4.2 素数の表示

プログラム [PrimeChk.java] では、入力された数が素数かどうか判定するものだった。それならはじめから素数の一覧表があると便利だろう。そこで素数の一覧を表示するプログラムを用意した。今度のプログラムでは、随所に簡潔だが分かりづらい—しかし、慣れれば分かりやすい—書き方をしている。簡潔なほど単純で分かりやすいというのは、プログラムを組む場合には当てはまらない。冗長で複雑そうに見えるほうが分かりやすい場合もある。記述の仕方は、ある程度好みの問題を含むので一概に言えない。私はここまでに、やや冗長な書き方をしてきたが、簡潔な記述を好む人もいるだろう。豆旅においても、目に付いた所を悠然と訪ねる旅程もあれば、名所を効率的に訪ねる旅程もある。目に付く場所を悠然と見ていれば、そこがどんな地か分かりやすい。冗長な記述はそういうものだ。一方、名所を効率的に訪ねるには、ある程度の知識が必要になる。簡潔な記述がそれにあたる。

プログラム [PrimeNum.java] では、所々簡潔な表現を用いた。何もそうする必要はなかったのだが、こんな書き方も可能だということを示しただけである。

programming list [PrimeNum.java]

```
1: public class PrimeNum {
2:
3:     final static int SUP = 100;
4:
5:     public static void main(String[] args) {
6:         System.out.print("2 ");
7:
8:         for(int n = 3; n < SUP; n += 2) {
9:             int i = 1, flag = 1;
10:
11:            while((i += 2) <= Math.sqrt(n)) {
12:                if(flag > 0) {
13:                    flag = n % i;
14:                } else {
15:                    break;
16:                }
17:            }
18:            if(flag > 0)
19:                System.out.print(n + " ");
20:                ;
21:        }
22:        System.out.println();
```

```
23:     }
24: }
```

プログラムを解説する前に、素数の調べ方について話しておこう。たとえば 49 が素数かどうか判定するには、2 から 48 までの数で割ってみればよい。何一つ割ることができなければ 49 は素数だ。しかし考えてみてほしい。素数は 2 を除いてすべて奇数である。奇数は偶数で割れっこない。したがって偶数での割り算を試す必要はないのだ。その結果、49 が素数かどうか調べるには 3 から 47 までの奇数で割ってみるだけでよい。これで計算量を半分に減らせる。

さらに、n が素数がどうか調べるのに、$n-1$ までの数で割る必要などない。結論を言えば $\sqrt{n}$ までの数で割るだけで十分である。たとえば 51 は 3 で割れる—商は 17 である—から、同時に 17 でも割れてしまう。数 n が p で割れるなら、同時に n/p でも割れているのだ。

EX. このことから n が素数かどうか調べるのに、$\sqrt{n}$ までの数で割ればよいことが結論できる。それを確認せよ。

以上の考察によって、プログラムは次の考えに基づいて組み立てられる。

a) 奇数 n だけを対象に素数かどうかを調べればよい。

b) 候補となる n を割る数は、3 から $\sqrt{n}$ までの奇数で十分である。

それでは `[PrimeNum.java]` を見ていこう。

3:行目で調べる数の上限を 100 としている。100 までには 25 個の素数がある。もちろん、もっと大きな素数まで求めたければ、ここの値を変えるだけでよい。`SUP` を `main()` メソッドの外側で定義したのは単に目立つからで、このプログラムでは `main()` メソッド内でも問題ない。

プログラムはすべて `main(){}` に詰め込んだ。6:行目は画面に "2 " を表示する文だ。それは 2 が最初の素数だからである。あれ？ **Java** が計算するんじゃないの？と感じたかな。何でもかんでもコンピュータに計算させようとするのは間違っている。自明なことは計算させることなどないのだ。もし君たちが、素数を 2, 3, 5, 7 まで確実に知ってると言うなら、6:行目は `System.out.print("2  3  5  7  ");` としてかまわない。そして 8:行目では、`n = 9` から調べればよいのだから。

しかし、このプログラムは正直に `n = 3` から調べている。それは 8:行目の `for` 構文を見れば分

かる。for() 内の最後が n++ でないことに注意してほしい。調査は奇数に対して行われるので、2 ずつ増えるよう n += 2 と書いている。これは n = n + 2 の省略形だったね。そして、この for 構文は 21:行目の } で終わっている。

9:行目の i は n を割り算で試す除数である。i = 1 としたのは、次の while 構文で i を 2 増やして、3 から割り始めるようにするためだ。もう一つの変数 flag はいままでの変数と違う使われ方をする。候補となる数が素数でなければ flag に 0 が代入され、素数であれば flag に 正の数が代入される。素数を画面に出力するかどうかは、flag の値で判断するのである。flag = 1 としたのは、とりあえず候補の数は素数であると仮定するためだ。素数でないことが判明し次第、flag には 0 が代入される。

さあ、ここからが厄介だ。11:行目は while 構文だが、この終わりは 17:行目の } である。したがって、while() の条件が正しければ {} 内の処理が行われ、条件が正しくなければ 18:行目の if 構文に処理が飛ぶのだ。

具体的には、while 構文に入る直前は n = 3, i = 1 だったから、i += 2 で i は 3 になっている。すると条件 (i += 2) <= Math.sqrt(n) は、$3 \leq \sqrt{3}$ が正しいかどうかを問うていることになる。$3 \leq \sqrt{3}$ は正しくない、すなわち**偽**だ。よって while 構文は {} 内の処理をしないで 18:行目に移る。

