第3章　Javaの制御構造

Java の制御構造の構文（**if** 文、**for** 文、**while** 文）は C と全く同じである。ここでは制御構造の復習を兼ねて、これらの制御構造を使った例題を取り上げる。また、Java に特有の事柄も途中でいくつか紹介する。

3.1　式文・複合文

まず、C や Java では、＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿をつけた形が最も基本的な文（式文）である。関数呼出し文や代入文もこの形になる。さらに、複数の文を単に並べ、その周りを＿＿＿＿＿＿（「{」と「}」）でくくったものも文法的に文として扱われる。

式文

式;

複合文

{ 文$_1$ ... **文**$_n$ **}**

複合文では、文は単純に前から順番に実行される。通常、C や Java のプログラムでは、1 行の 1 つの文を書くが、単に見やすさのためであり、改行や空白の個数は、基本的には＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿。C や Java はレイアウトフリーの言語である。

もちろん、人間にとってプログラムを読みやすくするためには、改行や字下げ（＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿）を適切に行なうことが重要である。

3.2　条件判断

if 文

if（条件式）文$_1$
if（条件式）文$_1$ **else 文**$_2$

条件式が成り立てば文$_1$を実行する。そうでなければ何もしない（1 番めの形式）。文$_2$を実行する（2 番めの形式）。文$_1$, 文$_2$は、当然ブロック（“{” と “}” で括った文の並び）でも良い。

なお、条件式の型は＿＿＿＿＿＿型である。`boolean` 型は `draw3DRect` や `fill3DRect` の引数としても用いられていたが、＿＿＿＿か＿＿＿＿の 2 つの値を取り得る型である。C 言語と異なり整数型（`int` 型）とは区別されている。このため `while (1) ...` のような文はエラーとなる。

問 3.2.1 *int* 型と *boolean* 型を区別することの長短をまとめよ。

.....
.....
.....
.....
.....

例題 3.2.2
時間の足し算を行なう。
ファイル AddTime.java

```
import javax.swing.*;
import java.awt.*;

public class AddTime extends JApplet {
  int hour1, minute1, hour2, minute2;
  public void init() {
    hour1   = Integer.parseInt(getParameter("Hour1"));
    minute1 = Integer.parseInt(getParameter("Minute1"));
    hour2   = Integer.parseInt(getParameter("Hour2"));
    minute2 = Integer.parseInt(getParameter("Minute2"));
  }
  public void paint(Graphics g) {
    int hour, minute;
    // まず単純に足し算
    hour   = hour1+hour2;
    minute = minute1+minute2;

    if (minute>=60) {          // 繰り上がりの処理
      hour++;
      minute-=60;
    }
    // 結果を出力
    g.drawString("答えは "+hour+"時間 "+minute+"分です。", 30, 25);
  }
}
```

例えば、2 時間 45 分と 1 時間 25 分を足すと、そのままでは答が 3 時間 70 分になってしまう。分の部分が 60 以上になった時は繰り上げの処理を行なう処理を行なう必要がある。

注意: Java では、＿＿＿＿＿を用いて String と String、String と int などを連接することができる。（このため C のように %d, %s などを使った書式指定は必要ない。）
また、＿＿＿＿＿＿＿＿＿＿＿＿は文字列から整数に変換するためのメソッド（クラスメソッド — 後述）である。

3.3 繰り返し

for 文, while 文

while（条件式 $_1$）文 $_1$
for（式 $_1$; 式 $_2$; 式 $_3$）文 $_1$

while 文は条件式 $_1$ が成り立つ間、文 $_1$ の実行を繰り返す。
for 文はループに入る前に、まず式 $_1$ を評価する。式 $_2$ が成り立つ間、文 $_1$、式 $_3$ の実行を繰り返す。

