目次

- プリミティブ型と演算子

★7　プリミティブ型と演算子 (p.76)

★7.1　プリミティブ型

プリミティブ型の一覧は p.76 参照．C 言語と異なる点がいくつかあることに注意（byte 型や boolean 型が存在すること，int 型や char 型のビット数，など）．

G09Primitive01.java の一部（println は省略）

```
 4  byte b = 100;
 5  short s = 4096;

 7  int x = 123, y = 0xf, z = 011;


10  long l  = 2147483647  + 1;
11  long l1 = 2147483647L + 1;


14  float f = 0.123f, g = 1.23e-1f;


17  double d = 234.5, e = 2.345e+2;


20  boolean b1 = true;


23  char c = 'a', c1 = '\n';
24  char c2 = 'あ', c3 = '\u266a';
```

実行結果

★7.2　演算子 (p.77)

★7.2.1　数の演算，変数への代入，キャスト

次のようなソースをコンパイルすると，コンパイルエラーとなる．

G09Primitive02.java の一部

```
3   byte b;
4   b = 100;   System.out.println(b);
5   b = b + 1; System.out.println(b);
```

コンパイル結果

```
$ javac G09Primitive02.java
G09Primitive02.java:5: 精度が落ちている可能性
検出値   : int
期待値   : byte
       b = b + 1; System.out.println(b);
                  ^
エラー 1 個
```

その理由を理解するためには，次の二つのことを知る必要がある (☆ 1)．

●演算の際に型が自動的に変換されてしまう場合がある

+,*などの二項演算において 2 数が共に byte,short,char,int 型のいずれかであった場合，int 型で演算を行い，値も int 型となる (☆ 2)．二つの数の一方が long,float,double 型のいずれかであった場合，下記に示すプリミティブ型の大小関係 (☆ 3) に従って大きい方の型に合わせて演算を行う（値もその型）(☆ 4)．

```
byte < short < int < long < float < double
```

● 代入文（ 変数 = 式 ;）において，変数の型が式の値の型のよりも「大きい」場合はよいが，変数の型が式の値の型より「小さい」場合は式の値を明示的に型変換（**キャスト**）する必要がある

☆ 1) 後述の G09TypeCast の 6 行目がコンパイルエラーとなるのも同じ理由である．

☆ 2) 上記の例の b + 1 の演算は，b の値を int 型に変換してから int 型で行われる．

☆ 3) 大きい型ほど扱える数の範囲が広い．char については p.78 参照．

☆ 4) 例： double 型+int 型の結果は double 型．

代入文において変数の型の方が大きい場合，式の値の型を変数の型に自動的に変換して代入してくれる．しかし，大小が逆の場合や自動的には変換されないような型変換をしたい場合には，次のような構文を用いて型を明示的に変換する必要がある．

(型名) 式

上記のソースの 5 行目の場合，(1)b + 1 の演算は b の値を型変換して int 型で行われ，(2) その結果式の値が int 型となるために，キャストなしで byte 型の変数に代入できない．エラーとならないようにするためには，b = (byte)(b + 1); とすればよい（b+1 の値を byte 型に変換）．

G09TypeCast.java の一部

```
 4  /*** (1) ***/
 5  int   x = 10;
 6  short y = x;  // コンパイルエラー．(short)x とすれば ok
 7  short z = 10; // 10 は int 型だけど定数なので許される
 8  System.out.println("(1) "+x+" "+y+" "+z);
 9
10  /*** (2) ***/
11  double a = 1/2;           // 1/2 の値は int 型の 0
12  double b = 1.0/2;         // double で除算するので 0.5
13  double c = (double)1/2;   // ((double)1)/2 という意味
14  double d = (double)(1/2); // int で除算してから double へ変換
15  System.out.println("(2) "+a+" "+b+" "+c+" "+d);
16
17  /*** (3) ***/
18  int p = Math.random() * 100;  // コンパイルエラー
19  int q = (int)Math.random() * 100;   // コンパイルは通るけど...
20  int r = (int)(Math.random() * 100); // r は ?? 以上 ?? 以下の整数
21  double s = Math.sin(2);  // sin() は引数に double 型の値を受け取る（p.41）.
22  // 2 は int だが定数なので double に自動変換されて sin() に渡される．
23  System.out.println("(3) "+q+" "+r+" "+s);
```

★ 7.2.2 式の評価と副作用

これまでみてきたように，**式**が表す演算を行うと結果として値が得られる．演算などを実行して式の値を求めることを，式を**評価する**と言う．例えば，1/2 という式を評価すると 0 という値が得られ (☆ 5)，Math.sqrt(100.0) という式を評価すると 10.0 という値が得られる．

式の中には，その式を評価することによって何らかの状態変化が引き起こされるようなものもあり，そのような式は**副作用**を持つという．例えば，m.moveTo(100, 0) という式を評価すると，(100,0) までの移動距離という値が得られるとともに「かめが (100,0) に移動する」という副作用が生ずる (☆ 6)．戻り値の型が void のメソッド呼び出し式（例えば m.fd(100)）は値を持たず副作用のみを持つ．一方，1/2 や 10+x といった式は副作用を持たない．

☆ 5) 1 も 2 も int 型だから，除算は int 型で行われる．1.0/2 なら値は 0.5.