18:行目は if 構文である。if 構文も while 構文同様、() 内の条件によって処理が分岐する。ここでの条件は flag > 0 かどうかだ。処理が 18:行目に来たとき、flag の値は 1 だったはずだ。だからこの if 構文は真の判定をし、そのための処理をするわけだ。

ところで if 構文の下には、19:行目に System.out.print; 文が、20:行目に ... ん？　; だけがある。しかも、ここの if 構文には {} と else が使われてないことに注意してほしい。

if(A){X} else{Y} のように else が使われている if 構文であれば、A が真のときには {X} の処理がされ、A が偽のときには{Y}の処理がされたことと思う。つまり二者択一でどちらか一方の処理だけがされる。だが、if(A) X Y では注意が必要だ。この場合は、A が真のときには X から処理が始まり Y 以下が処理される。A が偽のときには X は飛ばされ Y から処理が始まるのだ。

するといまはどうなるんだろう。18:行目の if(flag > 0) は flag = 1 であるから真の判定が下される。真であれば 19:–20:行目に処理が進むはずだ。20:行目は n の値を出力するので画面には “3” が表示され、さらに 20:行目も処理される。

ところで 20:行目は ; しかない。これはいささか矛盾を含む言い方だが**空文**と呼ばれ、何もしな

い文である。では、何のためにあるのか？　それは `if(flag > 0)` が偽になったときのためである。`if(flag > 0)` が偽のときは素数でないのだから、何もせず次の数へ進む必要があるのだ。

さて、プログラムは 18:行目以降に飛んでしまったが、11:行目からの `while` 構文はいつ実行されるのだろうか？　それは `(i += 2) <= Math.sqrt(n)` が真になるときだから、`n = 9` になって初めて実行される。

このとき 12:行目で `if(flag > 0)` の判定があるが、この時点での `flag` は `1` である。したがって真の場合の処理、13:行目が実施される。

13:行目の処理はちょっとうまい方法を用いている。それは `flag` に n/i の余りを代入していることである。`n` が素数でなければ余りは `0` だから、`flag` が `0` になるのだ。素数の可能性があれば必ず `0` 以外の余りが出るので、何度でも 13:行目が試されることになる。もし、何度試しても割り切れず、そうこうするうち `(i += 2) <= Math.sqrt(n)` が偽になれば、`while` 文を抜けて 18:行目以降で素数と判定されるのである。

一方、13:行目で余りが `0` になれば、次回の `if(flag > 0)` の判定は偽になって、`else` へ処理が移る。

`else` は 15:行目の `break;` である。`break;` は "現在行われている `{}` 内の処理" を中断し、`}` の直後へ出る命令だ。おっと、この言い方だと誤解を生じかねない。具体的に言おう。この場合、"現在行われている `{}` 内の処理" というのは、`else{}` ではなく、`if(){}` 全体—すなわち `while(){}` 内の処理—のことである。したがってここでは 17:行目の直後へ出るが、この場所はまだ `for` 構文の中である。

しかも、`break;` 文で 17:行目の直後へ出たときは、`flag` の値は `0` だから 19:–20:行目は無いに等しい。よって、`n` が `2` 増えて、引き続き `for(){}` が繰り返される仕組みだ。

もし `break;` の出どころが不安なら、11:–17:行目の `while` 構文は

programming list [change of line 11–17.java]

```
11: :         while( (i += 2) <= Math.sqrt(n) ) {
12: :             if( !(flag > 0) )
13: :                 break;
14: :                 flag = n % i;
15: :         }
```

とするとよい（16:, 17:行は空行）。これなら `while(){}` の外へ出ることが一目瞭然だ。記号 `!` は

否定演算子と呼ばれ、真偽を逆にする効果がある。すなわち、`flag > 0` が真なら`!(flag > 0)` は偽となって `flag = n % i;` が実行され、`flag > 0` が偽なら`!(flag > 0)` は真となって `break;` が実行され `}` の直後に出る。理解できただろうか。正直に言って、`break;` の出どころはややこしい。

EX. いまのように `if` 文で `{}` を用いない場合、`if(flag > 0)`、`flag = n % i;`、`break;` の順に書いたのではうまく動作しない。その理由を考えてみよ。

さてと。話の腰を折らないよう、11:行目に使われた `Math.sqrt()` には触れなかったので、いま触れておこう。**Java** で数学的処理を行うには `Math` クラスを必要とする。今回は `Math` クラスに含まれる `sqrt()` メソッド— `sqrt` 関数と考えてもよい—を使っているので、記述の仕方が `Math.sqrt()` となったのだ。

実はこれまでにもさんざん使ってきた `System.out.print;` 文も、`System` クラスに含まれる `out` 内に割り振られた `print()` メソッドを利用して画面出力をこなしていたのである。つまり **Java** では、いろいろな命令がいろいろなクラスに含まれていて、プログラマは、どのクラスのどんなメソッドを利用するかを考えながらプログラムするのだ。

そして、何らかのクラスやメソッドを利用したければ、`import;` 文によって事前に必要とするクラスを取り込む仕組みなのである。でも、プログラムをよく見てほしい。このプログラムでは `Math` クラスを `import` してないね。なのに `sqrt()` メソッドが使えてる。どうして？

その答えは、**Java** は自動的に `Math` クラスを読み込んでいるからなんだ。だったら最初から全部のクラスを読み込んでくれれば楽なのに、と思うよね。しかし、そうすると大いなる無駄を含んだアプレットが出来上がってしまう。**Java** はほどよいクラスだけを最初に準備するだけなのだ。