プログラミング基礎'00　＃2

久野　靖*

2000.6.10

0　はじめに

前回の皆様の感想を拝見しましたが、「理解できる」というご意見が多かったのでよかったです。とにかく回数が5回しかないので、その範囲で「やってよかった」という結果になるように、多少カバーする範囲に偏りが出るかも知れませんけど、テーマを絞ってその代り扱ったテーマはちゃんと理解してもらえるようにしようと思います。

まず前回の演習問題から抜粋して回答例と解説を掲載しますが、詳しく説明していると時間がなくなりますから、重要と思われるポイントや新しい内容に関わる部分だけ拾って説明します。あとは各自読んでおいてください。

今回の内容としては、前回の感想で要望が多かった「なぜ Java なのか?」という話題からはじめて、次に予告通りアプレットを扱います。また、これと併せて「前回の続きとして」、さまざまなオブジェクトを利用すること、および制御構造の追加としての for 文について学びます。

1　字下げその他について

演習 6a や 6c は 6b ができれば同じことだから略。6b の疑似コードは次の通り。

- 実数 x、y を入力する。
- x+y、x-y、x*y、x/y を出力する。

で、Java では次のようになる。

```
import java.io.*;

public class r1ex6b {
  public static void main(String args[]) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("x = "); System.out.flush();
    double x = (new Double(in.readLine())).doubleValue();
    System.out.print("y = "); System.out.flush();
    double y = (new Double(in.readLine())).doubleValue();
    System.out.println("x + y = " + (x+y));
    System.out.println("x - y = " + (x-y));
    System.out.println("x * y = " + (x*y));
    System.out.println("x / y = " + (x/y));
  }
}
```

*筑波大学大学院経営システム科学専攻

自分が作ったのと違う、と思いました? もちろん、それで当然であり、どっちが正解ということはない。ちょうど日本語で同じことを言うのにさまざまな言い方があるのと同じこと。ただし、スマートだとか短いとか分かりやすいとかそういった点で違いはあるかもしれない。美観の問題! だから自分の好みもとりまぜて、考えて選ぶこと。

その他注意してほしい点:

- まず、字下げ (左側に空白を入れること) をしない人がいるけど、これは絶対やめた方がよい。たとえば、

```
if(...) {          if(...) {
    ....               ....
    ....               ....
}                  }
```

  左のように書いてあれば、どこまでが「もし〜ならば」の範囲かすぐ分かる。しかし右のようにべったり揃えてしまうとそれが分からないし、たとえば間違って「}」をつけ忘れたり別の場所に入れてしまうともう大混乱になる。たかがスペースキーを打つ手間を惜しんで後で間違い探しに 1 時間も掛けるのは阿呆みたいでしょ? プログラムは「美しく」書こう。

- ついでながら、空白の空け方や改行のしかたが分からないという質問も頂いたので…Java をはじめ今日のプログラミング言語はほとんどが「フリーフォーマット」、つまり「名前や定数の途中でない限り自由に空白、改行を入れてもよい」ことになっている。つまり以下のどれでも同じ。

```
if(x>0){System.out.println("positive.");}
if(x > 0) { System.out.println("positive."); }
if ( x > 0 )
{
  System . out .
           println  (  "positive."  ) ;
}
```

  名前の「途中に」入れてはいけないので「println」を「print ln」などと空けるのはだめ。で、この自由度は何に使うべきか? それはもちろん、プログラムが「書きやすく読みやすく」なるように使うべき。

- 和、差、商、積をいったん変数に入れていない。もちろん、前回の例題のように変数に入れてもよいが、上のようにいきなり出力してもよい。一般に変数が書けるところには代りに任意の計算式を書いてもよい。ところで、式を丸かっこで囲んでいるのは、「`"...." + x + y`」とすると足し算は行われず全部文字列として連結されてしまうから。

2 枝分かれについて

演習 7a は 2 つの枝分かれですから例題とほとんど同じ。まず疑似コードを見よう。

- 実数 a、b を入力する。
- もし a $>$ b であれば、
- 　　max $\leftarrow$ a。
- そうでなければ、
- 　　max $\leftarrow$ b。
- 枝分かれおわり。
- max を出力する。

Java では次の通り。

```
import java.io.*;

public class r1ex7a {
  public static void main(String args[]) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("a = "); System.out.flush();
    double a = (new Double(in.readLine())).doubleValue();
    System.out.print("b = "); System.out.flush();
    double b = (new Double(in.readLine())).doubleValue();
    double max;
    if(a > b) {
      max = a;
    } else {
      max = b;
    }
    System.out.println("larger : " + b);
  }
}
```

しかし、次のような「別解」はどうだろう。

- 実数 a、b を入力する。
- max ← a
- もし b > max であれば、
- 　　max ← b
- 枝分かれおわり。
- max を出力する。

これの Java 版は次のとおり。

```
import java.io.*;

public class r1ex7a1 {
  public static void main(String args[]) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("a = "); System.out.flush();
    double a = (new Double(in.readLine())).doubleValue();
    System.out.print("b = "); System.out.flush();
    double b = (new Double(in.readLine())).doubleValue();
    double max = a;
    if(b > max) {
      max = b;
    }
    System.out.println("larger : " + max);
  }
}
```