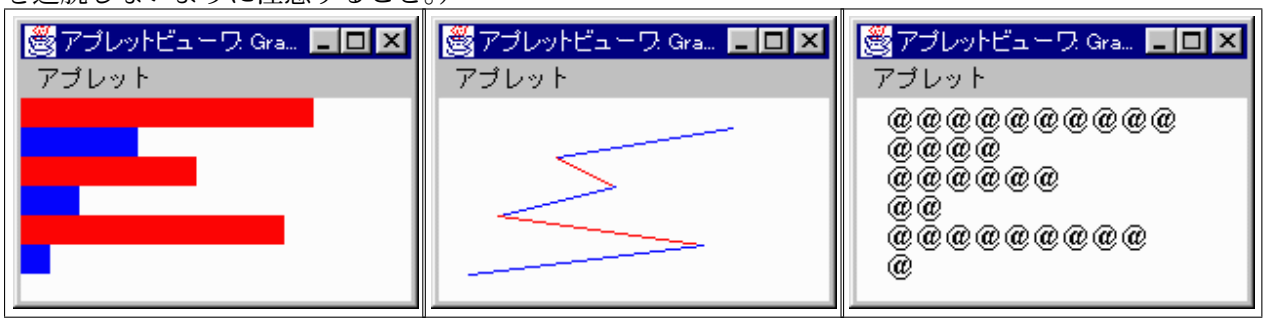
例題 3.3.1 グラフの描画
整数のデータを与え、そのデータの棒グラフを描く。

ファイル Graph.java

```
import javax.swing.*;
import java.awt.*;

public class Graph extends JApplet {
  int[] is = {10, 4, 6, 2, 9, 1};
  Color[] cs = {Color.red, Color.blue};
  int scale = 15;

  public void paint(Graphics g) {
    int i;
    int n = is.length;              // 配列の大きさ

    for (i=0; i<n; i++) {
      g.setColor(cs[i%2]);            // %は余り
      g.fillRect(0, i*scale, is[i]*scale, scale);
    }
  }
}
```

配列オブジェクトの *length* というメンバ *(?)* によって配列の大きさ（要素数）を知ることができる。これも *C* 言語と異なる点である。*for* 文の中のブロックは変数 *i* が *0*〜*n-1* まで変化する間、繰り返される。

問 3.3.2 与えられた数値データから折れ線グラフを生成するアプレットを書け。（配列の添字が範囲を逸脱しないように注意すること。）



棒グラフ　　折れ線グラフ　　キャラクターによるグラフ

（**注:** *n* 個の点を結ぶ線は *n-1* 本）

例題 3.3.3 *StringTokenizer* による文字列の分割
配列のデータをアプレットのパラメータとして渡せるように、*Graph.java* を拡張する。
Graph.html からグラフの数値のデータを空白で区切って渡せるようにする。

ファイル Graph.html

```
<html>
<head></head>
<body>
<applet code="Graph.class" width="200" height="200">
<param name="ARGS" value="10 4 6 2 9 1"> <!-- 数を空白で区切って渡す -->
</applet>
</body>
</html>
```

ファイル Graph.java

```
import javax.swing.*;
import java.awt.*;
import java.util.StringTokenizer;    // この import 文が新たに必要となる

public class Graph extends JApplet {
  ...

  public void init() {
    String args = getParameter("ARGS");
    StringTokenizer st = new StringTokenizer(args, " "); // 区切りは空白
    int i;
    int n = st.countTokens();      // いくつトークンがあるか

    is = new int[n];
    for(i=0; i<n; i++) {
      is[i] = Integer.parseInt(st.nextToken()); // トークンを整数に変換
    }
  }
 ...
}
```

ここでは、文字列を分割するために＿＿＿＿＿＿＿＿＿＿クラス[1]を用いた。このため import 文を 1 行追加している。 このクラスの nextToken メソッドを使ってスペースで区切られた形の文字列を分割し、さらに Integer.parseInt で文字列から整数へ変換している。上の init メソッドは、空白で区切られた文字列を配列に変換する典型的な方法である。 StringTokenizer のコンストラクタの 2 番 めの引数は、区切りに使用する文字である。これを","に変更すると、コンマで区切られた文字列を分割することができる。 また 2 番めの引数を省略すると空白文字（タブ・改行を含む）が区切り文字として使われる。

new オペレータは配列を生成する時にも使用することができる。＿＿＿＿＿＿は、動的に長さ n の int の配列を生成する式である。

[1] *(JDKDIR)*/docs/ja/api/java.util.StringTokenizer.html を参照

例題 3.3.4 時間のデータを *"9:45 12:35 4:42"* というように、空白で区切ってパラメータとして渡し、その時間の合計を表示するアプレットを書け。

