

Chapter 10. 繰返しと条件分岐の続き

10.1. for文を使った繰返し

■for文の書き方

while文の場合、繰返しの中でしか使わないループ変数の宣言やその最初の値を代入する場合など、while文が始まる前に独立して書かなければなりません。これらの不便を解消してくれる強力な繰返しのための構文が存在します。それが以下に述べるfor文です。Java言語のfor文は、内部で変数の宣言もできます。

▼for文の書式

```
for ( 最初に行なうこと ; 継続条件 ; 繰り返すときに行なうこと ){
    繰り返したい内容
}
```

書式では、for文の4つの部分がすべて記述されていますが、どの部分も、省略することが可能です。ただし、継続条件が省略されたときは、そこに**true**が書いてあるものとみなします。この場合、永遠に繰返すこととなるでしょう。

★for文の意味

それぞれの部分について、もう少し詳しく実行の過程を追ってみましょう。

1. 最初に、繰返しを行なう前に「最初に行なうこと」の式が評価されます。これは、繰返しを1つもしない場合でも実行されます。
2. 次に、「継続条件」を満足しているかチェックする。条件式を評価し、真（true）であれば繰返しを継続しますし、偽（false）であれば繰返しを止めて次の処理に制御を移します。
3. 繰返しを行なうときは、まず「繰り返したい内容」の方を先に実行します。そして、次の繰返しに移る前に「繰り返すときに行なうこと」の部分を評価します。
4. 以降の繰返しは、2.のところから行なわれます。

以上をfor文の各部分をABCDという形で分け、制御の流れを図示しますと、次のようになります。

```
for ( A; B; C ){
    D;
}
```

A→B→D→C→B→D→C→ B→D→C→B→次へ
↑
B→D→C→が何回か繰り返される

★初期値設定とループ変数

for文の「最初に行なうこと」の部分には、変数の宣言ができます。変数の宣言をする場合は、初期値を代入する必要があります。この変数を使って、10回の繰返しをする記述の仕方をみてみましょう。

```
for ( int count=0; count < 10; count ++ ){
    System.out.println( "Count up: " + count );
}
```

```
}
```

上の繰返しは、変数`count`を0から9まで変化させて、計10回の繰返しを行なわせています。毎回の繰返しの中で、変数の値を端末画面に出力しています。なお、for文の中で宣言された変数は、for文の繰返しのブロック内でしか用いることができません。

```
for ( int x=1; x<10; x++){
    sum = x + 5;    // このブロックの中では、変数xを用いることができる
}
int y = x + 10;    // エラー、繰返しが終わった後は、変数xを用いることができない
```

★それぞれの値を設定する目安

for文のそれぞれの部分に関して、さまざまな記述ができますが、たとえばループ変数の値を繰返しする中で増やしていく場合は、次のように考えるといいでしょう。

```
for ( int i = 初期値; i < 上限値 ; i += 間隔 ) {
    .....
}
```

このようにすると繰返しをする回数は、（上限 - 初期値）／間隔で求められます。ループ変数と上限値との比較が不等号で行なわれていることに注意してください。たとえば、次の繰返しは、7回繰返しをします。

```
for (int w = 10; w < 150; w += 20) {
    System.out.println( "The value of w is " + w + ".");
}
```

★for文を使った様々な記述例

以下のプログラムの断片は、それぞれfor文を使った例を示しています。初期値や上限値、不等号や等号の使い方、あるいはループ変数をどのように辺かせているか注意してみてください。

1から10まで値の二乗を表示する

```
for (int i = 1; i <= 10; i++) {
    System.out.println( "square of " + i + " is " + i * i);
}
```

50から-50までの値について、10間隔で表示する。

```
for (int w = 50; w >= -50; w -= 10) {
    System.out.println( "Value of w: " + w);
}
```

1から100000以下の数について、2のべき乗を表示する。

```
for (int power = 1; power <= 100000; power *= 2 ) {
    System.out.println( "power of 2: " + power );
}
```

```
}
```

何もしないで永遠に繰り返す（A～Dまでの部分はいずれも省略可能です）。

```
for (;;) {} // プログラムが止まらなくなるので要注意、forever
```