どっちが好みですか? これもどちらが正解ということではない。

しかし 7b はもうちょっとややこしい。まず考えるのは、a と b の大きい方はどっちか決めて、それぞれの場合についてそれを c と比べるというもの。

- 実数 a、b、c を入力する。
- もし a > b であれば、
- 　もし a > c であれば、
- 　　max ← a。
- 　そうでなければ、
- 　　max ← c。
- 　枝分かれおわり。
- そうでなければ、
- 　もし b > c であれば、
- 　　max ← b。
- 　そうでなければ、
- 　　max ← c。
- 　枝分かれおわり。
- 枝分かれおわり。
- max を出力する。

うーむ大変だ。これを Java にしておく。

```
import java.io.*;

public class r1ex7b {
  public static void main(String args[]) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("a = "); System.out.flush();
    double a = (new Double(in.readLine())).doubleValue();
    System.out.print("b = "); System.out.flush();
    double b = (new Double(in.readLine())).doubleValue();
    System.out.print("c = "); System.out.flush();
    double c = (new Double(in.readLine())).doubleValue();
    double max;
    if(a > b) {
      if(a > c) {
        max = a;
      } else {
        max = c;
      }
    } else {
      if(b > c) {
        max = b;
      } else {
        max = c;
      }
    }
  }
  System.out.println("largest : " + max);
}
```

こうなると字下げしてないとごちゃごちゃになるでしょう? しかし字下げしてあってもこれはかなり苦しい。ときに、先の別解から発展させるとどうだろう?

- 実数 a、b、c を入力する。
- max ← a
- もし b > max であれば、
- 　　max ← b
- 枝分かれおわり。
- もし c > max であれば、
- 　　max ← c
- 枝分かれおわり。
- max を出力する。

この方がすっきりしているでしょう? Java でも次のとおり。

```
import java.io.*;

public class r1ex7b1 {
  public static void main(String args[]) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("a = "); System.out.flush();
    double a = (new Double(in.readLine())).doubleValue();
    System.out.print("b = "); System.out.flush();
    double b = (new Double(in.readLine())).doubleValue();
    System.out.print("c = "); System.out.flush();
    double c = (new Double(in.readLine())).doubleValue();
    double max = a;
    if(b > max) { max = b; }
    if(c > max) { max = c; }
    System.out.println("largest : " + max);
  }
}
```

今度はどちらが好みですか? 一般には、枝分かれの中に枝分かれを入れるよりは、枝分かれを並べるだけで済ませられればその方が分かりやすいといえる。なお、if 文が 1 行に書かれているが、どこで行を変えるかは自由に選べるので、この場合はこの方が見やすいと思ってこうしてみた。

3　多方向の枝分かれ

演習 7c は 3 通りに分かれるのだから、if の中にまた if が入るのはやむを得ない。しかし、ちょっと工夫すると分かりやすくなる。Java コードから見ていただく。

```
import java.io.*;

public class r1ex7c {
  public static void main(String args[]) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("x = "); System.out.flush();
    double x = (new Double(in.readLine())).doubleValue();
    if(x > 0.0) {
      System.out.println("positive.");
    } else if(x < 0.0) {
```

```
      System.out.println("negative.");
    } else {
      System.out.println("zero.");
    }
  }
}
```

これは実は最初の if の else のすぐ後ろに次の if がくっついた、という形をしているが、こういうパターンを利用するとプログラムが分かりやすくなる。順序が前後したが、疑似コードだと次のようになる。

- 実数 x を入力する。
- もし $x > 0$ ならば、
- 　　「positive.」と出力。
- そうでなくて $x < 0$ ならば、
- 　　「negative.」と出力。
- そうでなければ、
- 　　「zero.」と出力。
- 枝分かれおわり。

一般に「そうでなくて〜ならば、」は何回現われてもよい。またそのどれもが成り立たない場合は「そうでなければ」に来るが、この部分は不要ならなくてもよい。これを Java にする場合は次の形になる。

```
if(...) {
  ...
} else if(...) {
  ...
} else if(...) {
  ...
} else {
  ...
}
```

なお、これはあくまでも Java の if 文の「else の後にすぐ次の if をくっつけた」というパターンなだけで、特別な文というわけではない。

ところで、「さまざまな文字列を出力」するのに上の例では枝分かれのそれぞれの枝で直接 `System.out.println()` を呼ぶようになっていたが、メッセージを変数に入れておいて最後に出力するようにしてもよい。その場合、メッセージは「文字列」すなわち `String` 型の変数に格納することになる。

- 実数 x を入力する。
- もし $x > 0$ ならば、
- 　　mesg ← ”positive.”。
- そうでなくて $x < 0$ ならば、
- 　　mesg ← ”negative.”。
- そうでなければ、
- 　　mesg ← ”zero.”。
- 枝分かれおわり。
- mesg を出力。

```
import java.io.*;

public class r1ex7c {
```

```
  public static void main(String args[]) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("x = "); System.out.flush();
    double x = (new Double(in.readLine())).doubleValue();
    String mesg;
    if(x > 0.0) {
      mesg = "positive.";
    } else if(x < 0.0) {
      mesg = "negative.";
    } else {
      mesg = "zero.";
    }
    System.out.println(mesg);
  }
}
```

