Chapter 5. プログラムの状態遷移と繰返し

５－１．変数と代入

５－１－１．宣言時における初期値代入

変数を宣言する際には、同時に初期値を代入することができます。これによって、宣言と最初の代入を１つにすることができます。次のような書式で記述します。

▼宣言時の初期値代入の書式：

```
型名　変数名　＝　式；
```

例えば数値型の変数の場合は、次のように使います。

```
int     z = 10; // 整数型の変数zを宣言し、その値を10にする
double  y = z * 67.0;     // 倍精度浮動小数点実数型の変数yを宣言し、その値をz * 67.0にする
```

上記の２番目の例のように既に存在する変数を使っての初期化することも可能です。しかし、右辺の式中に出現する変数は、それまでに値が代入されていなければいけません。

同じ型の変数の宣言は、カンマ（，）で区切っていくつも書いても構いません。

```
int     x = 20, y = 30;           // 整数型の変数xとyを宣言し、その値を設定する
int     w = x * 20,  u = y / 20;  // 同様
```

なお、次のように記述した場合は、変数yは初期化されますが、変数xは初期化されないので注意してください。

```
int     x, y = 30;                // 整数型の変数xとyを宣言し、yの値だけ30に設定する
                                  // 正解は、int x=30, y=30;　と記述する
```

変数には、一度宣言してしまったら、それ以降の代入には、宣言のための型名を書く必要はありません。よくある間違いは以下のようなものです。

```
int     w = 20;           // 整数型の変数wを宣言し、その値を設定する
int     w = 40;           // 既に変数wは宣言されているのでコンパイラエラーになる。
```

５－１－２．変数の自己参照による代入

例えば、以下の記述があるとしましょう。この代入分の羅列は、変数の値を変えていますが、その変数が持っていた過去の値とは、まったく関係なしに、新しい値をどんどん代入してしまっています。

```
x = 30;
x = 20;
x = 10;
```

一方、次のように変数の過去の値を参照して、その変数の新しい値を計算することができます。

```
x = x – 10;
```

これは、今までの数学の常識から考えれば、かなりおかしな記述です。しかしながら、Javaの代入の＝というのは、数学の等価を示す＝とまったく異なる働きをするのです。＝の左辺と右辺ではまったく意味が異なります。

▼式の左辺と右辺

＝の左辺に変数が出現したら
…変数が新しく保持する値を＝の右辺の式で設定することを意味する。

＝の右辺に変数が出現したら
…変数が今まで保持していた値が参照される。

例 `i = i + 5;`
…変数*i*が保持していた値に５を足して、新しい*i*の値とします。

`x = 10 / x;`
…10を変数*x*が今まで保持していた値で割って、新しい*x*の値とします。

`y = ( 20 * y ) - z;`
…むむ、自分で考えてみなさい、、、、

５－１－３．代入の省略形

★再帰代入文と演算子

なお、同一の変数に代入する代入文（これを再帰代入文と呼びます）には、次のような省略形が用意されています。優先順位に注意して例を見てください。

+=	例：`y += 45;`	→	`y = y + 45;`
–=	例：`z –= 5 * x;`	→	`z = z – 5 * x;`
*=	例：`w *= x + 15;`	→	`w =w * ( x +15 );`
/=	例：`r /= y – 45;`	→	`r = r / ( y – 45 );`
%=	例：`p %= p/2;`	→	`p = p % (p/2);`

演算子の順番を逆に記述した場合は、=+や、=-、エラーになりませんが、予期した再帰代入の効果がありません。それは、=の後の+やーが、単なるプラスやマイナスを示す単項演算子として解釈されるからです。=*や、=/などは、コンパイラによってエラーにされてしまいます。

```
x=+6;          // これは、単に変数xに6が代入されるだけ
```

★インクリメント・デクリメント演算子

再帰代入文に関して、+=や–=といった便利な記述があると説明しましたが、変数の持っている値に１を足したり、１を引いたりするのは、更に省略形が用意されている。これらはインクリメント・デクリメント演算子（Increment/DecrementOperator）と呼ばれて重宝されています。それぞれ、++と－－で表現することが