★ループ変数のトレース

1から10までの値の総和を求めるプログラムの断片を記述してみましょう。

```
int    sum = 0;
for (int i = 1; i <= 10; i++) {
    sum = sum + i;
}
System.out.println( "total: " + sum );
```

変数の値がどのように移り変わっていくのかに注意してください。**for**文の中で宣言された変数は、繰返しを制御するためのループ変数になっています。特に、ループ変数とループ変数に関係のある変数の変遷を追っていく（トレースする）ことは重要です。上記の総和のプログラムにおいては、変数`sum`の値とループ変数 `i` の値の移り変わりに注目してみましょう。

```
sum:   0 → 1 → 3 → 6 → 10 → 15 → 21 → ....
i:      1 → 2 → 3 → 4 → 5 → 6 → 7 → ....
```

2つの代入 `sum = sum + i;` と `i++` によって、このように、値が相互に変わっていきます。`sum`は、最初が0でどんどん値が足し込まれています。これは、例えば風呂桶などに、小さな桶を何回も使って水を足していくのに似ています。



図10-1 水を足し込んでいく

★止まる繰返しを設計しよう

while文のときと同様に、いずれは停止する繰返しを記述するべきです。継続条件は、適切なものを選択しよう。また、継続条件と停止条件の区別をしましょう。**while**文も**for**文も条件として書くのは、繰返しを継続するときの条件です。

```
for ( int i=1; i >= 10; i++) { .... } // 一回も実行されない
for (int x=10; x>0; x++) { .... } // 止まらない
```

★for文で間違いやすい例

for文で、初期設定、継続条件、再設定をカンマ（,）では区切ってはいけません。また、閉じる丸括弧（）の直後にセミコロン（;）を入れると意図していない動作になります。

```
for (int i=1, i<=10, i++) { .... }           // 正解はセミコロン;

for (int i=1; i<10; i++) ; { .... }          // ここにセミコロンを書いてもコンパイラはここで
                                              // エラーを出さない（if文、while文でも起こり得る）
```

■while文とfor文との互換性

両方を書き直してみましょう。for文がwhile文に比べて強力な記述ができることがわかります。

★while文をfor文で書き直してみる

```
while ( B ) {                               for (; B;) {
      D                                     D
}                                           }
```

★for文をwhile文で書き直してみる（注1）

```
for ( A; B; C ) {                           A;
      D                                     while ( B ) {
}                                           D
                                           C;
                                           }
```

★書き直してみた例

第6章の格子枠を書くpaintメソッドをfor文で書き直してみました。かなり短い記述で済んでいます。

```
public void paint( Graphics gc ) {
    for( int y = 10; y <= 70; y += 20 ) {
        for ( int x = 10; x <= 70; x += 20 ) {
            gc.drawRect( x, y, 10, 10 );
        }
    }
}
```

第8章のカラーテーブルを書くpaintメソッドをfor文で書き直してみました。

```
public void paint( Graphics gc ) {
    for ( int blue=0; blue<256; blue += 16 ) {
        for ( int red=0; red < 256; red += 16 ) {
            Color mycolor = new Color( red, 0, blue );
            gc.setColor( mycolor );
            gc.fillRect( blue+30, red+30, 14, 14, true );
        }
    }
}
```

```

    }
}
}

```

(注1) 変数の有効範囲を考えると、左側の for 文と同じにするためには、右側の while 文ではその外側にもう一つブロックが必要になります。

10.2. グラフィックスと繰返し

グラフィックスで図形を表示させるメソッドには、drawLineの他にdrawRect、drawOvalあるいはdrawArcなど、中を塗りつぶすメソッドも含めるとかなり多数のメソッドを取り上げてきました。ここでは、その中で一番基礎的なdrawLineを使って、繰返しと組み合わせたときの効果を復習します。また、メソッド呼出しのためのパラメータを、どのように設計しなければならないかということも、drawLineを使って学んでいきましょう。

■drawLineとfor文による1次関数グラフ

for文とdrawLineを使って、様々な図が掛けることに注目しましょう。まず、drawLineをパラメータの指定の仕方を復習してみましょう。

```
drawLine( 始点のx座標 , 始点のy座標 , 終点のx座標 , 終点のy座標 );
```