これも「どちらかが正しい」ということはない。

4 繰り返しについて

演習 8a であるが、これは前回説明したように 2 ずつ引いていけばよい。

- 整数 x を入力する。
- $x > 1$ である間繰り返し:
- $\quad x \leftarrow x - 2$
- ここまで繰り返し。
- もし $x = 0$ ならば、
- 「偶数」と出力。
- そうでなければ、
- 「奇数」と出力。
- 枝分かれ終わり。

Java で書くと次の通り。

```
import java.io.*;

public class r1ex8a {
  public static void main(String args[]) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("x = "); System.out.flush();
    int x = (new Integer(in.readLine())).intValue();
    while(x > 1) { x = x - 2; }
    if(x == 0) {
      System.out.println("even.");
    } else {
      System.out.println("odd.");
    }
  }
}
```

このような繰り返しの条件に注意。「x $>$ 1 の間繰り返す」という条件で、しかも繰り返しの中では x は小さくなる方向に変化するから、いつかは x が十分小さくなってこの繰り返しは確実に止まる。さらに、繰り返しが止まったときは「$x \leq 1$」が成り立っているから、つまり x は 1 であるか 0 であるかのいずれか、になっている。なのでそのどちらかを調べれば奇数か偶数かが分かる。つまり、繰り返しを使う時には

- 最終的に成り立って欲しい条件を決め、
- 繰り返しの内側ではその条件が成り立ちやすい方向に変化を起こす

ようにする。これらが守られていないと、繰り返しが無限に続いたり、繰り返しが終わった時に役に立つ値が計算されていない、ということになる。

たとえば、人生で金持ちになるための手順も同様である。

- 手持ちの金額が G 円未満である間繰り返し、
- 　　手持ちの金を増やす※。
- ここまで繰り返し。

ただし問題は、人生では※を確実に行う手順が知られていないことである。

ところで、x が正の整数でないと、当然このプログラムは正しく動作しない。それでいいのか、と思うかも知れないが、それは構わない。プログラムを作る時はあくまでも「プログラムが動作すべき条件が満たされた時に正しく動作するプログラムを作る」ことだけが求められる。余計なオマケのために労力を費すのがいいかどうかは必ずしも分からない。

ただし、自分でプログラムの動作を定める時には、正の整数以外でも扱いたいかどうか、十分考えてよいと思うように決めること。

5　for 文による繰り返しについて

演習 8c は 8b が分かれば同じこと。8b は要するに y を x 回足せばよい。これを行うのにいろいろな方法があるが、普通はもう 1 つ変数を用意して、これを 1 ずつ足しながら回数を数えていく。

- 整数 x、y を入力する。
- seki ← 0。
- i ← 0。
- i $<$ x である間繰り返し:
- 　　seki ← seki + y。
- 　　i ← i + 1。
- ここまで繰り返し。
- seki を出力する。

これを Java にすると次の通り。

```
import java.io.*;

public class r1ex8b {
  public static void main(String args[]) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("x = "); System.out.flush();
    int x = (new Integer(in.readLine())).intValue();
    System.out.print("y = "); System.out.flush();
    int y = (new Integer(in.readLine())).intValue();
    int seki = 0;
```

```
    int i = 0;
    while(i < x) {
      seki = seki + y;
      i = i + 1;
    }
    System.out.println("product is: " + seki);
  }
}
```

ここでもいくつか注意を。

- 上のコードでは「x 回繰り返す」のに、i を 0、1、2、…、$x-1$ まで変化させている。このように「0 から数える」のは計算機ではよく使う。
- まず、変数 seki を最初 0 にしておき、次に y ずつ増やしていることに注意。合計を求めるには「`seki = seki + y`」のようにして「足し込んで」行くのが定石。なお、これはよくでて来るパターンなので「`seki += y`」と書くこともできる。他に「引き込む」「掛け込む」等もあり。
- 特に「1 足す」「1 引く」はよく使うのでさらに短く「`++i`」「`--i`」と書くこともできる。

さて、上で出て来た i のように「数を数える」ための変数を「カウンタ」と呼ぶ。カウンタを 0、1、2、…、$N-1$ まで変化させて繰り返す、というのはとてもよく使うので疑似コードで次のように書くことにする。

- 変数 i を 0 から $N-1$ まで変化させながら繰り返し:
- ....
- 繰り返し終わり。

また、Java でもこの部分は while 文の代りに for 文を使って書くことが多い。

```
for(int i = 0; i < n; ++i) {
    ....
}
```

for 文は while 文の「親戚」だが、カウンタ用の変数の初期設定と毎回の更新 (加算) を一緒に書いて見やすくできる。なお、上のようにカウンタ変数の宣言を for の中に入れた場合は、その変数は for の内部でのみ使える。これを使って先の例題を書き直すと次の通り。

- 整数 x、y を入力する。
- seki ← 0。
- 変数 i を 0 から x-1 まで変化させながら繰り返し:
- seki ← seki + y。
- ここまで繰り返し。
- seki を出力する。