できます。

```
y = y + 1;      →    y += 1;      →    y++;
z = z –1;  →    z –= 1;      →    z– –;
```

更に、式中ではこれらの演算子を別の演算子と組み合わせて使うことができます。

```
8 * y++              →    8 * y,  y = y + 1
z = y– – / 10;       →    z = y / 10;  y = y – 1;
```

ただし、このようにしますと式を評価しているのか、代入をやっているのか、かなりわかりにくくなります。C言語系でない他の言語のプログラマは、これを副作用（Side Effects）と呼んで、忌み嫌っていました。そこまで嫌う必要はありませんが、あまりに頻繁に使ってしまうと自分でもわからなくなってしまうので、使う際には注意した方がよいでしょう。

インクリメント・デクリメント演算子は、変数の前か後ろのどちらかにつけることができます。前につけるときと、後ろに付けるのは微妙に副作用が違うので注意が必要です。

▼副作用の違い

前につける	その変数を先に更新してから評価する
後ろにつける	その変数を評価した後に更新する

例えば、変数yが値30を持っていると仮定して、それぞれの場合で次のように変化します。

```
z = ++y * 10;    →    y = y + 1;    z = y * 10;
z = y++ * 10;    →    z = y * 10;   y = y + 1;
```

両者ともyの値は31になりますが、zの値は上の方が310、下の方が300となります。

▼インクリメント・デクリメント演算子の副作用のまとめ

++変数名	変数の値を１増やす	評価する前に更新
変数名＋＋	変数の値を１増やす	評価してから更新
－－変数名	変数の値を１減らす	評価する前に更新
変数名－－	変数の値を１減らす	評価してから更新

５－２．プログラムの状態遷移

プログラムが進行していくに連れ、変数が保持している値が変わっていくことに注意しましょう（水は絶えず流れていて、しかも元の水にあらず）。

５－２－１．プログラムの状態

プログラムが実行されてプロセスとして動いている途中で、変数が一定の値にあることを「状態（State）」と呼んでいます。この状態が、プログラムの進行にあわせて変わっていくことを「状態遷移（State Transition）」と呼びます。

```
public class Transient {
        public void main( ) {
                int x;

                x = 10;
                x = x + 11;
                x = x - 78;
                x = 32 / 4;
        }
}
```

// x の値がどんどん変わっていく

図5-1 プログラムの状態遷移

プログラムがある状態にあるということは、プログラムがもつ変数の状態が一定であるということですから、変数の値が状態遷移するということは、プログラムが状態遷移するということと等しいのです。

それでは、次のプログラムは、変数の値はどのように変わっていくでしょうか？実際に入力して、変数の値の移り変わりを確かめてみてください。

```
public class TransitionTester {
        public static void main( String a [ ] ) {
                int     x = 34;
                x ++;   x += 5;  x = x * x;   x %= 47;          x -= 23;
                System.out.println( x );          // 変数xの保持する値を表示する
        }
}
```

５－２－２．変数の宣言と有効範囲

変数は、必要になった時点で宣言(Declaration)して導入することができます。とにかく、値を代入する前には、宣言しておいてください。また、値を参照する前には変数に値を代入しておいてください。

```
int x;                          // この行以降は、xが有効
x = 34;                         // この行以降は、xを参照可能
int y;                          // この行以降は、yが有効
y = x * 12;                     // この行以降は、yを参照可能
```

前に紹介しましたように、同じ型の変数は、カンマで区切って、複数同時に宣言することができますから、プログラムの途中で、一つの宣言文で一挙に複数の変数を導入することができます。

```
int x, y, width, height;        // この行以降は、x, y, width, heightが有効
```

変数の有効範囲（Scope）は、宣言してから、その宣言文が含まれているブロックが終了するまでです。ブロックとは、波括弧（braces） {} で囲まれた範囲のことを指します。

```
{ int x;                        // xが有効になった
                                // この外側のブロックでは、xだけが有効である
        { int y;        /       / yが有効になった
                                // この内側のブロックでは xもyも有効である
        }
                                // 内側のブロックが終了するとyが消えてしまった....
}
                                // 外側のブロックが終了すると、xも消えてしまった...
```