縦線と横線についてパラメータがどうなっているかを考えてみると、次のようになります。

縦線	始点のx座標と終点のx座標が等しい
横線	始点のy座標と終点のy座標が等しい

たとえば、横線を使って平行四辺形を書くことを企図してみましょう。このときは、始点も終点も y座標はfor文で使うループ変数をy座標に当てはめてみます。x座標はループ変数を使って、幅を持たせてみます。

```

public void paint (Graphics gc ) {
    for (int i=1; i<=100; i++) {
        gc.drawLine( 100-i, i, 200-i, i );
    }
}

```



図10-2 横線で描画した平行四辺形

始点と終点のx座標を変えましたが、始点は固定し、終点のx座標だけ変えてみると次のような図になります。

```

public void paint (Graphics gc ) {
    for (int i=1; i<=100; i++) {

```

```

        gc.drawLine( 100, i, 100+i, i );
    }
}

```



図10-3 始点のx座標を固定した場合

同じことは縦線でy座標を変えて行くような場合にも適用できます。

■for文と2次関数を使ってグラフを描いてみる

$y = x^2$ という懐かしい響きのするグラフを書いてみましょう。まずは、横線で描くことを考えてみます。横線なので、2番目と4番目のパラメータが同じ値になります。

```

public void paint (Graphics gc ) {
    for (int i=1; i<=100; i++){
        gc.drawLine( 1, i, i*i, i );
    }
}

```

終点のx座標をループ変数で指定しましたが、二乗するとあまりにも数が大きくなるので、すぐ表示領域からx座標がはみ出してしまう。しかも終点を動かしたので白黒の関係がなんとなく反対であるようみえます。



図10-4 終点を移動させた場合

そこで、終点のx座標を変更していましたが、逆に始点のx座標を変更の対象とし、かつ二乗した数を50ぐらいで割って、横方向の大きさを圧縮してみましょう。

```

public void paint (Graphics gc ) {
    for (int i=1; i<=100; i++){
        gc.drawLine( i*i/50, i, 200, i );
    }
}

```

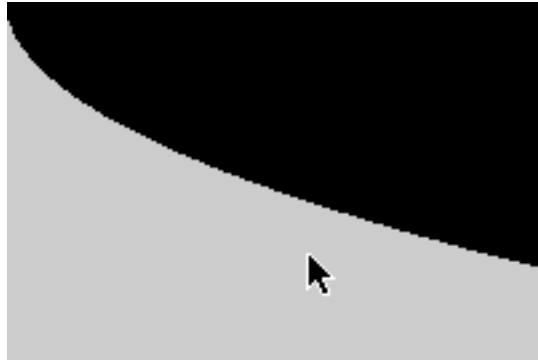


図10-5 始点を移動させ、かつ圧縮をした場合

頭を90度回転させれば、2次曲線に見えてきました。それでは、今度は縦線で濃く塗り潰すような形で書いていきましょう。まず、y方向がアプレットでは逆方向になっているので、最大値（ここでは仮に100としています）から引く形で描いてみましょう。つまり、 $y = 100 - x^2$ という式になります。今度は、縦線なので始点と終点のx座標が等しくなります。

```
public void paint (Graphics gc ) {
    for (int i=1; i<=100; i++){
        gc.drawLine(i, 0, i, 100 - i*i/50);
    }
}
```



図10-6 縦線で描いた場合

だんだん、昔教科書で見たような図形に近づいてきました。対称形にするために、x座標をマイナス50から+50までの範囲を求めてみましょう。しかし、アプレット上のx軸は0から始まります。マイナスの部分は表示されません。そこで、中心を予め右にずらしましょう。ということで、x座標にいつでも+50を足します。

加えて、上の結果を見ると、x座標が100になる前に曲線の部分が終わってしまっています。これを避けるために、-50や+50の場合についてももう少し精密に考えてみましょう。xの値が50のときは、圧縮を掛けないとすると、 $y = 100 - 50 * 50 = 100 - 2500 = -2400$ となって、途方もなくアプレットの描画範囲からはみ出してしまいます。そこで、縦方向に圧縮を掛けましょう。-50や+50のときに2乗した結果がちょうど100になるようにすれば、 $y = 100 - 100 = 0$ となって描画範囲からはみ出ません。そのための圧縮率は、 $100/2500$ すなわち、 $1/25$ ということになります。よって、最終的な2次関数は、 $y = 100 - x^2/25$ ということになります。これを終点のy座標として指定してみます。

```
public void paint( Graphics gc ) {
    for (int x = -50; x<= 50; x++){           // xを-50から+50まで1つずつ変化させる
        gc.drawLine( x+50, 0, x+50, 100 - x * x / 25 );
    }
}
```

