プログラミング言語各論
第1回講義ノート

2005年度担当: 中井央

2005年9月1日

1 はじめに

この授業ではJava言語を扱う。受講生には、プログラミング言語演習I, II程度のC言語によるプログラミングの技術を要求する。授業は演習を主体とした形式で進めていく。

2 Javaで Hello, World!

ここでは慣例 (??) にしたがって、Java で Hello, World!を表示するプログラムから入っていこう。

```
1 class Hello {
2   public static void main(String[] args){
3     System.out.println("Hello, World!");
4   }
5 }
```

図 1: Hello, World! を表示する Java プログラム (Hello.java)

Java プログラムのコンパイルは次のように行う。

```
uni% javac Hello.java
```

コンパイルに成功すると Hello.class が作成される。失敗した場合は何かエラーメッセージを表示して止まる。

```
uni% ls
Hello.class    Hello.java
```

コンパイル済みの Java プログラムを実行するには main メソッドを含むクラス名を java コマンドに続けて指定する。

```
uni% java Hello
Hello, World!
```

図 1 のプログラムを簡単に説明しよう。詳細は後ほど説明する。

1 行目： Hello というクラスの宣言。

2 行目： main 関数[1] の宣言。C 言語と同様にプログラムは main から実行される。

3 行目： 画面に表示する関数を呼び出している。

uni 上の java

uni 上の java コマンドは少し古いため、新しいバージョンのコマンドもインストールされている。これを利用するには次のようにする。

```
uni% set path=(/opt/java1.4/bin $path)
```

これを設定することで /opt/java1.4/bin に入っているコマンドをフルパス指定することなく利用できるようになる。上の設定無しでも、いちいち /opt/java1.4/bin/javac のようにフルパスで指定すれば利用可能である。なお、この設定は上の set を実行したシェルが終了されるまで有効である。すなわち、dtterm もしくは ttssh (tera term) などが終了するまでは有効であるが、新たにそれらを起動した際には set コマンドを上記のようにして実行する必要がある。

これを毎回実行するのが面倒だと言う人は、~/.cshrc というファイルに上の一行 (最後に改行コードを忘れずに) を記述しておけば、自動的に実行してくれる。このような UNIX 上のコマンド利用環境についても各自勉強しておくと今後役に立つ。

[1] C 言語では関数というが、Java ではメソッドという。

さて、新しいバージョンの java を実行する際には-pa11 というオプションを付ける必要がある。例えばコンパイルするには次のようにする。

```
uni% /opt/java1.4/bin/javac -pa11 Hello.java
```

演習 1

図 1 を入力し、コンパイル、実行せよ。

3 Java の基本型

Java には次のような基本型がある。

1. 真偽値型 boolean
2. 整数型 int, long, byte, short
3. 実数型 double, float
4. 文字型 char

以下、各々説明していく。

3.1 真偽値型

条件式を実行すると値として真か偽かのどちらかが得られる。Java では真を表すのに `true` を用いる。そして偽を表すのに `false` を用いる。

ちなみに C 言語では 0 が偽を表し、0 以外の値は真を表す。Java では真偽値が来るところに数値が来るような記述はできない。

3.2 整数型

整数を表す型は 4 つある。以下、そのそれぞれを説明する。

3.2.1 int

int は 32 ビット符号付きの 2 の補数表現の数である。値の範囲は $-2,147,483,648$〜$2,147,483,647$ である。プログラム中に現れる生の数字のことをリテラルという。int のリテラルには次の 3 つの書き方がある。

- 10 進数のリテラル (例：10, -1024)
- `0` で始まる 8 進数のリテラル (例：0777)
- `0x` で始まる 16 進数のリテラル (例：0xa5)

3.2.2 long

long は 64 ビット符号付きの 2 の補数表現の数である。値の範囲は $-9,223,372,036,854,775,808$〜$9,223,372,036,854,775,807$ である。long のリテラルを表現するには数字の列の最後に `L` もしくは `l` をつける。ただし、`l`(エル) は `1`(いち) に似ていて紛らわしいため、`L` を使うほうが良い。

3.2.3 byte

byte は 8 ビット符号付きの 2 の補数表現の数である。byte を使うのは一般的な 8 ビットの値を扱う場合、外部のファイル中の値との整合性を保つ場合、メモリを節約する場合などである。値の範囲は -128〜127 である。

byte のリテラルはない。

3.2.4 short

short は 16 ビット符号付きの 2 の補数表現の数である。主にメモリを節約する目的で使われる。値の範囲は $-32,768$〜$32,767$ である。

short のリテラルはない。

3.3 実数型

実数型には double 型と float 型がある。以下、それぞれについて説明する。

3.3.1 double

double 型は 64 ビットで保持される浮動小数点数で、IEEE(アイトリプルイーと発音する)754 標準に従って表現される。およそ $-1.7 * 10^{308}$ から $1.7 * 10^{308}$ までの数値を有効桁数 14〜15 の精度で保持できる。

double のリテラルは小数点を用いた 3.14 や 1.0 などの記述と明示的に double であることを示すために 6.02e+23d (6.02×10^{23}) などのように末尾に d を付けるかする。

3.3.2 float

float は IEEE754 標準で表現される 32 ビットの浮動小数点数である。およそ $-3.4 * 10^{38}$ から $3.4 * 10^{38}$ までの数値を有効数字 6〜7 桁の精度で保持することができる。float のリテラルであることを明示するには末尾に f もしくは F を付ける。

IEEE 754 標準について

コンピュータ上で整数を表す際、2 進数で表現するのは、電気的な内部回路で情報を保持する最小単位が、電圧の高低によるからである。例えば、電球 1 つをつけている場合は 1 、つけていなければ 0 とすると、電球を 4 つ並べれば $2^4 = 16$ 通りの電球の付け方ができるから、これによって 0 〜 15 までの整数を表現できる。電球を実際には計算機のメモリ上の 1 つの単位に当てはめれば、1 バイト＝ 8 ビットで 0 〜255 の数を扱えるわけである。