ファイル AddTime2.java

```
import javax.swing.*;
import java.awt.*;
import java.util.StringTokenizer;

public class AddTime2 extends JApplet {
  int[] t = {0,0};   // 初期値 0時間 0分

  int[] addTime(int[] t1, int[] t2) { // 時間の足し算を関数として定義する。
    int[] t3 = new int[2];   // 時間を大きさ 2の配列で表す。

    t3[0] = t1[0]+t2[0];
    t3[1] = t1[1]+t2[1];
    if (t3[1]>=60) {  // 繰り上がりの処理
      t3[0]++;
      t3[1]-=60;
    }

    return t3;       // 新しい配列を返す。
  }

  public void init() {
    String args=getParameter("Args");
    StringTokenizer st = new StringTokenizer(args, " ");

    while (st.hasMoreTokens()) {     // まだトークンが残っているか?
      String s = st.nextToken();
      StringTokenizer st2 = new StringTokenizer(s, ":");
                            // ':'が時と分の区切り
      int[] time = new int[2];

      time[0] = Integer.parseInt(st2.nextToken());
      time[1] = Integer.parseInt(st2.nextToken());
      t=addTime(t, time);
      // addTimeの呼出し前に tに入っていた配列は不要になる。
      // あとで GCされる。
    }
  }

  public void paint(Graphics g) {           // 結果を出力
    g.drawString("答えは "+t[0]+"時間 "+t[1]+"分です。", 30, 25);
  }
}
```

AddTime2.javaでは時間の足し算の処理はメソッドaddTimeとして独立させた。addTimeはその中で配列を確保して戻り値に用いている。return文の書き方はCと同様である。

return 文

return 式;

return 文は、メソッドの戻り値として、式を計算した結果を返す。

new オペレータは、C の _____に近い働きをする。また、このようにして確保された配列は、init の中で addTime を呼ぶ時に次々と捨てられるが、これは _____（_____, GC）によって自動的に回収される。（C のように free による明示的なメモリの解放は必要ない。）GC のある言語ではこのように次々と新しいデータを生成して、古いデータを捨てるというスタイルが可能になる。

例題 3.3.5 正多角形の描画
整数 *n* をパラメータとして受け取り、正 *n* 角形を描画する。
ファイル N_gon.java

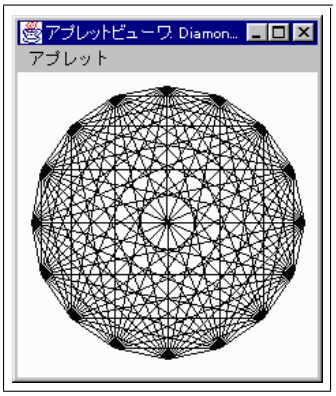
```
import javax.swing.*;
import java.awt.*;

public class N_gon extends JApplet {
  int n;
  int sc = 100;

  public void init() {
    n = Integer.parseInt(getParameter("NumPoints"));
  }

  public void paint(Graphics g) {
    int i;
    double theta1, theta2;
    for(i=0; i<n; i++) {
      // 単位 ラジアン
      theta1 = Math.PI*2*i/n;       // 360*i/n 度
      theta2 = Math.PI*2*(i+1)/n;  // 360*(i+1)/n 度
      g.drawLine((int)(sc*(1+Math.cos(theta1))), (int)(sc*(1+Math.sin(theta1))),
                 (int)(sc*(1+Math.cos(theta2))), (int)(sc*(1+Math.sin(theta2))))
    }
  }
}
```

Math.PI は 円周率 π （=3.1415 . . . ）、Math.sin, Math.cos は正弦、余弦関数である。これらは、通常のインスタンス変数やメソッドと異なり、Math というクラス名を介してアクセスされる。これらをそれぞれ、_____、_____（あるいは static メソッド）と呼ぶ。クラス変数、クラスメソッドは特定のオブジェクトに関連づけられない（つまりインスタンス変数に依存しない）、クラスで一つに定まる変数・メソッドである。（クラス変数は C の大域変数・クラスメソッドは C の通常の関数とほとんど同じものであると考えて良い。）クラス変数、クラスメソッドを定義する時は、 修飾子 _____を付ける。