}

上のメソッドでは、drawLineのx座標の値が始点・終点共に、 $x + 50$ という同一の値になっているので、縦線を引くことになり、かつその縦線は、y座標として、0から、2次関数のカーブを表す座標まで描かれます。これによって描かれる図は、次のような感じになります。



図10-7 最終的な2次曲線を持つ図形

drawLineメソッドを使って、繰返しを用いて図形を描くにはどのようにすれば良いかを考えてきました。このような簡単なメソッドでも、与えるパラメータの式を工夫することによって、いろいろな図形を描くことができるのです。アプレットの描画領域に収めるためには、圧縮をしたり、一定の数を足したりして、パラメータの式をうまく設計しないとはいけません。圧縮率や最大値などは、どの範囲を描きたいかによって自分で工夫してみてください。ちなみに、縦線ではなくて、始点をまったく1つの点にしてしまえば（すなわち、`gc.drawLine( 50, 0, x + 50, 100 - x * x / 25 );`にすれば）、こんな渋い図形も描けます。

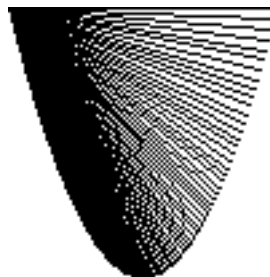


図10-8 始点を一点にした場合

10.3. 論理式と複雑な条件分岐

■条件分岐のネスト

if文の中にif文が入った構文を条件分岐のネスト（多重化：nesting）と呼んでいます。条件分岐のネストは、次のような処理を記述するときに使われます。

- ・ 複雑な条件を組み合わせるとき
- ・ 大きく処理を分岐した後、より細かな条件分岐を行いたいとき

例えば、月に関して分類してみましょう。8章で出てきた現在の時間を求めることができる Calendarクラスを使って、今日の月が変数monthに次のように求めているとします。なお、このクラスはjava.utilパッケージに収められています（使うためには、プログラムの先頭で**import java.util.*;**と記述しておく必要があります）。

```
Calendar today = new GregorianCalendar( TimeZone.getTimeZone("JST") ); // 日本時間の場合
int month = today.get( Calendar.MONTH ) + 1;
```

この場合に、小の月（月の日数が31日に満たない月）すなわち2, 4, 6, 9, 11月を求めてみましょう。まず、偶数の月か奇数の月か分けてみます。その後に、偶数の月であれば8よりも小さい月、奇数の月であれば8よりも大きい月に分類しています。

```

if ( month % 2 == 0 ) {           // 偶数の月
    if ( month < 8 ) {           System.out.println( month + "月は、小の月です。" ); }
    else {                       System.out.println( month + "月は、大の月です。" ); }
}
else {                           // 奇数の月
    if ( month > 8 ) {           System.out.println( month + "月は、小の月です。" ); }
    else {                       System.out.println( month + "月は、大の月です。" ); }
}

```

■複雑な条件式

条件分岐を組み合わせたときは、条件分岐のネストを使う方法以外に、条件自体を組み合わせた記述の仕方もあります。単純な比較式を組み合わせた条件式を論理式（Logical Expression）と呼んでいます。

▼論理式の書式

条件式 && 条件式	両方の条件が成立（論理積）	AND（かつ）
条件式 条件式	どちらかの条件が成立（論理和）	OR（または）
!(条件式)	否定	NOT（でない）

ちなみに演算子の優先順位としては、否定の!が一番高く、その次は論理積と呼ばれる&&、最後は論理和と呼ばれる||の順になっています。いくつかの例を挙げてみましょう。

```

100 <= x && x <= 200           // 100 <= x <= 200は間違い
month % 2 == 0 && month < 8     // 偶数の月の小の月の条件
x == 0 || x == 100             // 0かあるいは100
!( x == 100 )                  // 100でない
!( x == 0 || x == 100 )        // 0でも100でもない