概念的には数は無限にあるわけだが、計算機には有限のメモリしかない。しかし、日常、10^{100} といった数を扱うことはないため、ある有限の桁数が扱えれば十分である。また、計算機の CPU が一度に扱えるビット数が決まっているため、数を扱う際にもそのビット数を基本と考えているのが一般的である。現時点では 32 ビット CPU が主流であり、整数も 32 ビットで扱うのを標準と考えている場合が多い。もちろん、計算機の種類、プログラミング言語の種類によって、これらは異なる。

整数には負数もあり、2 の補数表現を使うことで、負数を扱っている。

数の仕組みは n 進数の場合、1 桁は 0 〜 n までを表現し、その範囲を超える場合は通常、その左にその右側で表現できる数の最大+1 が何個あるかを記載する。いま、小数を考えなければ、n 進数は右端の桁から順に n^0, n^1, $n^2 \cdots$ の個数が記載されていく

小数は小数点の方から n^{-1}, n^{-2}, $n^{-3} \cdots$ と記載されていく。例えば、10 進数ならば、3.14 は $3 \times 10^0 + 1 \times 10^{-1} + 4 \times 10^{-2}$ である。

さて、小数を有限のメモリで表現するには、工夫が必要である。単精度浮動小数は 32 ビットで構成され、最左のビットが符合 (S) を、その次の 8 ビットが指数 (E) を、最後の 23 ビットが仮数 (F) を表す。

すると 2 進小数は次の形で書き表すことができる。

$(-1)^S \times F \times 2^E$

ここで F は 2 進数であるが表現として、その先頭の整数部分 1 桁が 1 である、$1.xxx$ の形式となるように考える。これを正規化という。例えば、0.5 は 2 進数では 0.1 であるが、正規化すると $1^{-1} = 1 \times 2^{-1}$ となる。正規化すると必ず整数部分が 1 であるので、その 1 桁は表現に含まなくてもよいことになり、1 桁得をすることになる。

E は 8 ビットであるが、指数には本来正負の数があるので、実際には $E - 127$ とみなして計算される。

さて、IEEE ではこのような小数の表現を規格化した。32 ビットによるものを単精度、64 ビットによるものを倍精度という。32 ビットの場合、上述のように符合 1 ビット、指数部 8 ビット、仮数部 23 ビットで表現される。64 ビットの場合、符合 1 ビット、指数部 11 ビット、仮数部 52 ビットで表現される。単精度では、指数部が 8 ビットであるため、正負を考慮すると -127 〜 127 の範囲を表現できる。すなわち、2^{127} 1.7×10^{38} を表現できる。仮数部は正規化されているので、実際にはその 2 倍となり、結果として、単精度で表現できる数の範囲は -3.4×10^{38} 〜 3.4×10^{38} となる。

ところで、単精度の場合、仮数部は 23 ビットであり、これは正規化による最初の 1 桁を考慮に入れて 10 進数にすると約 1.6×10^7 程度である。表現できる値の範囲に比べて、有効に数を表現できる桁数は小さい。すなわち、精度が低い。また、コンピュータ内で表現されている実数の値は、実際の値の近似である。要は 1/2, 1/4, 1/8, … の組合せで与えられた値に近い値を表現し、保持しているに過ぎない。

3.4 文字型

1 つの文字データを保持するためのデータ型が char である。Java では Unicode を採用したため、char は 16 ビットのデータ型である。

3.4.1 char

char は 16 ビットの符号なしの整数であり、文字を表現するのに用いる。値の範囲は、Unicode のコードセットの 0〜65535 である。リテラルは次のいずれか

である。

- シングルクォートで括った一文字 (例：'A')
- '\n'(改行), '\r'(キャリッジリターン), '\f'(フォームフィード), '\b'(バックスペース), '\t'(タブ), '\\'(バックスラッシュ), '\"'(ダブルクォート), '\''(シングルクォート)
- 8 進数のエスケープシーケンス。これは\nnn のような形をしている。(例：'\12')
- Unicode のエスケープシーケンス。これは\uxxxx の形をしていて、xxxx には 4 桁の 16 進数が入る。

3.5 異なるデータ型の間での演算

多くのプログラミング言語では整数型は実数型のサブタイプと考え、両方の型が混在する演算の場合、暗黙の型変換が行われている。例えば C 言語では整数どうしの割り算の結果は整数として扱うため、1/2 のような演算の結果は 0.5 にはならず、0 となる。また、整数型の変数に実数の値を代入すると小数点以下が 0 に近いほうへ丸められたりする。

このようなことを含めて異なるデータ型の間で演算を行う場合は注意が必要である。以下、Java で注意すべき点を挙げていく。

3.5.1 式の型

プログラム中で式を記述した場合、式は型を持つ。演算子に対し、被演算子がどのような型をとるかによってその (部分) 式の型が決まる。

整数の演算の場合、byte や short は int へ変換されてから演算が行われる。char の場合は先頭 16 ビットを 0 とした int 型へ変換される。long 以外の整数演算の場合は結果の型は int 型である。long が含まれている場合は結果の型は long になる。

ここで注意が必要で、byte 型の変数 b1, b2, b3 があった場合、b1 = b2+b3; はコンパイルエラーとなる。右辺の b2 および b3 は int 型にして演算が行われるため、右辺の型は int 型となり、int を byte へ代入することはできないからである。

演算に浮動小数が含まれる場合、演算は浮動小数点演算となる。被演算子として double 型が含まれていると演算は double 型で行われ、結果の型も double 型となる。

3.5.2 整数型どうしの代入

long, int, short, byte, char ではそれぞれ値を表すためのビット数が異なる。このため、代入文の右辺に記述された int 型のリテラルが代入文の左辺の変数のデータ型で扱える値の範囲を超えている場合、Java コンパイラはエラーメッセージを出力する。

次のような Java の記述がある場合、short の値の範囲を超える値 (65535) を short 型の変数に代入しようとしているため、Java コンパイラはエラーメッセージを出力する。

```
shrot s;
...
s = 65535;
```

また、a=b のような代入文の場合、左辺の変数の型のビット数が右辺の変数の型のビット数よりも小さい場合、コンパイラはエラーメッセージを出力する。このような代入が必要な場合、明示的にキャストする必要がある。キャストとは型の変換のことである。a が short 型で b が int 型の場合、a=(short)b のように記述する。