この方が見やすいでしょう? Java では次の通り。

```
import java.io.*;

public class r1ex8b1 {
  public static void main(String args[]) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("x = "); System.out.flush();
    int x = (new Integer(in.readLine())).intValue();
    System.out.print("y = "); System.out.flush();
```

```
    int y = (new Integer(in.readLine())).intValue();
    int seki = 0;
    for(int i = 0; i < x; ++i) {
      seki = seki + y;
    }
    System.out.println("product is: " + seki);
  }
}
```

6 演習 9

疑似コードは前回の資料に載っていたので略。

```
import java.io.*;

public class r1ex9 {
  public static void main(String args[]) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("x = "); System.out.flush();
    int x = (new Integer(in.readLine())).intValue();
    System.out.print("y = "); System.out.flush();
    int y = (new Integer(in.readLine())).intValue();
    while(x != y) {
      if(x > y) {
        x = x - y;
      } else {
        y = y - x;
      }
    }
    System.out.println("GCD : " + x);
  }
}
```

なぜこれで最大公約数が求まるのだろう? 次のように考えてみるとよい。なお、x と y は正の整数であるも t のします。

- $x = y$ であれば、最大公約数は x そのもの。当然ですね。
- $x > y$ であれば、x と y の最大公約数は $x - y$ と y の最大公約数に等しい。[1]
- したがって、$x - y$ を改めて x とおいて、x と y の最大公約数を求めれば酔い。
- $x < y$ の場合も同様。
- この手順の反復ごとに、x または y のどちらかがより小さくなるが、0 以下にはならない (大きい方から小さい方を引くから)。
- ということは、この反復は有限回で止まる。
- ということは、そのとき $x = y$ が成り立ち、x が一番最初の x と y の最大公約数に等しい。

[1] 証明: 最大公約数を G とおくと、x も y も G の整数倍なのだから、$x - y$ もまた G の整数倍である。ということは、G は $x - y$ と y の公約数である (最大かどうかはまだ分からない)。ところで、もし最大公約数で「なかった」とすると、最大公約数 $H(> G)$ が別にあるわけで、H は y の約数かつ $x - y$ の約数。ということは、H は $x - y + y = x$ の約数でもある。これは G が x と y の最大公約数であるということに矛盾する。従って G は $x - y$ と y の最大公約数でもある。

どうですか、繰り返しを使うときは「必ず止まって、止まった時には求める状況が成り立っている」ように設計する、という意味が分かります?

7 演習 9

これも疑似コードは略。

```
import java.io.*;

public class r1ex10 {
  public static void main(String args[]) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("x = "); System.out.flush();
    double x = (new Double(in.readLine())).doubleValue();
    double a = 0.0;
    double b = x;
    while((b - a) > 0.0000001) {
      double c = 0.5 * (a + b);
      if(c*c > x) {
        b = c;
      } else {
        a = c;
      }
    }
    System.out.println("square root of " + x +" : " + a);
  }
}
```

さて、このプログラムでなぜ「平方根を求める」ことができるか考えてみよう。ただし x は 1 より大きな数とする。

- x の平方根を R とすると、$0 \leq R < x$ である。
- $a = 0$、$b = x$ とおくのだから、$a \leq R < b$ である。
- $c = \frac{a+b}{2}$ としたとき、$c^2 > x$ であれば $a < R < c$ であるので、c を改めて b と置いたとき、$a \leq R < b$ である。そうでなければ、$c \leq R < b$ であるので、c を改めて a と置いたとき、$a \leq R < b$ である。
- $b - a$ は、反復ごとに a と b の平均 c を a、b のいずれかと置き換えるので、毎回半分ずつに小さくなっていく。ということは、いつかは必ず 0.0000001 以下になって停止する。
- このときも $a \leq R < b$ は成り立っているので、つまり a と R の差は 0.0000001、ということはその誤差以下で R が求まったと言える。

8 値とオブジェクト

前回もちょっと振れたが、重要な点なので再度値とオブジェクトの話をしよう。Java が扱うデータには大きく分けると「値」と「オブジェクト」の 2 種類があり、その区別は次のようになっている。

- 値 — 数値 (四則演算の対象になるもの) と、文字。四則演算程度の機能だけを持つ。単純なデータ。
- オブジェクト — さまざまな「もの」を表すデータ。込み入った構造や機能を持つことができる。クラスによって定義され (てい) る。言い替えればクラスとはオブジェクトの種類である。

値としては int(整数)、long(倍精度整数)char(文字)、float(実数)、double(倍精度 — 精度の高い — 実数)、などがある。一方、文字列などは複雑な構造を持つ (中に文字がたくさん詰まっている) のでオブジェクトで表す。ところが困ったことに、整数や文字などを表すオブジェクトも別途ある。これは「値」の方はデータの変換などの込み入った機能が入れられないため。そのような値とクラスを次に示しておく。クラス名は大文字で始まることに注意。

種別	値	クラス
真偽値	boolean	Boolean
バイト	byte	Byte
整数	int	Integer
倍精度整数	long	Long
文字	char	Character
実数	float	Float
倍精度実数	double	Double