```

これらの論理式は、if 文の条件として、あるいはwhile文やfor文の継続条件として用いることができます。論理式を使えば、さきほどのネストを使って記述した大の月、小の月の分類を行なう if文をネストを使わずに記述することができます。&&の方が||よりも優先的に評価されますので、余分な括弧をつけていません。

```

if ( month % 2 == 0 && month < 8 || month % 2 == 1 && month > 8 ) {
    System.out.println( month + "月は、小の月です。" ); }
else {
    System.out.println( month + "月は、大の月です。" ); }

```

■論理式の変換規則

条件式で使われる論理式は、同じ条件を記述するのに複数の記述の仕方ができます。ある条件式と別の条件式が同じ条件を示しているかどうかは、変換法則を適用することによって判別することができます。

★ド・モルガンの定理

ド・モルガン (de Morgan) の定理は論理式の有名な変換法則です。右と左は、等しい記述になっています。

$\neg(\text{論理式 1} \ \&\& \ \text{論理式 2})$	$\neg(\text{論理式 1}) \ \ \neg(\text{論理式 2})$
$\neg(\text{論理式 1} \ \ \text{論理式 2})$	$\neg(\text{論理式 1}) \ \&\& \ \neg(\text{論理式 2})$

たとえば次のような例を見てみましょう。左右は等しい記述です。

$\neg(x == 0 \ \ x == 100)$	$\neg(x == 0) \ \&\& \ \neg(x == 100)$ あるいは、 $x != 0 \ \&\& \ x != 100$
$\neg(100 <= x \ \&\& \ x <= 200)$	$\neg(100 <= x) \ \ \neg(x <= 200)$ あるいは $x < 100 \ \ x > 200$

★それ以外の変換法則

普通の不等号や等号に関して、論理式を使えばいろいろな形で記述することができます。簡単にまとめてみました。左右は等しい記述となっています。AとBは、それぞれ何らかの式であると思ってください。

$\neg(A == B)$	$A != B$	等号と不等号の変換
$A <= B$	$A < B \ \ A == B$	不等号の分解
$\neg(A > B)$	$A <= B$	逆の不等号への変換

■繰返しの途中脱出

繰返しにおいて、繰返しの途中まで処理を行ない、それ以降の処理を行わずに繰返しを終了させたり、次の繰返しを始めたりしたいことがあります（かも知れませんが）。そのときに、用意されているのが、それぞれ **break** 文と **continue** 文です。

★break 文

break 文は、**for** や **while** 文などの繰返しから強制的に脱出するという効果を持ちます。**break** 文は単体で現れることはなく、**if** 文とともに脱出条件を指定して、途中脱出するのに用いられます。

▼break 文の典型的な使用例：

```
while (true) {           // for (;;) と書いてもよい
    .....
    if ( 終了条件 ){ break; }
    .....
}
```

たとえば、総和が初めて100を越えるときの総和と数を表示してみましょう。次の繰返しは **while** 文を使ってみました。が、**if** 文で100を越えたときのループ変数 *i* と変数 *sum* の値を表示して、繰返しを脱出しています。

```

int sum = 0;
int i = 0;
while ( true ) {
    sum += i;
    if ( sum >= 100 ) { System.out.println( "sum: " + sum + " where i: " + i ); break; }
    i++;
}

```

break文を好んで用いるプログラマもいます。何故でしょうか？**while**文や**for**文などでは、継続条件を指定する必要があります。それは、脱出条件の反対の指定になっています。それに対して、**if**文と**break**文を用いた場合は、脱出条件を直接記述することができます。脱出条件の方がわかりやすいと考えているプログラマは、後者の方法を好むのです。次の2つの繰返しは、同じ内容を表しているのですが、どちらの方がわかりやすいでしょうか？わかりやすい書き方を用いてください。

```

for ( int power = 1; power <= 100000; power *= 2 ) {           // 100000以下ならば続ける
    System.out.println( "power of 2: " + power );
}

for ( int power = 1; ; power *= 2 ) {
    System.out.println( "power of 2: " + power );
    if ( power > 100000 ) { break; }                            // 100000より大きくなったら脱出
}