3.6 精度の問題

実数型の値が整数型へ代入される場合、小数点以下の部分は 0 に近いほうに丸められ、失われる。

int 型の値を double 型の変数へ代入することは可能である。int 型の値を float 型の変数へ代入する際には float の有効桁数から正確な値を保持できない場合がある。

また、整数どうしでキャストをし、コンパイルには通ったとしても意図した結果の値を保持していない場合もある。例えば short は 16 ビットで値を保持するため、17 ビット必要な int 型の値をたとえキャストして代入しても、正しく保持できない。

また、浮動小数を扱う際は精度を十分に保てない場合がある。

図 2 は浮動小数が実数の近似に過ぎない 1 つの例を示している。このプログラムを実行すると

```
class inexact1 {
  public static void main(String[] args){
    float pear = 0.0F;
    for (int i = 0; i<10; i++){
      pear = pear + 0.1F;
    }

    if (pear == 1.0F){
      System.out.println("pear is 1.0F");
    }
    if (pear != 1.0F){
      System.out.println("pear is NOT 1.0F");
    }
  }
}
```

図 2: 浮動小数点数は実数の近似に過ぎない例 ([5] より)

```
pear is NOT 1.0F
```

が表示される。

4 クラス

ここではひとまずクラスとは C 言語の構造体を拡張したものと捉えることにしよう。このため、クラスはメンバを持つ。Java ではメンバとは言わず、フィールド (field) と呼ぶ。C 言語からの拡張としては `struct` というキーワードを使う代わりに `class` というキーワードを使うことである。もちろん、もっともっと色々な拡張機能がある。また、クラスにはフィールドのほかにメソッド (method) も持たせることができる。メソッドとは C で言えば関数である。

C 言語では構造体に対して、`malloc` でメモリを確保し、その構造体に対する実体を何個も作ることができた。構造体を宣言するだけでは使用することはできず、`malloc` を行ってその構造体用のメモリを確保することで初めてその構造体を使ったことになる[2]。この行為に相当する Java での動作は `new` を使った実行文である。

クラス (構造体) というものをもう一度見直してみよう。クラスとは「雛型」のことである。鋳型といってもよい。金属で同じ形のものを何個も作りたい場合、「型」をあらかじめとっておき、溶けた金属をその中に流し込む。少し待つと冷えてその「形」になった金属片が得られる。金属を流し込んだ数だけ、結果の生成物が得られる。クラスも同様である。クラスを宣言するということは「型」を作ったことに相当する。別の捉え方では設計図を書いたともみなせる。どんな変数を持っているのか、どんなメソッドを持っているのかを設計図として記述することがクラスを記述することである。そして `new` を実行するということは、型に金属を流し込んで各個体を作り出していくことである。

このような作り出された**実体**をインスタンス (instance) と呼ぶ。場合によってはオブジェクトと呼ぶこともある。

[2] C 言語の場合、ポインタ変数ではない通常の変数宣言をすると、宣言と同時にメモリも確保される。

4.1 main メソッド

図 1 とその実行の様子を見て分かるようにあるクラスに `main` メソッドが含まれていれば、コマンドラインからそのクラスを指定して `java` コマンドを実行することでプログラムが開始される。プログラムの実行が `main` 関数 (メソッド) から始まると言う点では、Java も C 言語も同様である (一部、例外もある)。

クラスの例

クラスの例を示そう。いま、座標平面上の点を表すクラスを考えることにしよう。座標は (x, y) の形で表され、x や y には実際には整数値が入るとする。これはクラスではなく、構造体と捉えても良い。C 言語で構造体として表現するなら図 3 のようになる。

```
 1 #include <stdio.h>
 2
 3 struct point {
 4   int x;
 5   int y;
 6 };
 7
 8 main(){
 9   struct point p0;
10
11   p0.x = 3;
12   p0.y = 4;
13
14   printf("(x, y) = (%d, %d)\n\", p0.x, p0.y);
15  }
```

図 3: Point.c

このプログラムでは、main 関数の中で構造体型の変数を宣言して使用している。構造体型の変数は必要に応じて (p1，p2，p3, …と) 増やすことができる。

図 3 と同様のことをするプログラムを Java で記述すると図 4 のようになる。

```
 1 class Point {
 2   int x;
 3   int y;
 4 }
 5
 6 class PointTest {
 7   public static void main(String args[]){
 8     Point p0 = new Point();
 9
10     p0.x = 3;
11     p0.y = 4;
12     System.out.println("(x, y) = ("
                 + p0.x + ", " + p0.y + ")");
13   }
14 }
```

図 4: PointTest.java

図 3 と異なる点を中心に図 4 について説明しよう。まず、図 3 の 3〜6 行目と図 4 の 1〜4 行目を見ると同じに見える。同じに書かれているが、意味に微妙な違いがある。これは後ほど説明していく。

図 3 の main 関数にあたるものは図 4 の PointTest というクラスの中に (敢えて) 記述した。図 3 の 9 行目にあたるのは図 4 の 8 行目である。Java では new というキーワードを用いた文が記述されている。これに合うようもう少し忠実に図 3 を記述し直すと 9 行目は次のようになる。

```
struct point *p0 =
(struct point*)malloc(sizeof (struct point));
/* チェックは省略 */
```

すなわち、p0 はポインタ型の変数であり、new というキーワードを用いるのは malloc によって動的にメモリを割り当てるのと同じである。

図 4 の 10, 11 行目ではクラス Point のフィールド x と y に値を代入している。このことは問題であるのだが、それは次節で述べる。図 4 の 12 行目は表示用のメソッドである。C 言語の printf と似たような使い方ができる。こちらも詳細はもっとあとで述べるが、ここでは System.out.print(引数) もしくは System.out.println(引数) とすると引数に与えた数もしくは文字列が表示されると思えば良い。ln がついている方は行末に改行コードを出力する。

演習 2

図 3 および図 4 をそれぞれ、入力し、コンパイル、実行してみよ。

例をもう少し

図 4 をもとにもう少し例を示そう。図 5 は図 4 を少し改造したものである。

```
 1 class Point {
 2   int x;
 3   int y;
 4
 5   int getX(){
 6     return x;
 7   }
 8
 9   int getY(){
10     return y;
11   }
12 }
13
14 class PointTest2 {
15   public static void main(String args[]){
16     Point p0 = new Point();
17
18     p0.x = 3;
19     p0.y = 4;
20     System.out.println("(x, y) = (" + p0.getX()
                        + ", " + p0.getY() + ")");
21   }
22 }
```