この有効範囲の規則を違反して変数を使うと、またもやコンパイラに叱られてしまいます。例えば、次のプロ

グラムでは、メソッドの処理を定義したブロックの中で変数*independent*を定義していますが、これをブロックの外側で使うことはできません。

```
public class ProgramWithProblems {
        public static void main( String [] args ) {
                int independent;
                independent = 5;
        }
        System.out.println( independent );              // independentはここでは無効
}
```

５－２－３．変数をいくつ使うべきか？

式を用いて様々な計算を表すことができるので、実際に使うべき変数の数はそれほど多くありません。例えば、以下の左右の例は、同じ処理をしていますが、右側は一つの変数だけでパラメータを計算しています。

```
int x, y, z;
x = 34;
y = x * 43;
z = x + y;
gc.drawString( "Hello", y, z );
```

```
int x;
x = 34;
gc.drawString( "Hello", x*43, x*44 );
```

プログラムを書き慣れていない人は、いくつもいくつも余分な変数を使いがちです。必要以外に変数を用意するのは、少し考えものです。しかしながら、下の例のように、明示的に変数が保持している値の内容を区別したいときには、それぞれの情報を別々の変数に分けた保持した方がよいでしょう。

```
int x, y;                                       // xとyは、それぞれx座標、y座標を示している
x = 60;  y = 30;
gc.drawString( "Good Morning", x+20, y + 40 );
```

同一の変数を途中から別の用途に使うこともできます。しかし、変数を何の用途に使っているのかを自分で混乱しないようにしなければいけません。慣れないうちは、避けた方が無難かも知れません。

```
int     count;          // 個数を覚えておく変数を用意した
count = 10;             // まずは、Orangeが10個であることを覚えていた
:                       // その後、いろいろありまして、、、
count = 5;              // ここからは、Pineappleが5個あることを示すことにした
```

５－３．while文による繰返し

５－３－１．繰返しを記述する構文

変数の値を変えて何かを計算するということを何行も同じように記述してきました。計算したい回数分だけ、同じ内容を記述するのではプログラムが無闇に非常に大きくなってしまいます。処理する内容が一定になっている場合、それをただ単に繰り返すように記述したいものです。そのためには、上から下へとただ単に順番に実行されるだけのプログラムの実行の流れを制御する必要があります。

プログラミング言語では実行の流れを制御する文のことを制御構文（Control Flow Statement）あるいは単に制御文（Contol Statement）や構文（Structural Statement）と呼びます。

★繰返し文の種類

繰返しには３つの制御構文が使えます。最初は、簡単なwhile文を使って繰返しを記述してみましょう。２つ目のdo while文は実際のプログラムではほとんど使われませんので、本書では省略します。for文は非常に強力で、C / C++ / Java系のプログラミング言語では頻繁に用いられます。while文には、for文ほどの機能はありませんが、古くから用いられている基本的な繰返しの機能を持っています。

▼Javaで用意されている繰返し用の構文

- while文　　単純な繰返し（継続条件を先に判定）
- do while文　単純な繰返し（継続条件を後に判定）
- for 文　　複雑な繰返し

C/C++/Java系以外の言語では、これ以外にも、ある回数分だけ繰り返すこと直接記述できるような構文を持っている場合もあります。しかしJavaでは、while文やfor文と変数を用いて繰返しの回数を制御するように記述します。

５－３－２．while文を使った記述の仕方

while文は、ある条件が満たされている間、処理を繰り返すために用いられます。

▼while 文の書式

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

「繰り返したいこと」の中には、文を複数書くことができます。この部分を囲んでいる括弧（｛と｝のこと）もブロック（Block）と呼ばれています（注１）。それでは、while文の意味をもう少し詳しく、例をみながら考えていきましょう。意味はセマンティクス（Semantics）と呼ばれることがあります。