基本的に、「値」に対してできることは各種の演算と自動的な文字列への変換だけで、それ以上の機能はすべて「クラス」の方の機能として用意されている。具体的な演算としては次のものがある。

```
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      ++  --
```

ビット演算とシフト演算は整数 (計算機内部では 2 進数で、つまり 0 と 1 の並びで表されている) をビット列とみなして演算する。また、論理演算は「〜かつ〜」「〜または〜」「〜でない」を表す。たとえば次のように使う:

```
a > b && a > c   → 「a が b より大きく、かつ a が c より大きい」
x != 0 || !(a > c) → 「x が 0 でないか、または a が c より大きくない」
```

次に、クラスやオブジェクトはどうやって使うかについて説明しておこう。まず、クラスは「オブジェクトの種類」に対応していることを上で述べた。あるクラスに属する個々のオブジェクトをそのクラスの「インスタンス」と呼ぶ。インスタンスを作るには「new クラス名 (...)」という形の式を使う。

クラスのさまざまな機能を利用するには「メソッド」を呼ぶ。メソッドは次の 2 種類がある。

```
クラス名.メソッド名(...)   ←クラスメソッド
式.メソッド名(...)         ←インスタンスメソッド
```

「式」は何らかのインスタンスを計算するものでなければいけない。クラスメソッドとインスタンスメソッドの違いは、前者がクラスに所属していてインスタンスと関係を持たないのに対し、後者はインスタンスに所属している (たとえば「私」オブジェクトの「名前を返す」メソッドの結果は「あなた」オブジェクトの「名前を返す」メソッドの結果とは違う) ことである。

WWW ブラウザを立ち上げ、GSSM の入口ページから「Java 2 Document」と記されたリンクをたどり、ドキュメントの中からさらに「Java 2 プラットフォーム API 仕様」のリンクをたどると、標準のクラス群としてどのようなものがあるかを記載したドキュメント (API ドキュメント) が見られる。ドキュメントは「パッケージ」と呼ばれる単位ごとに分かれている。たとえば PrintStream クラス (System.out には PrintStream のインスタンスが入っている) は java.io パッケージに入っている。すべてのクラス名は曖昧さがないようにパッケージ名をつけた「java.io.PrintStream」のような形で指定してもよい。

演習 1 API ドキュメント中の java.lang.Double の箇所を眺めてみよ。これまでに使った「new Double(...)」や「doubleValue()」の定義を確認せよ。

演習 2 API ドキュメント中の java.io.PrintStream の箇所を眺めてみよ。知っている/使ったことのあるメソッドの説明を読め。

9 なぜ Java 言語か?

ではここで、前回あまり詳しく触れなかった、Java 言語とはどういう言語で、なぜこの科目で Java 言語を扱っているのかについて簡単に説明しておこう。

Java はもともと、Sun Microsystems 社 (代表的な Unix ワークステーション/サーバのメーカ) がセットトップボックス (TV の上に載せるアダプタ) プロジェクト用に開発した。つまり、ネット経由でアダプタにさまざまなプログラムをその場でダウンロードしてくることで、「何にでもなるテレビ」を実現しようとしたわけである。そのため、Java は「どのようなプラットフォームでも動く」「安全な」言語としてデザインされた。そして、Java そのものは「何でも書ける (汎用の)」「オブジェクト指向言語」であるという点で、特定目的用の言語ではないという点も重要である。

実際には上記のプロジェクトは SGI に負けてポシャってしまったのだが、その直後の WWW の急速な普及 (いわゆる「インターネットブーム」でチャンスが訪れた。つまり、TV アダプタではなくてブラウザの中にプログラムをダウンロードして来られるようにして、これを「アプレット」(アプリケーショのちっちゃいもの、という意味) と名付けたわけである。当時のブラウザはゆっくりしたやり取りしかできる仕組みを持っていなかったので、アニメーションなどが自在に作れるアプレットは一躍注目を浴びて、「Java ブーム」が到来した。

現在では Java はその汎用言語としての特徴 (安全かつ計算機科学の進歩を素直に取り込んだオブジェクト指向の言語であると言う点) を活かして、アプレット以外にさまざまな用途に使われている。今プログラミングを学ぶとしたら、他の言語だとどういう問題があるか挙げておこう。

- Fortran、Pascal、COBOL、C — これらはオブジェクト指向言語でない。今学ぶならオブジェクト指向言語の方が多様な機能を容易に取り入れられるという点で有利。
- C++ — 代表的なオブジェクト指向言語ではあるが、言語としての設計がつぎはぎ的できたない。学ぶ上で複雑かつ罠に陥りやすい。
- VisualBasic、VBA — 特定目的の言語であり、なおかつ Microsoft 文明でしか動かない。Microsoft のセンスは信用できない。

そういうわけで、Java は C++などの代りに「何でも書ける」言語なわけだが、さらに利点として「さまざまなものを書くためのライブラリ」が非常に豊富に標準化されていることも挙げられる。それはこれから見て行けば納得されると思う。決して「アプレット用の言語」というわけではない。

10 アプレット

さて、ここまでに作ってきた Java プログラム — 「java」コマンドで動かす奴 — は「スタンドアロンアプリケーション」と呼ばれる種類のものである。これとは別に、見たことはある人が多いと思うが、Web ブラウザの画面内で動くような Java プログラムもあり、これを「アプレット」と呼んでいる。