図 5: PointTest2.java

クラス Point に getX と getY というメソッドを付け加えた。これらはそれぞれフィールド x および y の値を返すメソッドである。メソッド呼出しにはフィールドの参照と同様にドット (.) を使用する。

なお、ファイル名はクラス名に合わせてある。Java ではファイル名とクラス名に関連がある。これについても後ほど述べる。

演習 3

1. 図 5 を入力、コンパイル、実行してみよ。

2. 図 5 をクラス Point が、原点からの距離 (三平方の定理を使う) を返すメソッド distance を持つように変更し、コンパイル、実行せよ。main メソッドから distance を呼出し、その結果を表示すること。クラス名を PointTest3 とし、ファイル名も PointTest3.java とせよ。なお、平方根を計算するには次のメソッドを用いること。
   java.lang.Math.sqrt(引数)

4.2 クラスと関数

C 言語のような**手続き型言語**では関数を作っていくことがプログラムを作っていくことといえる。すなわち、ある問題を解くためにどのような処理 (**手続き**) を行えば良いのかを主体的に考えていくことでプログラミングが行われる。

一方、Java のようなオブジェクト指向言語では、そこにどのようなモノ (オブジェクト) が存在するかを考えて、そのオブジェクトどうしの関連 (相互作用) を記述していくことでプログラミングが行われる。ただし、実際には Java などは手続き型のプログラミング言語をベースにして考え出されており、オブジェクトが行う処理の記述は手続き型言語と同様の記述となる。オブジェクトは自身ができること (すべきこと) を知っていて、外側から「これやってよ」と指示が来るとそれを行う。このように指示を送ることをメッセージパッシングという。実際には、あるオブジェクトの持つメソッドを利用する (呼び出す) ことがメッセージパッシングであると捉えてもよい。

なお、ここでは「関数」という言葉を用いた。これは C 言語における慣習にならったためであるが、一般的には引数を受けとり、内部で計算をした結果が直接外部に影響を与えない (副作用がない) ものを関数と呼ぶ。また、関数は結果の値を必ず呼出し元へ返す。値を返さない、C 言語でいうところの返り値型が void であるような「関数」のことを一般には「手続き」と呼ぶ。

4.3 フィールド

フィールドについてもう少し説明をしよう。

まず情報隠蔽 (information hiding) という言葉から説明しよう。例えば、C 言語でプログラムを書いているとき、各関数からアクセス可能な**大域変数**(global variable) は便利であるが、危険でもある。プログラム規模が大きくなればそれだけ、どこでその変数を参照しているのか、変更しているのかということが掴みにくくなるからである。大域変数に対してある特定の部分でのみ有効な変数を局所変数 (local variable) という。

クラスのフィールドについてもどこからでも参照したり、変更したりできるとプログラム作成上問題が出る場合がある。このため、フィールドをクラスの外から直接アクセスできないようにするためのキーワード private が用意されている。この時点では皆さんはクラスの概念がまだわかっていないかもしれないが、要はクラスの外部から誰にでもアクセスできるのは危険を伴うため、外部から直接参照する必要がない変数などは外部から見えないようにしよう、というのが情報隠蔽である。

```
 1 class AnInfo {
 2   /* private */   int a = 0;
 3
 4   public void print(){
 5     System.out.println("a = "+a);
 6   }
 7 }
 8
 9 class TestIH {
10
11   public static void main(String[] args){
12     AnInfo x = new AnInfo();
13
14     x.print();
15     x.a = 10;
16     x.print();
17   }
18 }
```

図 6: 情報隠蔽のテスト用プログラム

図 6 は情報隠蔽のテスト用プログラムである。まず、このプログラムをそのままコンパイルし、実行すると次のような結果が得られる。

```
[nakai@uni]% java TestIH
a = 0
a = 10
```

この挙動自体はみなさんが期待したとおりであろう。次に 2 行目のコメントをはずし、private を有効にし、コンパイルしようとすると次のようになる。

```
[nakai@uni]% javac TestIH.java
TestIH.java:15: a has private access in AnInfo
    x.a = 10;
     ^
1 error
```

すなわち、private がついているため、変数 a はクラス AnInfo の外部からは変更 (アクセス) できない。

このようなアクセスの制御が必要な理由を次のような例で説明しよう。いま、なんらかのプログラムを作成するにあたって、あるクラスに数学の試験の得点のフィールドを設けるとする。試験は 100 点満点で行われたものとする。このフィールドをクラス Test の中に int mathtest; で宣言しているとしよう。

どこからでもアクセス可能な場合、どこかで誤って ....mathtest = 250; のように記述されていた場合、結果に誤りが生じる。

しかし、実際には外部でそのフィールドの値を書き換えたり、そのフィールドの値を取り出したりする必要は十分生じる。このときは、そのフィールドに対するアクセスメソッドを用意する。

例えば、次のようになる。

```
...
public void set_mathtest(int point){
  if (0 <= point && point <= 100){
    mathtest = point;
  }
  else {
    /* エラー処理 */
  }
}

public int get_mathtest(){
  return mathtest;
}
```

ここでは set_mathtest と get_mathtest がアクセスメソッドである。アクセスメソッドのことをアクセッサ (accessor) ということもある。

この例の中には public というキーワードも見られる。これは、指定した名前を誰からもアクセスできるようにするという意味を持つ。しかし、図 6 では public がついていない変数 a も外部から変更することができた。ここでは取り急ぎ private の概念を説明したが public など、アクセス制御についての詳細は継承などの他の要素を述べてから再び詳しく述べることにする。

演習 4

1. 図 6 を入力、コンパイル、実行せよ。
2. 図 6 のコメントを外し、private を有効にし、コンパイルしてみよ。
3. private を public にし、コンパイルしてみよ。コンパイルに成功した場合は実行せよ。
4. private の機能について考察せよ。

4.4 カプセル化

カプセル化 (encapsulation) について簡単に述べる。

オブジェクト指向では対象 (object) を何か抽象化 (abstract) して捉え、対象となるモノどうしの関係を記述することでプログラムを作成する。というととても難しそうに聞こえるが、構造体と同様に関連するものはまとめてしまい、情報隠蔽をすることで、1 つの単位として外部から守るようにするわけである。