★while 文のセマンティックスと使い方

「継続条件」が満足されている間、「繰り返したいこと」に書かれている文を繰り返し実行するのがwhile文の制御の仕方です。「継続条件」は、繰返しが行なわれるたびに、その冒頭で毎回チェックされます。これだけのセマンティクスなので、繰返しはいかようにも記述できるのですが、もし３回だけ繰り返したいのであれば、次のように変数を伴って、変数の値が変化するように記述してやるのがよいでしょう。

```
int     count= 1;                   // 回数を制御するための変数を宣言し、最初は１回目とする
while ( count  <= 3 ) {   // ３回までやって
        繰り返したいこと
        count = count + 1;          // やった回数を数えるために１回繰り上げする
}
```

このようにすると、繰り返したいことを３回だけ実行してくれます。例えば、「繰り返したいこと」の部分に、System.out.println("Hello");と書きますと、Helloを３回表示してくれます。変数*count*の値は、1→2→3→4と変化していき、4になった段階で継続条件を満たさなくなり、繰返しが終了します。

実際に、次のアプリケーションプログラムを実行してみて様子を見てみましょう。次のプログラムでは端末画面にどのような表示が現れるでしょうか？

```
public class   RepeatTester  {
        public static void  main( String  [ ] a  ) {
                int       count  = 1;
                while  (  count <= 7 ) {
                        System.out.println( "いちめんのなのはな" );
                        count = count + 1;
                }
                System.out.println( "かすかなるむぎぶえ" );
                System.out.println( "いちめんのなのはな" );
                System.out.println( "作：山村暮鳥" );
        }
}
```

５－３－３．変数の値を使う

これまでの繰返しの例では、変数の値を繰返しの回数を数えるために用いました。この変数の値を「繰り返したいこと」の中で参照して利用することもできます。たとえば、単純に変数の値を表示したければ、次のようになります。

```
int       number = 1;
while  (  number <= 10 ) {
        System.out.println(  number  );              // 変数の値を表示する
        number = number + 1;                         // 表示した後に、１つ増やす
}
```

このようにすれば、変数の値が1から10まで順番に表示されます。ところで、上の例で繰り返す内容の部分を逆にして、１つ増やした後に、変数の値を表示させるようにしてみましょう。

```
int       number  = 1;
while  (  number <= 10 ) {
        number = number + 1;                         // １つ増やす
        System.out.println(  number  );              // 変数の値を表示する
}
```

こうした場合、値の表示は２から11までとなります。なぜだか考えてみてください。もう少し、実用的な例を考えてみましょう。フィートとメートルの換算を表示してみましょう。下の例を実行してみてください。

```
int       number = 1;
while  (  number <= 10 ) {
        System.out.println(  number + " feetは、" + number * 0.3048 + " meterです。" ) ;
        number = number + 1;
}
```

変数を使うのですから、別に１つずつ値を更新していかなくても構いません。次の例では、変数*y*を使って、値を20ずつ増やしていきながら繰返しを記述しています。しかも、変数の値をｙ座標の値としても利用しています。

```
int       y = 20;
while  (  y < 100   )  {
        g.drawString( "Hello",  100,  y  );  // yをパラメータの値としても用いている
        y = y + 20;
}
```

この記述の意味は、*y*の値が100よりも小さい間、*y*の値を20ずつ増やしながら、Helloという文字列を書いていくという処理になっています。この記述を用いて、繰返しの様子をもう少し詳しくみていきましょう。

★変数の値の変化

繰返しでは、注目している変数が保持する値の推移を理解することが一番重要なのです。この変数は、ループ変数（Loop variable）と呼ばれることがあります。この変数は継続条件で毎回かならずチェックされます。

*y*の値	継続条件	繰り返しの中で行なわれること
20	20 < 100	
↓		*g*.drawString("Hello", 100, 20); *y* = *y* + 20;
40	40 < 100	
↓		*g*.drawString("Hello", 100, 40); *y* = *y* + 20;
60	60 < 100	
↓		*g*.drawString("Hello", 100, 60); *y* = *y* + 20;
80	80 < 100	
↓		*g*.drawString("Hello", 100, 80); *y* = *y* + 20;
100	100 < 100	