アプレットも (スタンドアロン) アプリケーションも Java プログラムであることに変りはないが、アプレットの場合は「main はブラウザの中にあり、そこからあなたが書くクラス (アプレット) を呼び出して利用する」という点が異なる。つまり自分が main を書く場合はそのプログラムの形はどうにでも決められるが、アプレットでは main は既にあるので、そこから「呼ばれる」形は予め決まっていて、それに合わせて自分のコードを書く必要がある。このような、「あらかじめ骨組みが決まっているものに、そのプログラム固有の部分をはめ込むことで全体を完成させる」やり方を「フレームワーク」(framework、骨組みの意味) と呼ぶ。

具体的にはアプレットは次のような書き方をすることになる。

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

public class 名前 extends Applet {
  型 変数 = 初期値;
  型 変数 = 初期値;
  ...

  public void paint(Graphics g) {
    文…
  }
  その他必要なメソッド定義…
}
```

もう少し細かく説明しよう。

- まず、これまでと使うライブラリが違うので、import 文で applet パッケージと awt パッケージを必ず指定する (他に必要なものがあれば import 文はいくつでも追加してよい)。
- 次に、アプレット用に作るクラスはクラス名の後に「extends Applet」という指定が必要 (その具体的な意味は長くなるのでもっと後で説明する)。
- アプレットに限らないが、クラスではメソッドを複数定義して使うことができ、それらのメソッドが共通に使う変数をまず書く。
- アプレットではさまざまなことができるが、最低限「何かを画面に描く」ことはしないと何も表示されないので意味がない。そのため、画面に描くためのメソッドを必ず用意する。これが paint() である。
- paint() は、ブラウザがアプレットに画面を描かせようと思った時何回でも繰り返し呼び出される。だから、この中であまりたくさん仕事をすると画面表示が遅くなってうまくない。呼び出されるとき、paint() には引数として Graphics クラスのインスタンスが渡される。このオブジェクトのメソッドを呼び出すことで、画面にさまざまな描画を行うことができる。
- アプレットでは paint() 以外にもいくつかメソッドを定義することができる。これらはブラウザ側からアプレットを制御するために使用される。代表的なものを挙げておく。

  ```
  public void init() { ... }  --- 初期設定のための処理
  public void start() { ... } --- ページが表示される時の処理
  public void stop() { ... }  --- ページが見えなくなる時の処理
  public void destroy() { ... } --- アプレットを破棄するときの処理
  ```

  これらはまた必要になった時に再度説明する。

では、最初のアプレットの例題を見てみよう。

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

public class R2Sample1 extends Applet {
  Font fn = new Font("Helvetica", Font.BOLD, 24);
  public void paint(Graphics g) {
    g.setFont(fn);
    g.setColor(new Color(100, 0, 255));
    g.drawString("Hello, World", 30, 30);
```

```
    }
  }
```

やってることは次の通り。

- まず、表示に使うフォント (字形) を大きなものにするため、Font オブジェクトを作成して変数 fn に入れる。これはアプレットの実行開始時に最初に行う。あとは全部 paint() が呼ばれたときの動作。
- paint() の中では、まず先に用意したフォントを使うように設定。
- 次に、描画に使う色を設定。
- そして最後に、画面上の位置を指定して文字列を表示する。

さて、アプレットを表示するためには、アプレットを埋め込んだ表示用の HTML ファイルが必要である。アプレットしか入っていない簡単なテスト用の HTML ファイルを示しておこう。

```
<html>
<head><title>sample</title></head>
<body>
<h1>R3Sample1</h1>
<applet code="R2Sample1.class" width=300 height=200></applet>
</body>
</html>
```

ここではこのファイルもアプレットの名前に合わせて R2Sample1.html というファイル名をつけることにしておく。

演習 3 以下の手順に従って、上の例題アプレットを含んだページを作ってブラウザで表示してみよ。

1. まず、ホームディレクトリの下の「.public_html」というディレクトリへ行く。

   ```
   cd
   mkdir WWW   ←まだ作ってない人だけ
   chmod go+rx WWW ←今作った人だけ
   cd WWW
   ```

2. エディタでこのディレクトリの下にファイル R2Sample1.java を作成する。(別のディレクトリに作っても無意味だから注意!)
3. 「javac R2Sample1.java」でコンパイルする。
4. エディタでこのディレクトリの下にファイル R2Sample1.html を作成する。(別のディレクトリに作っても無意味だから注意!)
5. ファイルを作成したりコンパイルしたあとは必ず、「chmod a+r *」を実行して全部のファイルを他人に見えるように設定する」
6. 完成したら Netscape でこのページを開く。URI は次の通り

   `http://smm/~ユーザ ID/R2Sample1.html`

   または、ブラウザの代わりにアプレットをテストするコマンド appletviewer を使って、「`appletviewer R2Sample1.html`」のようにファイルを指定して起動する。

なお、Java コードを手直しした場合には「Shift キーを押しながら再読み込み」させないと Netscape は新しいプログラムを読み込んでくれないので注意すること。