一般的に我々がプログラムを作成する際、データとそのデータを操作する関数があることに気づく。

例えば、プログラムでスタックを実現することを考えてみよう。スタックには、データをその一番最後に入れる (push)、データを一番最後から取り出す (pop) という作業がある。データをしまうためのデータ構造としては色々考えられるが例えば配列を使ったとしよう。最後に入れられたデータが入っている要素を指すインデックス変数が設けられるはずである。

ところで、もしこのインデックス変数を勝手に変更したり、インデックスを無視して、配列の適当な要素を変更するようなことができたらどうであろうか。もちろん、それが便利に感じるときもあるが、その融通が利く点はプログラム規模が大きくなり、個人の脳みそでどの変数がどのように使われているかを把握できる範疇を超えた場合、悲劇を生む元になる可能性もある。

すなわち、カプセル化とは、プログラマが考えているデータなどの単位をカプセルでくるんだように捉え、その操作と一まとめにすることである。従来の C 言語などのプログラミング言語では、このことを言語としてはサポートしていなかった (プログラムの書き方を工夫すればカプセル化されているように記述することはできる)。

ここで図 7 にサンプルプログラムを示そう。

図 7 はスタックを表すクラスのプログラムである。

```
 1  class Stack {
 2    private int a[] = new int[10];
 3    private int i = 0;
 4
 5    int push(int x){
 6      if (i>9){
 7        System.out.println("stack over flow!");
 8        System.exit(0);
 9      }
10      a[i++] = x;
11      return x;
12    }
13
14    int pop(){
15      if (i==0){
16        System.out.println("stack under flow!");
17        System.out.println("stack is empty!");
18        System.exit(0);
19      }
20      return a[--i];
21    }
```

```
22
23    boolean isEmpty(){
24      return i==0;
25    }
26  }
27
28  class StackTest {
29    public static void main(String args[]){
30      Stack s = new Stack();
31
32      s.push(3);
33      s.push(1);
34      s.push(4);
35      s.push(1);
36      s.push(5);
37
38      for(int i=0; i<6; i++){
39        System.out.println(s.pop());
40      }
41    }
42  }
```

図 7: StackTest.java

ここではサンプルのため、手を抜いている部分がいくつかある。このプログラム例で着目して欲しいのは、private を利用してスタックの内部でだけ必要となる変数を外部から参照できなくしている点である。また、スタックという「もの」(object) をクラスとして表現して、main メソッドではそれを利用していることにも着目して欲しい。

この段階ではクラス、カプセル化、情報隠蔽について、ぼんやりわかったというところだろうか。理解するには沢山プログラムの例を知り、沢山プログラムを書いてみることが重要である。

演習 5

1. 図 7 を入力し、コンパイル、実行せよ。
2. 図 7 を改造し、Stack クラスに print メソッドを導入せよ。print メソッドは引数には何も取らず、現在のスタック内の要素を順に出力する。返り値もない。main メソッドから使ってみること。
3. 図 7 の main メソッドを変更し、stack over flow!が表示されるようにせよ。
4. 前問と同様に今度は stack under flow!が表示されるようにせよ。
5. 同様に isEmpty メソッドを利用するように変更し、実行してみよ。

4.5 抽象化

この節ではとりあえず、クラスの特徴のうち、情報隠蔽とカプセル化について述べた。クラスにはフィールドとメソッドを定義することができる。フィールドは宣言する際、private を付けることでクラス外からアクセスできないようにできる。

カプセル化は抽象化 (abstraction) を行うために必要である。抽象化とはものごとを扱う際、扱う対象の共通のことがらを調べてまとめることである。また、計算機科学においては、計算機で扱うのに必要な要素を抜き出し、不要な要素を取り除く意味もある。

オブジェクト指向のプログラミングでは、対象となる問題に現れるオブジェクト (モノ) を表現するために抽象化が必要となる。

例えば、「人」をモノと捉えて扱うことにしたとしても、その文脈によって「人」の表し方はさまざまである。単純な電話帳プログラムを考えたとき「人」は名前と電話番号から構成される。ゲームでは歩いたり、戦ったりという機能を持つかもしれない。

4.6 コンストラクタ

コンストラクタはクラスの初期化関数である。コンストラクタはクラス名と同じメソッド名を持つ。クラ

スを実体化する (instanciate) 際に使われる。コンストラクタに返り値はない。コンストラクタを定義する際、引数は任意に設定することができる。

また、オーバーロードの機能 (次節) を使用して、引数を変えた複数のコンストラクタを定義できる。

クラスの実体化とはここでは new を使ってメモリ確保することだと思えば良い。その際にコンストラクタは起動される。

演習 6

1. 人を表すクラス Person とそのテスト用の main メソッドを含む PersonTest クラスを作成してみよ。フィールドとして適当なものを持たせ、そのアクセッサを作成し、それらを表示するようにせよ。なお、main メソッドでは Person クラスの複数のインスタンスを生成するようにせよ。

```
 1 class Overload {
 2
 3   static void mymethod(){
 4     System.out.println("no args");
 5   }
 6
 7   static int mymethod(int a){
 8     System.out.println("one arg : int "+ a);
 9     return 0;
10   }
11
12   static void mymethod(String a){
13     System.out.println("one arg : String "+ a);
14   }
15
16   public static void main(String[] args){
17     mymethod();
18     mymethod(10);
19     mymethod("Hello!");
20   }
21 }
```

図 8: Overload.java

4.7 オーバーロード

メソッドのオーバーロード (overload) とは引数の違う同じ名前のメソッドを複数定義することである。

図 8 に例を示そう。このプログラムでは mymethod を 3 つ定義している 3〜5 行目では引数なし、返り値なしのものを、7〜9 行目では int 型の引数を 1 つとり、返り値が int 型であるものを、そして 12〜14 行目では String 型の引数を 1 つとり、返り値がないものを定義している。

なぜこのようなものがあるのか考えてみよう。

ここで算術演算の+を考えてみよう。+演算子は 2 つの引数をとる関数と考えることもできる。つまり、+(a, b) のように考えることができる。一般に a や b には整数型かもしくは実数型が来る。a と b が共に整数なら結果も整数である。どちらか一方もしくは両方が実数なら結果は実数となる。両者の演算を記述する場合に区別する必要がないことはプログラム記述者にとってありがたい。