この例では、*y*の値が20ずつ増えていきながら、最終的に値が100になったところで、100は100よりも小さくないので、継続条件が満足されなくなってしまい、終了するという形になります。繰返しは４回行なわれました。繰返しされる回数について考えてみましょう。以下のように繰返しが記述されるとしましょう。

```
int     x = 初期値;
while ( x < 最終値 ) {
        // 繰り返したいこと
        x = x + 増分値;
}
```

この場合、繰り返される回数は、（最終値ー初期値）／増分値になります。たとえば、先程の例では、(100-20)/20なので、４回と求まりました。この計算式での割り算は整数除算だと思ってください。

注１　ブロックを使わずに繰返しを記述することも可能なのですが、本書ではわかりやすくするため、および記述の間違いをなくすために、制御構文では必ずブロックを伴う形で記述します。たとえ、ブロックの中の文が一つの文だけの場合でも、ブロックとして記述します。

５－４．条件式について

５－４－１．継続条件の書き方

継続条件は、条件式（Conditional expression）を使って書くことができます。単純な条件式としては下のような比較（comparison）をする条件を記述することができます。

▼比較条件の書式

```
式　比較演算子　式
```

比較演算子（Comparison Operator）としては次のようなものがあります。

▼比較演算子の種類

演算子	意味	例
<	左辺の値が小さい	$x < y$
<=	小さいか等しい	$100 <= y / 4$
>	左辺の値が大きい	$x / 20 > 10$
>=	大きいか等しい	$y >= 100$
==	等しい	$y / 2 == x * 4$
!=	等しくない	$x != 100 - z * 10$

特に==は、代入文の=と異なり、等号を２つ記述する必要がありますので、注意してください。不等号と等号が組み合わさった演算子は、「不等号を先に書く」と覚えておくと間違えないで済みます。=<や=>といった形で記述しますと、コンパイラに叱られます。

★条件式と論理値

継続条件で使われる条件式で、条件が成立することを示すのは、論理値（Boolean Value）によって表現されます。論理値には、次の２つの値しかありません。それぞれが、成立するかどうか表現しています。

▼論理値の２つの値

true	条件が成立する（真である）
false	条件が成立しない（偽である）

条件式は評価されて、最終的に論理値になります。

例：　10 < 20　→　成立する　→　**true**
　　10 != 10　→　成立しない　→　**false**

★１つずつ減らしていく例

while文は、継続条件を毎回の繰返しの際に評価し、継続条件がtrueである間、繰返しを続けるのです。例えば、次の繰返しの記述は、変数positiveの値が正の数である場合は、*positive*>0と記述された条件式がtrueになります。最終的に、positiveの値が0になったときは、条件式がfalseになり、繰返しを終了します。

```
int     positive = 10;
while ( positive > 0 ) {
        System.out.println( "Count Down :" + positive );
        positive –= 1;
}
```

端末には、10から1までの数が表示されます。ところで、先ほどの例と同じように繰返しのブロックの中にあるSystem.out.println メソッドの呼出しと*positive*–=1; の文の順番を逆にした場合、端末にはどのような表示がされるのでしょうか？考えてみてください。

★while文はどういうときに使うか？

同じようなことを延々と記述しなければならないときにwhile文を使います。まずは、繰返しで記述できるように、変数を使って式の記述を調整します。たとえば、速度を10km/hごとに、ノット（１海里を１時間で進む速さ）に換算してみます。100km/hまで換算してみます。

```
int  speed = 10;
System.out.println( speed + " km/h is  equal to " +  speed*1.852 + "kt" );
speed += 10;
System.out.println( speed + " km/h is  equal to " +  speed*1.852 + "kt" );
speed += 10;
          :
```

```
int  speed = 10;
while ( speed <= 100 ) {
        System.out.println( speed + " km/h is  equal to " +  speed*1.852 + "kt" );
        speed += 10;
}
```

