1 . . . の豆旅

1.1 フィボナッチ数列

数列とは、数が何らかの規則で並んでいるものを言う。

$$2,\ 4,\ 6,\ 8,\ 10,\ \ldots$$

は偶数の数列だが、偶数列を作る規則は『2 から始めて前の**項**に 2 を加える』というものである。

さて、ここに 1, 1 から始まる数列がある。次の項を作る規則は『直前の 2 項の和』である。すなわち 1, 1 の次に来る項は 2 となり、数列は 1, 1, 2 と延びる。規則を繰り返し当てはめれば、次の項は 3 で数列は 1, 1, 2, 3 と延び、さらに次の項は 5 である。これが延々となされ

$$1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ \ldots \tag{1}$$

なる数列が出来上がる。このようにしてできる数列を**フィボナッチ**[1)]**数列**と呼ぶ。

フィボナッチ数列を **Java** で再現してみよう。

programming list [Fibonacc.java]

```
 1: public class Fibonacc {
 2:
 3:     public static void main(String[] args) {
 4:         final int SUP = 20;
 5:         int f1 = 1, f0 = 1;
 6:
 7:         for(int i = 3; i <= SUP; i++) {
 8:             System.out.println(f1 + f0);
 9:             f1 = f1 + f0;
10:             f0 = f1 - f0;
11:         }
12:     }
13: }
```

はじめに気が付くのは 4:行目の `final int SUP = 20;` だろう。`int` の前の `final` は、変数に代入された値が最終値であることを示している。別の言い方をすれば、`SUP` は `20` という変更できな

1) フィボナッチ (1174?–1250?) : イタリア、ピサのレオナルドの通称。

い定数値を与えられたということだ。すると以後 SUP が出るたびに、それは 20 に置き換わる。いまは 20 項までの計算をさせたいため、あらかじめ SUP を 20 にしているわけだ。こうする利点は、あとで 100 項まで計算させたくなったとき、この行の変更だけで済ますことができる。このような短いプログラムでは恩恵が分かりにくいが、あちこちに一定の値が使われるプログラムなら、final で定義しておくとよいだろう。変数名を大文字で書いたのは定数を目立たせるためで、他の変数のように小文字でもかまわない。

次に 5:行目で、フィボナッチ数列の計算に必要な二つの変数 f1, f0 を用意した。数列は 1, 1 の項から始まっているので、変数を宣言すると同時に 1 を代入している。変数はどこかで代入による初期化をしなくてはならない。この例のように変数を宣言したときに初期化することは、以前から for 構文でも行っていたことに思い当たるだろう。もし、ここで初期化を怠ると、8:行目でなされる f1 + f0 の結果がいくつになるか予想できない。

7:行目の for 構文は i = 3 から始めて、SUP—いまは 20 で定義されている—まで 1 ずつ増加させている。i = 3 から始めた訳は、f1 + f0 で与えられる項がすでに第 3 項になるからである。そんなことを気にしないなら i = 1 から始めても結構だ。

8:行目の System.out.println; 文は、f1 + f0 を計算した値を表示する。あれ？　こういうときの + は文字を連結するんじゃなかったっけ？　ごもっともだが、ここには文字列が現れない。つまり、文字列がない場合においては—正確には **Java** が文字列を認識するまでと言うべきだろう— + は普通の足し算として扱われるのだ。

9:–10:行目はかなりトリッキーなことをしている。フィボナッチ数は常に f1 + f0 で計算するので、ここの f1 + f0 の値が次は f1 となり、いままで f1 だった値が f0 にならなくてはならない。だったら、f1 = f1 + f0;、f0 = f1; でいいんじゃない？　と思うだろう。残念だがそれは違う。

EX.　なぜだめなのか、よ〜く考えてみよ。

ならば代入の順番を変えて、f0 = f1;、f1 = f1 + f0; としたらどうだろう。だが、これもさっきの理由と同じでだめなのだ。

そこでトリッキーな解決をしている。9:行目が終わったとき f1 には f1 + f0 が代入される。これは予定通りだ。次に f0 には “古い”f1 の値がほしいのだが、この時点での f1 は “新しい” ものになってしまった。しかし安心してほしい。“新しい”f1 は “古い”f1 を含んでいる。なぜなら

`f1 = f1 + f0` だからだ。`f0` がなければ“古い”`f1` を取り出せる。そこで 10:行目の式となる。見事に `f0` に“古い”`f1` の値が入っただろう。

このプログラムでは、あまり先の項まで調べることは無理だが、ある程度のところまでのフィボナッチ数列を感じることができるはずだ。

TRY! `final int SUP = 20;` の値を変更して、もう少し先のフィボナッチ数を求めてみよ。

TRY! このままのプログラムでは、たとえば 34 が何項目のフィボナッチ数か分かりづらい。そこで、“fibonacci[9] = 34” のような表示をするプログラムにせよ。

TRY! ここではトリッキーな代入をしているが、新しいフィボナッチ数を代入するための変数 `f2` を用意すれば、もっと直観的にプログラムが組める。変数 `f2` を追加してプログラムを書き換えよ。

さあ、この地まで豆旅をしてきてだいぶ見通しがよくなってきたことだろう。大過なく豆旅を続けられているだろうか。しかし、プログラムはちょっとした気のゆるみで、とんでもなくおかしな動作をすることがある。いま `for` 構文を上限を決めて実行できるようになったのだが、上限の値を不正なものにするとプログラムが暴走する危険だってある。だから、上限に設定する値は常に確認することが重要だ。そしてもっと重要なのは、プログラムが暴走したり無限ループに入り込んだとき、それを止める方法を知っていることである（大抵は ctrl+c キーや ctrl+z キーで止まる）。