なお、整数型と実数型での演算の場合、整数型を実数型に変換してから演算が行われる。この型変換を型強制 (coercion) という。

さて、同様の概念の処理だが受け取るものによって返すものを変えたいような場合、メソッド名は同一にしておきたい。例えば、C 言語では 1+2.3 と記述しても内部では 1.0+2.3 のように扱ってくれる。しかし、+や*演算子を上記のように関数だと見立てると 1+2.3 は+(1,2.3) となる。+関数は 2 つの引数を取り、1 つめは int 型、もう 1 つは double 型である。このように考えると実数と整数のどの組合せでもよいようにするためには内部で+関数を 4 つ用意しておく必要がある。

図 9 にコンストラクタとそのオーバーロードのサンプルプログラムを示す。

演習 7

1. 図 9 を入力し、コンパイル、実行せよ。
2. 図 9 を改造し、引数として Point 型の値を 1 つだけ取るクラス Point のコンストラクタを追加し、main メソッドの中で p2 として引数 1 つのコンストラクタを使用したインスタンスを作成し、35 行目などと同様にして、値を表示させよ。コンストラクタの内部ではアクセッサを用いて与えられたインスタンスの x と y の値を取り出し、自身の x と y にそれぞれセットする。

```
 1 class Point {
 2   private int x;
 3   private int y;
 4
 5   Point(){
 6     x = 0;
 7     y = 0;
 8   }
 9
10   Point(int a, int b){
11     x = a;
12     y = b;
13   }
14
15   int getX(){
16     return x;
17   }
18
19   int getY(){
20     return y;
21   }
22
23   double distance(){
24     return java.lang.Math.sqrt(x*x+y*y);
25   }
26 }
27
28 class PointTest4 {
29   public static void main(String args[]){
30     Point p0 = new Point();
31     Point p1 = new Point(3, 4);
32
33     System.out.println("p0 = (" + p0.getX() +
34                        ", " + p0.getY() + ")");
35     System.out.println("p1 = (" + p1.getX() +
36                        ", " + p1.getY() + ")");
37   }
38 }
```

図 9: PointTest4.java

4.8 final 修飾子 (1)

キーワード final についてクラス内で宣言した変数に使う場合について説明する。メソッドに使う場合は別途説明をする。

4.8.1 変数に対して使用する場合

キーワード final を変数に付けて宣言するとその変数への代入は 1 回だけに限られる。このため例えば図 10 をコンパイルすると次のようなエラーメッセージが出力される。

```
Final.java:5: cannot assign a value to final
variable pi
    pi = 3.0;
    ^
1 error
```

```
 1 class Final {
 2   public static void main(String[] args){
 3     final double pi = 3.1415926535;
 4
 5     pi = 3.0;
 6   }
 7 }
```

図 10: Final.java

ところで、クラス型の変数 (参照変数) の場合、その変数が指すものは変更できないが、その変数が指す先は変更してもよい。この例を図 11 と図 12 に示す。図 11 では t は 11 行目で一度 new によりオブジェクトを指すようになった。13, 14 行目ではその t が指すオブジェクトの内容を書き換えている。すなわち、t 自体の値 (これは C 言語風に考えればオブジェクトへのポインタ) は変更されていない。一方、図 12 では一度 11 行目でオブジェクトを指しているが、13 行目で t が別のオブジェクトを指すように指示している。しかし t は final 修飾子をつけて宣言されているため、t 自体への (アドレスの) 代入は 1 回のみ許される。このため、コンパイルすると次のようなエラーメッセージが表示される。

```
Final3.java:13: cannot assign a value to final
variable t
    t = new Test();
    ^
1 error
```

final 修飾子がつけられた変数は必ず 1 回だけ値を代入されなければならない。final 修飾子をつけて変数だけ宣言しておき、あとで 1 回だけ値を代入することも可能である。このとき、final 宣言された変数をコンストラクタ内で初期化する必要があるなら、すべてのコンストラクタ内で初期化されるように記述する必要がある。図 13 にその例を示す。コンパイルすると次のようになる。

```
 1 class Test {
 2   public int x = 0;
 3
 4   public void print(){
 5     System.out.println(x);
 6   }
 7 }
 8
 9 class Final2 {
10   public static void main(String[] args){
11     final Test t = new Test();
12
13     t.x = 10;
14     t.print();
15   }
16 }
```

図 11: Final2.java

```
 1 class Test {
 2   public int x = 0;
 3
 4   public void print(){
 5     System.out.println(x);
 6   }
 7 }
 8
 9 class Final3 {
10   public static void main(String[] args){
11     final Test t = new Test();
12
13     t = new Test();
14
15     t.x = 10;
16     t.print();
17   }
18 }
```

図 12: Final3.java

```
Final4.java:4: variable x might not have been
initialized
  Test(){
       ^
1 error
```

演習 8

1. 図 10 を入力し、コンパイルしてみよ。
2. 図 11 を入力し、コンパイル、実行してみよ。
3. 図 12 を入力し、コンパイルしてみよ。

```
 1 class Test {
 2   final int x;
 3
 4   Test(){
 5     System.out.println("nothing to do");
 6   }
 7
 8   Test(int x){
 9     this.x = x;
10   }
11 }
12
13 class Final4 {
14   public static void main(String[] args){
15     Test t1 = new Test();
16     Test t2 = new Test(100);
17   }
18 }
```

図 13: Final4.java

4. 図 13 を入力し、コンパイルしてみよ。図 13 でコンパイル時のエラーにならないようにするにはどうしたらよいか。

4.9 継承

継承とは一般には「受け継ぐ」ことを意味する。オブジェクト指向にはあるクラスの持つ機能を受け継ぐ、継承という機能がある。

この節では継承について述べる。

4.9.1 サブクラス

まず、図 14 のようなクラス A があるとする。

一応、このプログラムで何をしているのか説明しておく。

1〜5: クラス A の宣言

2〜4: このクラスのメソッド print を宣言している。void 型を返す、つまり、何も返さないメソッドであり、文字列 class A: を表示する。

7〜14: クラス A の実体 (オブジェクト) を作成し、それを使うための main メソッドを含んだクラスの宣言。

8〜13: main メソッドの宣言。クラス A の変数 a を宣言し、new を用いてインスタンス (オブジェク

```
 1 class A {
 2   public void print(){
 3     System.out.println("class A");
 4   }
 5 }
 6
 7 class Inherit {
 8   public static void main(String[] args){
 9
10     A a = new A();
11
12     a.print();
13   }
14 }
```

