9      }
10   }
11 }
```

また、タートル m の色を変えるには、`m.kameColor = java.awt.Color.red; m.fd(0);` のようにする。(`m.fd(0);` は画面に表示されているタートルの色を即座に変えるためのトリックである。)

5.1.4 リストの最後に要素を追加する

5.1.2 で見たように、すでにあるリストの 先頭に タートルを追加するには、いままでの「先頭」を「次」とするようなタートルを作ればよかった。一方、リストの 最後に 要素を追加するには、リストの最後のタートルを探し、それの「次」に新しいタートルをしまえばよい。そのためには次のような再帰的なメソッドを LinkedTurtle クラスに追加すればよい:

```
1  public class LinkedTurtle extends Turtle {
2      インスタンス変数・コンストラクタ・他のメソッド (略)
3      public void append(LinkedTurtle m) {
4          if (next == null) {       // 自分がリストの最後なら
5              next = m;             // m を次にする
6          } else {
7              next.append(m);       // 次のタートルに「最後に m を追加」と指示
8          }
9      }
10 }
```

ListDemo の側では、新しく作ったタートルを引数としてこの append メソッドを呼び出すことでリストの最後にタートルを追加できる:

```
1  if (「追加」ボタンが押された) {
2      int x = (略), y = (略);                          //クリックされた場所を調べる
3      LinkedTurtle m = new LinkedTurtle(x,y,0,null); //その場所にタートルを作り
4      m を画面に表示する (略)
5      if (l==null)                                    //先頭も空の場合
6        l = m;                                        //自分が先頭かつ最後
7      else
8        l.append(m);                                  //タートルを最後に追加する
9  } else
10     他のボタンが押されたときの処理 (略)
11
```

5.1.5 リストから要素を削除する

「削除」ボタンが押されると、一番最後にリストに追加されたタートルがリストから取り除かれる。これを実現するには、変数 l が参照している「先頭のタートル」を、「次のタートル」に書き換えるだけである。

ListDemo 中では、「削除」が押されたときに l を書き換えればよいので、次のようになる:

```
1  if (「追加」ボタンが押された) {
2      タートルを追加する
3  } else if (「削除」ボタンが押された)) {
4      l = l.next;                    // リストの先頭を「続くリスト」に変える
5  } else
6      他のボタンが押されたときの処理 (略)
```

(注: この例ではタートルは画面に表示されたままになる。ただし、「前進」を押してもリストから外れたタートルは前に進まなくなる。)

次に、画面上で 1 番上の位置にあるタートルを 1 つリストから削除することを考える。

この場合、リストの途中にある要素を取り除くことになるが、これには、その要素の 1 つ前の「次」を書き換えることになる。

先頭のタートルが画面上で 1 番上にあった場合も考えると、リストから特定のタートルを含む要素を取り除くメソッドは、次のように「要素が取り除かれたリストを 返す メソッド」として定義するのがよい:

```
1  public class LinkedTurtle extends Turtle {
2      インスタンス変数・コンストラクタの定義・他のメソッドの定義 (略)
3      public LinkedTurtle delete(LinkedTurtle m) { // m を除いたタートルリストを返す
4          if (this == m) {                         // 自分が m だったら
5              return next;                         // 続くタートルを返す
6          } else {                                 // m を含んでいなかったら
7              next = next.delete(m);               // 続くタートルから m を除いて
8              return this;                         // 自身を返す
9          }
10     }
11 }
```

7 行目では「次」(next) の delete メソッドを再帰的に呼び出し、その返り値を next に代入し直していることに注意せよ。もし「続くリスト要素」に m があった場合、「続くリスト要素」の次のリスト要素が返り値となるので、結果として次の要素が削除される。

ListDemo でも、下の 5 行目のように、「l から m が取り除かれたリスト」を l に代入し直す。もし m が l の先頭にあった場合、l の値も変更しなければならない。一方、m が l の先頭にない場合は、最初のリスト要素は同じオブジェクトである。下のように l.delete(m) の返り値を l に代入し直すことで、両方の場合に正しく働く:

```
1  if (「追加」ボタンが押された) {
2      タートルを追加する
3  } else if (「削除」ボタンが押された)) {
4      LinkedTurtle top = l.findTop(); //一番上にあるタートルを探す
5      l = l.delete(top);              //top をリストから外す
6  } else
7      他のボタンが押されたときの処理 (略)
```

練習 11-10: (最上のタートルの削除) ListDemo の中の削除ボタンが押されたときの定義を変更し、画面の一番上の位置にあるタートルをリストから外すようにせよ。(画面の一番上の位置にあるタートルは、練習 11-3 で定義したメソッドを使う。)

練習 11-11: (特定の場所への追加)* ListDemo のリスト中に、タートルが X 座標の小さい順に並ぶように、タートルを追加するときにリストの途中にに追加するようにせよ。タートルが移動して X 座標が変化する場合は考えないことにする。(ヒント: タートルをリストの適当な位置に追加し、追加されたリストを返すメソッドを定義すればよい。このメソッドは、自身の 手前に 新しいタートルを挿入すべきかどうかを判断し、挿入する場合は、自身を「続くリスト要素」とするようなリスト要素を作り、返せばよい。)

練習 11-12: (配列との比較) ListDemo と同じ機能を持ち、リスト構造を使わずに、配列を使って「全てのタートル」を管理するようなプログラムを作ってみよ。配列では簡単に行えない操作は何か。