```

break文は、繰返しからネストしているときは、一番内側繰返し（ブロック）から脱出するという効果を持ちます。もしブロックを何重にもネストしているときに、その外側に脱出したいときは、ラベルを使った脱出法があります。たとえば、次の例ではラベルを予め指定してあります。ラベルは、コロン（:）を伴って、記述しておきます。**break**文の後に、そのラベル名を指定したら、そのラベル書かれているレベル（ブロック）まで脱出してくれます。

```

escapeLevel:                                                   // ラベルで指定します
for ( int x = 1; x < 10; x ++ ){
    for ( int y = 1; y < 10; y ++ ){
        System.out.print( x + " multiplies with " + y + " is " + x * y );
        if ( x * y > 50 ) {                                     // 掛け合わせた結果が50より大きければ
            break escapeLevel;                                 // ラベルで指定したレベルに脱出
        }
    }
}
// 脱出先はここになります

```

★**continue** 文

continue 文は、それ以降の処理を行わずに次の繰返しを行なわせるために用いられる制御文です。

▼**continue** 文の典型的な使用例

```

while ( 継続条件 ) {
    A
    if ( 条件式 ) { continue; }
    B
}

```

上の使用例では、**if**文の中の条件式が満足されるとBの部分を実行せずに、次の繰返しに進むことになります。条件式が満足されない場合は、Aの部分の後に、Bの部分も続けて実行されます。

たとえば、東京エリアのTVチャンネルに関して、表示することを考えてみます。2チャンネルはビデオとして、それ以外の1, 3, 4, 6, 8, 10, 12チャンネルが通常のTVチャンネルである仮定します。以下は、各チャンネルに何が割り当てられているのかを表示するようなプログラムの断片です。

```

for ( int i = 1; i <= 12; i++ ){
    System.out.print( "Channel: " + i );
    if ( i == 2 ) {      System.out.println( " Video Channel" ); continue ; }
    if ( 12 % i != 0 && i % 2 != 0 ) { System.out.println( "" ); continue ; }
    System.out.println( " TV Channel in Tokyo Area" );
}

```

上の例では、12をその数で割ったときに割り切れなくて、かつ奇数の場合には、次の繰返しに行くように記述されています。すなわち変数*i*の値が、5, 7, 9, 11のときは最後のprintlnメソッドの呼出しはスキップされます。

■return文

paintやinitメソッド、あるいは既出のactionPerformedなどのメソッドにおいて、メソッドの実行を途中で終わらせたいときは、**return**文を用いることができます。**return**文は、メソッドの定義の章でもう一度説明しますが、ここではメソッドを強制終了させるものとしてだけ覚えておいてください。

たとえば、次の例はdrawLineで線を描いているのですが、始点と終点の座標値が同じになったら、paintメソッドを終了させるようにしています。

```

public void paint( Graphics gc ) {
    int x1 = 10, y1 = 100, x2 = 300, y2 = 200;
    while ( true ) {
        if ( x1 == x2 && y1 == y2 ) { return ; }
        gc.drawLine( x1, y1, x2, y2 );
        x1 += 2; x2 -= 2;
        y1 += 1; y2 -= 1;
    }
}

```

■do～while文

do～while文は繰返しを行なう制御構文の一つですが、あまり使う機会は多くないようです。どちらかと言えば、この文を使うよりも**break**文で脱出させる方がわかりやすいでしょう。**do～while**文では、継続条件を最後にチェックします。ですから、繰返しをさせたいことは、必ず1回は実行されてしまいます。**do～while**文

は、次のような書式で書きます。特に、継続条件を書き終わった後、セミコロン (;) を書かなければいけないので、注意してください。このセミコロン（通常の **while** 文や **for** 文の後にセミコロンを書くとは誤動作をする）のこともあって、あまり使われないようです。

```
do {  
    繰り返したいこと  
} while ( 継続条件 );
```

次のプログラムの断片は、8章で出てきた60秒だけ待つ例題ですが、**do**～**while** 文を使った方がすっきりと記述することができます。このように、一度やってみて、後から繰り返しかどうかを判定したい場合は、**do**～**while** 文の方が同じことを何回も書かなくて済みます。

```
Calendar origin = new GregorianCalendar( TimeZone.getTimeZone("JST") );  
long start = origin.getTime().getTime( );  
long second = 0;  
do {  
    Calendar current = new GregorianCalendar( TimeZone.getTimeZone("JST") );  
    long now = current.getTime().getTime( );  
    if ( (now - start) / 1000 > second ) {           // 次の秒になったら  
        second = (now - start) / 1000;  
        System.out.println( second + " seconds" );  
    }  
    second = (now - start) / 1000;  
} while ( second < 60 );
```