図 14: Inherit.java

ト) を作成している。そして、オブジェクト a のメソッド print を呼び出している。

実行結果は次のようになる。

```
uni% javac Inherit.java
uni% java Inherit
class A
```

ここまでは予想通りであろう。

ここで継承という機能を用いてクラス B を作成してみよう。図 15 のようになる。

```
 1 class A {
 2   public void print(){
 3     System.out.println("class A");
 4   }
 5 }
 6
 7 class B extends A {
 8 }
 9
10 class Inherit2 {
11   public static void main(String[] args){
12
13     A a = new A();
14     B b = new B();
15
16     a.print();
17     b.print();
18   }
19 }
```

図 15: Inherit2.java

クラス A は図 14 と変わらない。クラス A を継承してクラス B を作成した。図 15 の 7, 8 行目がそれにあたる。ここでは継承するとどうなるかを見るため、あえてクラス B の中身は定義していない。

図 15 の 10〜18 行目が継承を確認するための main メソッドである。簡単に説明すると次のようになる。

13: クラス A の変数宣言と new による実体の作成

14: クラス B の変数宣言と new による実体の作成

16: クラス A のオブジェクトに対し、print メソッドを呼び出す

17: クラス B のオブジェクトに対し、print メソッドを呼び出す

ここで疑問が浮かぶ人もいると思うが、まず、実行結果を示そう。

```
uni% java Inherit2
class A
class A
```

最初の class A という表示は a.print() によるものであり、2 番目のものは b.print() によるものである。クラス B はクラス A の機能を継承したので、クラス B もメソッド print を持っている。

さらに Inherit2.java に手を加えたものを図 16 に示す。

こちらも簡単に説明しておこう。

8〜10: Inherit2.java と違い、ここでは print メソッドを定義している。

13〜22: main メソッドを含むクラス Inherit3 を定義している。ここではクラス A, B それぞれの変数を宣言し、その実体を得ている。そしてそれぞれに対し、print メソッドの呼び出しを行っている。

その実行結果は次のようになる。

```
uni% javac Inherit3.java
uni% java Inherit3
class A
class B
```

```
 1 class A {
 2   public void print(){
 3     System.out.println("class A");
 4   }
 5 }
 6
 7 class B extends A {
 8   public void print(){
 9     System.out.println("class B");
10   }
11 }
12
13 class Inherit3 {
14   public static void main(String[] args){
15
16     A a = new A();
17     B b = new B();
18
19     a.print();
20     b.print();
21   }
22 }
```

図 16: Inherit3.java

今度は class B という表示もされている。最初のものはもちろんクラス A の print メソッドが呼ばれたものであるし、2 番目のものはクラス B の print メソッドが呼ばれたものである。

結果を見てわかるように、Inherit3.java ではせっかくクラス A から print メソッドを継承していたのにこれを上書きしてしまった。上書きすることをオーバーライド (override) と呼ぶ。この機能がどのようなところで有効なのかは後ほど述べる。

継承しても、メソッドについては同じ名前で自分用の振る舞いができるように設定できる、つまり上書きできる、ということがオブジェクト指向でのありがたみである。

さて、さらに Inherit3.java に手を加えたものを見てもらおう。図 17 にそれを示す。

今回の変更点は 19 行目と 23 行目である。19 行目ではクラス A の変数を宣言し、そこに new によってクラス B の実体を入れている。これまでの C 言語の知識からすれば、このような代入はエラーとなるはずである。

さて、先に実行結果を見ていただこう。

```
 1 class A {
 2   public void print(){
 3     System.out.println("class A");
 4   }
 5 }
 6
 7 class B extends A {
 8   public void print(){
 9     System.out.println("class B");
10   }
11 }
12
13 class Inherit4 {
14   public static void main(String[] args){
15
16     A a = new A();
17     B b = new B();
18
19     A ab = new B();
20
21     a.print();
22     b.print();
23     ab.print();
24   }
25 }
```

図 17: Inherit4.java

```
uni% javac Inherit4.java
uni% java Inherit4
class A
class B
class B
```

コンパイルに通るし、実行もできている。3 行あるうちの最後の出力結果に注目してほしい。これは 23 行目の print の呼び出しによるものであるが、ab はクラス A の変数であるにもかかわらず、クラス B の print が呼ばれている。

これは次の考えに基づいている。

クラス A を継承したということはクラス B はクラス A でもあるわけである。

オブジェクト指向の解説書などではよく生物の分類を題材に継承を説明していたりする。例えば脊椎動物というカテゴリの中に哺乳類があったりする。このほか爬虫類や両生類、魚類なども存在する。これをクラス A, B の関係に当てはめると脊椎動物が A に相当し、哺乳類が B に相当する。哺乳類を脊椎動物だと言っても間違いではない。

すなわち、クラス A の変数でクラス B の実体を指

すというのは、「クラス B は A なのだ」という考え方に基づいている。これはオブジェクト指向の考え方の中では非常に重要である。

演習 9

1. 図 14〜図 17 までをそれぞれ入力し、コンパイル、実行してみよ。
2. 図 14〜図 17 のそれぞれに対し、なぜそうなるのか、考えて、適当に手を加え、コンパイル、実行し、考察せよ。

4.9.2 スーパークラスとサブクラス

上記のようにあるクラス A を継承してクラス B を定義した場合、クラス B から見てクラス A をスーパークラス (superclass) といい、逆にクラス A から見たクラス B をサブクラス (subclass) という。

あるクラスのサブクラスを作るには上述のようにキーワード `extends` を使用する。Java ではある 1 つのクラスのサブクラスを記述することはできるが、複数のクラスをスーパークラスとする (サブ) クラスを記述することはできない。このことを単一継承という。