☆ 6) 「副作用」という言葉から誤解しやすいが，この例からもわかるようにプログラミング言語における副作用は決して「余計な」作用ではない

副作用を起こす式が組合わさった式の場合，部分式の評価の順序が違えば異なる実行結果となる場合がある．例えば，かめ m が (0,0) にいるときに

```
m.moveTo(100, 0) + m.moveTo(100, 100)
```

という式を評価すると「m がまず (100,0) に移動し，次に (100,100) に移動する」という副作用が起こり，この式の値は 200 となる (☆ 7)．これは Java では**基本的に式は左から右へ評価される**ためである．moveTo() の順序を入れかえて

```
m.moveTo(100, 100) + m.moveTo(100, 0)
```

とすると，違う実行結果となる．この例では式の値も異なり，241.42... となる．

☆ 7) p.13 にあるように moveTo メソッドは動いた距離を値として返すので，式の値は $100 + 100 = 200$ となる．

実は，代入文によって変数の値が変化するのは，そこで用いられる代入式 (☆ 8) の副作用である．代入式は副作用だけでなく値も持っており，例えば，x = 0 という代入式は，x を 0 にするという副作用とともに，代入された値そのものすなわち 0 という値を表す．そのために，

☆ 8) 代入式： 変数 = 式

```
y = x = 0
```

という式で x も y も 0 を代入することができる (☆ 9)．if 文の条件部に次のような書き方ができるのも同様の理由による (☆ 10)．

☆ 9) よく考えると二つの代入式の評価順序がおかしいと思うかもしれないが…詳細は次節

☆ 10) この場合，代入式を囲む () は必須．その理由は次節参照．

```
if( (y = m.moveTo(0, 0)) > 100 )
```

演算子&&と||については，式の評価に関して注意が必要である．A && B（A かつ B）という式の評価では，A が false ならば B がどちらでも式の値は false となるので，この場合には B は評価されない（サボってしまう）．A || B（A または B）では，A が true の場合には B は評価されない．したがって，次の if 文は安全に（0 除算例外を発生させずに）実行できる．

```
if( x != 0 && 10/x == 0)
```

B が評価されない場合には当然その副作用も生じない．勘違いしてバグを生みやすいところである．

Q1. 次の (1),(2) のうちエラーとなるのはどちらか．その理由も答えなさい．

(1)
```
int i = 0; int[] a = {1, 2, 3};
while(i < a.length && a[i] != 0) i++;
```

(2)
```
int i = 0; int[] a = {1, 2, 3};
while(a[i] != 0 && i < a.length) i++;
```

★ 7.2.3 演算子いろいろ

Java には，+，*，!，&&，=など様々な**演算子**が存在する．そのうち，引数が二つのものを**二項演算子**，一つのものを**単項演算子**という (☆ 11)．演算子 - は，単項／二項どちらの演算子としても使われる．-2-3 という式の場合，左の-は単項，右は二項の演算子である．

☆ 11) C 言語同様に三項演算子も存在する．例えば
(x > 0) ? x : -x
という式の値は x の絶対値となる．

前節で述べたように演算子 = は変数の値を変化させるという副作用をもっているが，これ以外にも変数の値を変化させる演算子が存在する．x += 2（x = x + 2 の意味）のように使われる += や -=，*=，/= ，**インクリメント／デクリメント**（変数の値を 1 増やす／減らす）の単項演算子 ++, -- などがその例である．演算子 ++,-- は変数の前に置くか後ろに置くかで意味が違ってくるので注意が必要である（下記の例参照）．

★ 7.2.4 優先順位，結合の方向

`y = 2 + 3 * x` という式には，`=`，`+`，`*`と三つの演算子が登場するが，演算の順序を `()` で示すと `y = (2 + (3 * x))` となる．このように演算子の並び順と式の評価順に違いが生じるのは，演算子に**優先順位**があるからである．p.59 の表を見るとわかるように，これら三つの演算子は `* > + > =` という順位をもつので，その順に式の評価が行われる．

同じ優先順位の演算子が二つ以上ならんでいる `x + y - z` や `x = y += z` のようなケースでは，演算子の**結合方向**にしたがって式の評価順が決められる．通常の演算子は左に結合するが，代入演算子は右に結合する．したがって，上の例は `(x + y) - z`，`x = (y += z)` という意味になる．

定められた優先順位や結合方向と異なる評価をさせたい場合には，`y = (2 + 3) * x` のように括弧を補えばよい．優先順位や結合方向がわからない場合も括弧を用いて評価順を明示すればよい．

例

```
int p = q = 37;
System.out.println(p++);   // 出力は 37 （インクリメント前の変数値が値となる）
System.out.println(++q);   // 出力は 38 （インクリメント後の変数値が値となる）

d = m.moveTo(100,0) + m.moveTo(100,100) + m.moveTo(0,100);
                      // かめは (100,0) → (100,100) → (0,100) と動く
e = m.moveTo(100,0) + m.moveTo(100,100) * m.moveTo(0,100);
                      // moveTo() の評価順は変わらないので動き方は上と同じ（d と e の値は異なる）

a = (int)Math.PI * 10;     // 優先順位は . > キャスト > * なので，a は 30 になる
b = (int)(Math.PI * 10)    //                                  b は 31 になる

int x=1, y=2, z=3;
x += y += z;         // (1) y が 3 増える，(2) その値ぶん x も増える → y は 5，x は 6
```

★ 7.3 ラッパークラス (p.81)

プリミティブ型の値は参照型のオブジェクトとは違うものであるが，プリミティブ値をオブジェクトのように扱いたいことがある．そのために，**ラッパークラス**という，プリミティブ型に対応したクラスが用意されている．前回の `Args02` で用いた `Integer.parseInt()` というメソッドは，`int` に対応したラッパークラスである `Integer` クラスのクラスメソッドである（p.82 参照）．

第 7 章の p.83 以降の内容（列挙型，ガーベッジコレクション，クラスパス）はスキップします．また，第 8 章から第 12 章までの内容もスキップします．次回は第 13 章からです．