問 3.3.6 *sin, cos* などの数学関数のグラフを描くアプレットを書け。
参考: (JDKDIR)/docs/ja/api/java.lang.Math.html

問 3.3.7 正 *n* 角形のすべての頂点を結んでできる図形（ダイアモンドパターン）を描画するアプレットを書け。

問 3.3.8 色のグラデーション（*2* 次元 — 縦方向と横方向が別の色に変わる）を作成するアプレットを書け。



ダイアモンドパターン

2 次元のグラデーション

1 次元のグラデーション

（参考） *1* 次元のグラデーション

ファイル Gradation1.java

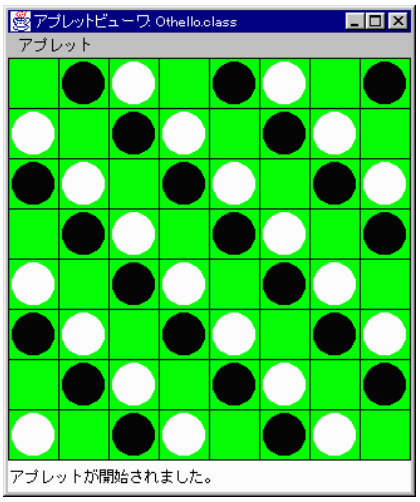
```
import javax.swing.*;
import java.awt.*;

public class Gradation1 extends JApplet {
  int scale = 4;

  public void paint(Graphics g) {
    int i;

    for (i=0; i<64; i++) {
      g.setColor(new Color(i*4, 0, 255-i*4));
      g.fillRect(i*scale, 0, scale, scale*10);
    }
  }
}
```

3.4 多次元配列

例題 3.4.1 *int* 型の *8*× *8* の大きさの配列の配列を調べて、*1* なら白丸、*2* ならば黒丸を画面上の対応する位置に描画する。

ファイル Othello.java

```
public class Othello extends JApplet {
  int scale = 40;
  int space = 3;
  int[][] state =
        {{0,1,2,0,1,2,0,1}, {2,0,1,2,0,1,2,0}, {1,2,0,1,2,0,1,2},
         {0,1,2,0,1,2,0,1}, {2,0,1,2,0,1,2,0}, {1,2,0,1,2,0,1,2},
         {0,1,2,0,1,2,0,1}, {2,0,1,2,0,1,2,0}};

  public void paint(Graphics g) {
    int i,j;

    for (i=0; i<8; i++) {
      for (j=0; j<8; j++) {
        g.setColor(Color.green);
        g.fillRect(i*scale, j*scale, scale, scale);
        g.setColor(Color.black);
        g.drawRect(i*scale, j*scale, scale, scale);
        if (state[i][j]==1) {
          g.setColor(Color.white);
          g.fillOval(i*scale+space, j*scale+space,
                     scale-space*2, scale-space*2);
        } else if (state[i][j]==2) {
          g.setColor(Color.black);
          g.fillOval(i*scale+space, j*scale+space,
                     scale-space*2, scale-space*2);
        }
      }
    }
  }
}
```



2 次元配列（配列の配列）を宣言するには、上のように `[]` を *2* つ重ねる。（*3* 次元以上も同様） *C* 言語の場合のように次元を宣言する必要はない。*`state`* は配列の配列で、例えば、*`state[0][1]`* は、*0* 番めの配列`{0,1,2,0,1,2,0,1}`の *1* 番めの数だか ら *1* である。つまりこの位置（*0* 列めの *1* 行め）には白丸が描画される。

注意: なお、Java の 2 次元配列と C の 2 次元配列はメモリ上の配置の仕方が異なる。（もっとも Java でメモリ上の配置を意識する必要はほとんどない。）このた め Java では C では許されない次のような 2 次元配列（異なるサイズの配列が混在している）

```
int[][] xss = {{1}, {1,2}, {1,2,3}};
```

も使用できる。

キーワード　`if` 文, `if`〜`else` 文, `boolean` 型, `Integer.parseInt` メソッド, `while` 文, `for` 文, 配列、`length` メソッド, `StringTokenizer` クラス, クラス変数, クラスメソッド, `static`, `Math` クラス、多次元配列