注意しなければならないのは、最後の **while** 文の条件式で用いられる変数はブロックの外側で定義しておかなければならないことです。変数 *second* は、ブロックの手前で宣言されています。また、*second* に新たな秒が代入されているのも、次の繰り返しの直前であるということに注意してください。

■switch文

定数と比較する条件式は、**switch** 文を使うとわかりやすい記述できる場合があります。プログラマによっては、**if** ～ **else if** 文を使うよりも、こちらの方を使う場合があります。

▼switch文の書式

```
switch ( 式 ) {  
    case 定数1:  
        式の評価結果が定数1のときに実行すること  
        break;  
  
    case 定数2:  
        式の評価結果が定数2のときに実行すること  
        break;  
        :  
    default:  
        それ以外のときに実行すること  
        break;  
}
```

これは、以下の**if**～**else if** 文と等しくなっています。

```
if ( 式 == 定数1 ) {
    式の評価結果が定数1のときに実行すること }
else if ( 式 == 定数2 ) {
    式の評価結果が定数2のときに実行すること }
    :
else {
    それ以外のときに実行すること }
```

たとえば、現在の時間が次のように変数`hour`に求まっているとしましょう。

```
Calendar cal = Calendar.getInstance(); // グリニッジ標準時間になります
int hour = cal.get( Calendar.HOUR_OF_DAY );
```

このときに、8時と12時と、午後7時（19時）ならば、食事に行くような条件分岐を書いてみましょう。

```
if ( hour == 8 ) {      System.out.println( "朝食を食べましょう" ); }
else if ( hour == 12 ) { System.out.println( "ランチを食べましょう" ); }
else if ( hour == 15 ) { System.out.println( "午後の紅茶を飲みましょう" ); }
else if ( hour == 19 ) { System.out.println( "ディナーを食べましょう" ); }
else {                 System.out.println( "食べ過ぎにご用心" ); }
```

これをswitch文を使って書き直してみますと、次のようになります。それぞれの場合の最後に、`break`文を入れなければならないことに注意しましょう。`break`文がないと、次のcaseまで実行してしまいます。

```
switch ( hour ) {
case 8:      System.out.println( "朝食を食べましょう" ); break;
case 12:     System.out.println( "ランチを食べましょう" ); break;
case 15:     System.out.println( "午後の紅茶を飲みましょう" ); break;
case 19:     System.out.println( "ディナーを食べましょう" ); break;
default :    System.out.println( "食べ過ぎにご用心" ); break;
}
```

■if 式

条件文を使いますと、いろいろな場合分けができるのですが、代入文などではいちいち条件分岐を使うのが面倒な場合があります。たとえば、次のプログラムの断片を考えてみましょう。

```
if ( month < 9 ) { season = 1; } else { season = 2; }
```

これをもっと簡単に記述するための式が用意されています。これを一般に**if式**と呼んでいます。

▼if 式の書式

(条件式) ? 条件式が真のときの値 : 条件式が偽のときの値

このif式を使いますと、さきほどの条件分岐は一つの代入文で記述することができます。

```
season = ( month < 9 ) ? 1 : 2;
```

このように、if文がなくなり、簡潔に記述できるので、これを愛用するプログラマはかなり多いのですが、あまり多用しすぎると、かえってプログラムを読みにくくします。使うときには読みにくくなるかどうか、いつも注意しながら使いましょう。なお、if式もネストさせることができます。例えば、季節毎に変数の値を1～4まで変えたいときには、if式をネストさせて、次のように記述します。これは、if～else if文に相当します。

```
season = ( month <= 3 ) ? 1 :  
          ( month <= 6 ) ? 2 :  
          ( month <= 9 ) ? 3 : 4;
```

■ローカル変数の有効領域とブロック

ブロックは、波括弧 (brace) すなわち {と} で囲まれた領域のことを指しています。プログラムの実行時において、while文、for文や、if文のブロックの中で宣言した変数というのは、それ以降使えなくなりますので、注意しなければいけません。これらの制御文のブロックの中で宣言した変数は、そのブロックの中でしか有効ではないのです。このことについて、幾つかの例で考えてみましょう。.....の部分は、何がしかの記述がされていると考えてください。

```
while ( ..... ) {  
    int x = 10;  
    .....  
}
```