演習 4 次のことを行え。

a. java.awt.Graphics のドキュメントを見て、drawString() の使い方をチェックせよ。次に上の例題を修正して、文字の表示位置を変更してみよ。

b. java.awt.Color のドキュメントを見て、色の作り方をチェックせよ。次に上の例題を修正して、文字の色を変更してみよ。

c. 例題では文字列を 1 回しか書いていなかったが、同じ文字列を「場所を変えながら、さらに色も変えながら複数回描く」ようにしてみよ。なるべくならループを使うこと。

演習 5 java.awt.Graphics のドキュメントを見ると、円、矩形、線分、楕円などさまざまなものを表示するメソッドがあることが分かる。どれか 1 つ好きなものを選び、上の例題に追加してその図形も描くようにしてみよ。文字とは色を変えること。

11 配列

さて、ループを使って「規則的な処理を反復する」方法は既に学んだが、実はループはそれと対になる「規則的なデータの並び」と一緒に使うことでずっと役に立つものがある。数学でも $x_1, x_2, \ldots$ のように変数の並びを扱うことで大量のデータを 1 つの式で表せるが、それと同様に考えればよい。

プログラミング言語の用語では、「整数の並んだもの」や「実数の並んだもの」等、一般に「同じ型が並んだもの」のことを「配列」と呼ぶ。Java では配列の型は元の型の後ろに「`[]`」をつけて表す。たとえば「`int[]`」は整数の並んだ配列、「`String[]`」は文字列の並んだ配列ということになる。

次に、配列を使うにはその配列型の変数を宣言した上で、大きさを指定して配列オブジェクトを用意しなければならない。具体的には次のような感じになる。

```
int[] a = new int[100];  ←要素数 100 の配列を用意
```

「new」を使うことから分かるように、配列も一種のオブジェクトである (ただし書き方がやや特別な形になっている)。

いちど用意してしまえば、配列の個々の要素は 1 つの変数と同様に扱える。ここで「どの要素か」を指定するのに `[...]` の中に式を書いて指定する (これを添字と呼ぶ — 数学では x_i のように小さい文字を使うが、プログラムでは小さい文字が使えないため)。たとえば上の例だと `a[0]`〜`a[99]` という要素があることになる (例によって 0 番目から数えるのに注意)。

たとえば、`Graphics` オブジェクトのメソッド `fillPolygon()` は「任意の多角形を塗りつぶす」機能を提供するが、多角形というのは要するに点の並びだから点の X 座標、Y 座標をそれぞれ整数型の配列の形で受け取るようにできている。これを使って三角形を描いてみよう。

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

public class R2Sample2 extends Applet {
  public void paint(Graphics g) {
    int[] x = new int[100];
    int[] y = new int[100];
    g.setColor(new Color(100, 0, 255));
    x[0] = 100; y[0] = 100;
    x[1] = 200; y[1] = 100;
    x[2] = 100; y[2] = 200;
    g.fillPolygon(x, y, 3);
  }
}
```

配列はなんとなく 100 個ぶんの領域を用意しているが、ここでは 3 個ぶんしか使っていない (本当は 3 個ぶんだけ用意する方が「お行儀がよい」けど例題だから)。

なお、「new 型 [N]」は「まだ中身の入っていない N 個ぶんの領域を持った配列オブジェクトの生成」だったが、初期値を指定したければ「new 型 []{ 式, 式, …} 」という形で指定できる。この場合、書いた式の数だけの要素を持つ配列オブジェクトができ、式がそれぞれの要素の初期値として格納される。

12 整数と実数の行き来

では次に、もうちょっと計算機らしく、ループを使って「正八角形」を描いてみよう。このような座標の計算をするときは、整数で計算すると切捨て誤差だらけになるし、sin や cos を計算するのは実数でないとメソッドが用意されていないので、実数を駆使することになる。ただし、最後に座標を指定するところではこれまで通り整数値を指定する。ということは、整数と実数の行き来が必要になるわけだ。まずはプログラムを見ていただく。

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

public class R2Sample3 extends Applet {
  public void paint(Graphics g) {
    int[] x = new int[100];
    int[] y = new int[100];
    g.setColor(new Color(100, 0, 255));
    double cx = 150.0;
    double cy = 100.0;
    for(int i = 0; i < 8; ++i) {
      double rad = (2.0 * Math.PI) * i / 8;
      x[i] = (int)(cx + 80.0 * Math.cos(rad));
      y[i] = (int)(cy + 80.0 * Math.sin(rad));
    }
    g.fillPolygon(x, y, 8);
  }
}
```

では要点を説明しよう。

- 32 ビットの整数型は int、64 ビットの実数型は double。ほかに 64 ビット整数型 long、32 ビット実数型 float もある。
- 「10」のように小数点のない定数は int、「10.0」のように小数点があると double。なお、「10l」のように小文字の L をつけると long、「10.0f」のように小文字の F をつけると float。
- 演算でこれらの型をまぜて使ったり、変数に入れたりするときには、「よりビット数の多い型」「整数から実数」へは自動的に変換してくれる。
- 逆に「ビット数の少い型」「実数から整数」へは「キャスト」と呼ばれる次の書き方を使う。