まず、繰返しなしで記述してみて、変数を使って適切な式が記述できたら、変数を使ってその部分を繰返しのブロックの中にまとめるという記述の仕方をしてみてください。

★繰返しが終わったら？

継続条件が満足せず、その評価値がfalseになって繰返しが終わったら、どうなるのでしょうか？ただ単に次の文（繰返しのブロックの後に記述されている文）をそのまま継続して実行するだけです。

５－４－２．繰返しを作るときのコツ

まず大事なのは、繰返しの中でループ変数を変化させて、継続条件がいずれは**false**になるように仕向けなければいけません。ループ変数の値は、足したり引いたりするだけではなく、掛け算などを使っても変化させることもできます。次の例は、掛け算を使っています。そして、いずれは継続条件を満たさなくなります。

```
int  x = 2;
while  ( x < 100 ) {                    // いずれは100を越えるように仕向ける
        x = x * 2 ;                     //  2 → 4→ 8→ 16→ 32→ 64→ 128と変化していく
}
```

継続条件が最終的に**false**にならない場合は、繰返しは止まらなくなります。下の例が、その見本です。

```
int x = 3;
while ( x > 0 ) {
        y = y + x;                      // 別の変数を変化させてもしょうがありません
}
```

次の例は、いつまでも継続条件を満たし続ける例です。繰返しは止まりません。

```
int x = 4;
while  ( x > 0  ) {                     // 継続条件を満たしているのですが
        x = x + 1;                      // 数はどんどん大きくなっていきます。
}
```

継続条件は、きつめにつくっておくとよいでしょう。例えば、次の記述では継続条件を x !=0 としてもいいのですが、熟練したプログラマは何かプログラムが後で拡張されることに備えて、継続条件を少しきつめに記述します。

```
int  x = 10;
while ( x > 0 ) {
        x - - ;                         //  次の例と比べてみてください
}
```

いつまででも継続してしまうような条件を書く人もいます。次の例をみてください。なぜ、延々と繰り返されてしまうのでしょうか？

```
int x = 9;
while ( x != 0 ) {                  // 0になったらやめるつもり
        x -= 2;                             // そのうち0になるでしょう。
}                                           // 終わらないぞ！なぜなのだー？
```

継続条件と終了条件を間違える人は非常に多いのです！これは、注意してください。こういう場合は、だいたいは、１回も繰返しが実行されないケースとなっています。

```
int x = 2;
while ( x > 100 ) {                 // xが100を越えたら終了したいつもり
        x = x * 2 ;                 // ところがのっけから何もせずに終了してしまう
}                                   // なぜなのだー？
```

プログラムが止まらなくなった場合は、それぞれのオペレーティングシステムやランタイムインタープリタに用意されている「強制的にアプリケーションやアプレットを終了させる」手段がありますので、それを利用して止めてください。Mac OS Xの場合は、Command + Option + Escapeでした。しかし、Xcodeからアプレットを起動した場合は、Xcodeに前面のアプリケーションを切り換えて、赤色の「停止」ボタンを押せば、強制的にアプレットビューワを終了してくれます。

５－５．課題

課題5-1
容積の度量換算です。１リットル、10リットル、100リットルと、10倍ずつ増やしていき、1000リットルまでについて、次のような単位では、どれくらいになるのか表示しなさい：石（=180.39リットル）、英ガロン（1UKgal=0.219969リットル）、米バレル（1bbl=0.00629リットル）。クラス名は、Literです。アプリケーションとして実行させなさい。

課題5-2
算数を考えてください。1 × 1 = 1でした。11 × 11 = 121ですね。やや、111 × 111 = 12321ですし、1111× 1111 = 1234321です。これは面白そうです。繰返しを使って、この後どうなっていくのか求めてください。クラス名はOnesです。アプリケーションとして実行させなさい。

ヒント：ループ変数として掛ける数を使います。最初は1にしておいて、この変数を10倍して、1を足していきます。毎回、この数とその二乗を表示させるようにしましょう。繰返しは、適当なところで終わるようにしてください。大きい数も表せるように、整数型として、**int**ではなく**long**を使いなさい。