この記述では、変数xは繰返しのたびに毎回宣言され、初期値が10として代入されます。while文が終了した段階で、このブロックで記述された変数xは消滅します。次の例は、ブロックだけを単体で記述したものです。内側のブロックでは、変数xもyも有効ですが、外側のブロックでは変数xのみが有効です。

```
{  
    int x = 10;  
    ..... // ここでは変数xのみが有効  
    {  
        int y = 20;  
        ..... // ここでは変数xとyの両方が有効  
    }  
    ..... // ここでは変数xのみが有効  
}
```

なお、次のような記述はどうなるのでしょうか？

```
for ( int x = 10; x < 20; x++ ) {  
    while ( y < 10 ) {  
        int x = 20;
```

```

.....
    }
}

```

for文で宣言した変数*x*には、最初の値が10になっています。しかし、内側の**while**文の中のブロックでは、変数*x*は毎回20として初期化されています。この場合、一番内側のブロックの同じ名前の変数が外側のブロックの変数を見えなくしています。これを変数の隠蔽（Shadowing）と呼んでいます。Javaでは、ローカル変数ごうしの隠蔽は、コンパイラによってエラーにしています。ただし、インスタンス変数はローカル変数で隠蔽できますから注意してください。

10.4. まとめと課題

■まとめ

★for文

- ・ 4つの部分を記述する
- ・ while文などとも互換性がある
- ・ 宣言した変数はfor文のブロックの中でしか使えない

★グラフィックと繰返し

- ・ パラメータの部分にループ変数を使った式を記述する
- ・ アプレットの描画領域に収まるように圧縮したり、移動したりする必要がある

★論理式の記述の仕方

- ・ 複数の条件式を「かつ」、「または」で繋げたり、「～でない」と反対の意味にしたりできる
- ・ ド・モルガンの法則などを利用した論理式の変換

★様々な条件分岐のための制御文

- ・ **break** 文
- ・ ラベルを伴った**break** 文
- ・ **continue** 文
- ・ **return** 文
- ・ **do～while** 文
- ・ **switch** 文
- ・ **if** 式

★課題

10-1.

一次関数と二次関数について、自分でアプレットを作成し、drawLineの4つのパラメータをfor文と共に変化させて、効果を確認めなさい。drawRectや、drawOval、あるいはdrawArcなども用いて、パラメータにfor文と共に変化させるような式を記述して、その効果を確認めてみなさい。

10-2.

次のような二次関数を用いて、横線から構成される図を描画するような、アプレットを作りなさい。クラス名は、Leafにちなさい。均等に色分けされた色は次の6色を使いなさい：上から緑（Color.green）、黄（Color.yellow）、橙（Color.orange）、赤（Color.red）、シアン（Color.cyan）、青（Color.blue）。

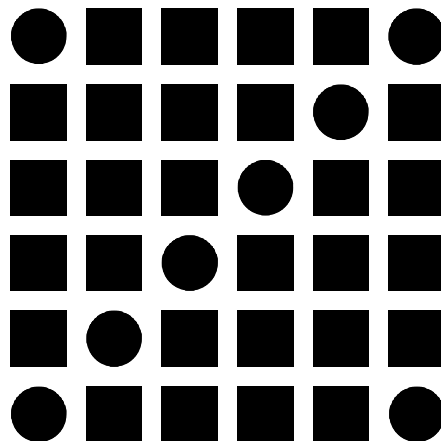


問題の図

ヒント：葉っぱ全体を包む正方形の一辺のサイズを W とすると、葉っぱの左端は、 y^2/W で、右端は $W-(W-y^2)/W$ で計算できます。横線であることに注意下さい。

10-3.

for文とif文を用いて、次のような図を描くアプレットを作成下さい。アプレットのクラス名は、OvalinRect。例えば、左上の角の座標は、(20, 20)とし、正方形の大きさは一辺15ドットで、上下左右の間隔は5ドットとします。



問題の図

10-4.

8章で出てきた60秒待つ例題を、**break**文を用いた繰返しで書き直して下さい。