  (型名)(式) --- 例: (int)(3.14f)、(float)(3.14 + x)

  つまりキャストは「ここで情報が落ちるのは分かっているが、とにかくそうやってくれ」という意思表示のようなもの。

なお、ついでながらクラス Math はクラス変数やクラスメソッドとして三角関数、定数などをひととおり提供してくれるものである。

演習 6 java.lang.Math クラスのドキュメントを眺めて上の例題に出て来るメソッドや変数を確認せよ。

演習 7 上の 2 つの例題をそのまま打ち込んで動かせ。

演習 8 次のようなアプレットを作れ。

a. 正 5 角形を描く。
b. 星型を描く。
c. ハート型を描く。
d. スイスの国旗 (赤地に白十字) を描く。

13 メソッド

ここまで、「メソッド」とはどういうものかあまり説明しないままとりあえずやって来た。ここでまとめて説明しよう。メソッドが「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、ということはお分かりになっていることと思う。

ここまで出て来た例題はどれもさほど複雑ではなかったので、すべての動作を 1 つのメソッドに入れてしまっても問題なかった。なので、アプリケーションでは `main()`、アプレットでは `paint()` というメソッドだけを作ればよかった。

しかし、もっと動作が込み入って来ると、自分の書く動作も複数のメソッドに分けた方が見通しがよくなる。たとえば、アプレットで「三角形」を描こうと思うと、fillPoligon() というメソッドが利用できるが、これは一般に多角形を描くためのものなので、座標を配列として渡さなければならず、その準備を含めるとかなり長い処理が必要になる。三角形を描くごとにそういう長い処理を書くとプログラムがごちゃごちゃで分かりにくくなる。

そこで、「三角形を描く」というメソッドを定義する。そうすれば、三角形が必要なところでは、どこでもこのメソッドを呼ぶだけで済む。そして実際に三角形を描くこちゃごちゃした作業は、そのメソッドの中で 1 処理を記述すればよい。具体的に見ていただこう。

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

public class R2Sample4 extends Applet {
  public void paint(Graphics g) {
    fillRegularPolygon(g, new Color(100, 0, 255), 70.0, 60.0, 7, 50.0);
    fillTriangle(g, new Color(255, 100, 0), 100, 100, 100, 200, 200, 100);
  }
  public static void fillTriangle(Graphics g, Color c0,
      int x0, int y0, int x1, int y1, int x2, int y2) {
    g.setColor(c0);
    g.fillPolygon(new int[]{x0, x1, x2}, new int[]{y0, y1, y2}, 3);
  }
  public static void fillRegularPolygon(Graphics g, Color c0,
      double cx, double cy, int n, double r) {
    int[] x = new int[n];
    int[] y = new int[n];
    for(int i = 0; i < n; ++i) {
      double rad = (2.0 * Math.PI) * i / n;
      x[i] = (int)(cx + r * Math.cos(rad));
      y[i] = (int)(cy + r * Math.sin(rad));
```

```
      }
      g.setColor(c0); g.fillPolygon(x, y, n);
    }
  }
```

ところで、これまでインスタンスメソッドは必ず「式.メソッド名(...)」で呼び出す、という説明をしてきたが、クラスの中で同じクラスのメソッドを呼び出す場合には単に「メソッド名(...)」で呼び出すこともできるのだった。

ところで、「三角形を描く」メソッドを定義したとしても、いつも同じ場所に同じ色の三角形しか描けないのでは役に立たない。さまざまな場所に、さまざまな色の三角形を描きたいからメソッドを用意するわけである。

このため、メソッドには「引数」ないし「パラメタ」と呼ばれる仕組みが備わっている。つまり、メソッドを呼ぶ時には、呼ぶ側で「これこれの座標に、これこれの色で」という情報をつけて呼ぶ。これを「実引数」という。そして、メソッドの先頭部分で実引数を受け取るための「変数のようなもの」の宣言を並べる。これを「仮引数」という。実際にメソッドの本体が実行されるときは、1番目の仮引数(の名前を持った変数)には1番目の実引数の値、2番目の仮引数(の名前を持った変数)には2番目の実引数の値、…が入っていて、それを自由に参照できる。これを利用することで、`fillTriangle()`や`fillRegularPolygon()`は呼ばれるごとに色や位置の違う三角形や正多角形が描けるわけである。

演習 9 上の例題を打ち込んでそのまま動かせ。

演習 10 演習9で打ち込んだものに次のようなメソッドを追加し、それらを呼んでさまざまな絵を描くアプレットを作れ。特に指定がない場合、どのようなパラメタを持たせるかも自分で考えて決めること。

a. 指定した中心位置、半径、色の円を描く。
b. 指定した位置、大きさの星型を描く。
c. 指定した位置、大きさの矩形を描く。
d. ハロウィンの顔を描く(ヒント: まず円を描き、その上に別の色で三角形の目と口を描く)。
e. 三角屋根で窓のある家を描く(ヒント: 矩形を2つと三角形を組み合わせる)。

できるところまででよいが、これがもし全部できたら、家の向こうに月が出て満天の星がきらめいている絵を描いてみよう。