Java では「継承する」という言葉の代わりに「拡張」という言葉も使われる。

いま、クラス B がクラス A を継承しているとしよう。これはクラス B がクラス A を取り込んでいるとイメージしても良い。オブジェクト指向でのプログラミングはオブジェクトに対して「これやって」とメッセージを送る (頼む) ことで処理を進めていく。具体的には Java ではクラス B のインスタンスを `b` という変数に保持している場合、`b.hoge()` のようなメソッド呼出しを行う。このときまず、クラス B として `hoge` が定義されているか確認し、定義されていればその `hoge` を使う。もし定義されていなければ内部に持っているクラス A の部分に `hoge` が定義されていないか調べ、もしあればそれを使う。なお、継承は何段階も重ねても良いので、その場合は定義されているスーパークラスが見つかるまで一段ずつ遡って探していくことになる。

さて、継承したのは実はクラス A で定義しているメソッド `hoge` を飾りたてたかったのだとしよう。本当は `hoge` でも十分なんだけど、`hoge` を実行する前に`!!!` を、`hoge` を実行した後に`!!!`を表示したいとする。このときのクラス B での `hoge` は次のように定義する。

```
void hoge(){
  System.out.println("!!!");
  super.hoge();
  System.out.println("!!!");
}
```

もし、`super` をつけずに単に `hoge` と書くと自身を再帰的に呼び出すことになる。`super` をつけることで、上の言い方でいえば、内部に持っているクラス A の部分に対して `hoge` を呼んだことになる。スーパークラスのコンストラクタを利用したい場合には、`super()` というメソッドを利用する。

また、「このクラス (インスタンス) 自身の」ということを明示したい場合があり、`this` によってそれを表す。よくあるのは次のような例である。

```
class A {
  int x, y;

  A(x, y){
    this.x = x;
    this.y = y;
  }
}
```

このプログラムでは引数で受け取った値を自身がもつフィールドへ格納するコンストラクタを持っている。実際のプログラムでは変数にはそれが何を示すか分かり易い名前をつける。それはメソッドの引数の場合でも同じである。しかし、クラスの定義の場合、その値を保持するためのフィールドとメソッドの引数にできれば同じ名前を使いたい。というより、わざわざ使い分ける (名前を考えてつける) ということを避けたい、こともある。上記はこのような一例である。もちろん、`this` を使用する目的はこれだけではない。

4.9.3 なぜ継承が必要なのか

オブジェクト指向のプログラムを作る際の考え方の 1 つに差分プログラムというものがある。プログラミングの経験が多くなればなるほど、ほとんど同じなのに一部だけ違うというプログラムを書くはめになるという経験も多くなる。継承を使うことで違う部分 (差分) だけをプログラムとして記述すればよくなる。

ファイルをコピーして、ファイル名を変えて、違う部分だけを書き直すということも考えられるが、書き換え時にバグが入る可能性や他との整合性が取れなく

なる可能性もあり、ソフトウェア開発では危険な行為として避けるべきである。

差分として考えられるものにはいくつかある。あるクラス A が 10 のメソッドを持っているとしよう。A の 9 つのメソッドは同じで良いけど、1 つだけは違う動きにしたい場合、11 個目のメソッドを追加したい場合などである。

前者は図 16 のようにオーバーライドを利用することで、書き換えが必要なメソッドのみ変更すれば良い。後者の場合は継承を利用し、追加したいメソッドだけ記述すれば良い。どちらも元のクラスのソースファイルには変更を加えなくて良い。

差分プログラムという考え方の他に継承を用いるもう 1 つの理由は以下で述べる多態性によるものである。

4.9.4 多態性

次に多態性 (polymophism) という言葉について説明しよう。先ほどクラス A を拡張してクラス B を定義した。図 17 ではクラス B のインスタンスをクラス A の変数に指させた (23 行目)。クラス B のインスタンスはクラス A であるともみなせることを述べたが、そうするとクラス B の実体はクラス A とクラス B の顔を持つと言える。すなわち、複数の側面を持つのでこれを多態という。

多態性の利点は、各クラスのもととなっているスーパークラス型として各インスタンスを扱うことが可能であることである。これまでの例に対し、クラス A を拡張してさらにクラス C, D, E を作ったとしよう。それらのインスタンスはクラス B と同様にクラス A の変数で指し示すことができる。さらにクラス B, C, D, E からサブクラスを作った場合、それらのインスタンスもクラス A 型の変数で指し示すことができる。

多態性を用いること＝オブジェクト指向プログラミングとも言える。どのような場面で多態性を使うのかということについては後に例を与えるが、現時点では図 17 の `ab.print` の動作を理解して欲しい。

最後にもう 1 つ例を示そう。図 18 はクラス A とそれを継承した 4 つのクラスが宣言されている。(`main` メソッド中の配列の使い方はまだ述べていないが) クラス A 型の配列を用意し、実際には**クラス A でもある**各派生型[3] のインスタンスをその要素として代入している。その後の `for` 文において各インスタンスをクラス A として扱っているが、実行してみれば、それぞれが持つ独自の `print` メソッドが呼び出されていることが分かる。

```
 1 class A {
 2   void print(){
 3     System.out.println(‘‘A’’);
 4   }
 5 }
 6
 7 class B extends A {
 8   void print(){
 9     System.out.println(‘‘B’’);
10   }
11 }
12
13 class C extends A {
14   void print(){
15     System.out.println(‘‘C’’);
16   }
17 }
18
19 class D extends A {
20   void print(){
21     System.out.println(‘‘D’’);
22   }
23 }
24
25 class E extends A {
26   void print(){
27     System.out.println(‘‘E’’);
28   }
29 }
30
31 class Inherit5 {
32   public static void main(String args[]){
33     A a[] = new A[5];
34     int i;
35
36     a[0] = new A();
37     a[1] = new B();
38     a[2] = new C();
39     a[3] = new D();
40     a[4] = new E();
41
42     for(i=0; i<5; i++){
43       a[i].print();
44     }
45   }
46 }
```

図 18: Inherit5.java

実行結果は次の通りである。

[3] ここでは継承と同じ意味で「派生」という言葉を使っている。言葉などの「派生」と同様で、元は 1 つで、それから亜種ができることを派生という。すなわち、あるクラス A を継承した別々のクラス B, C があれば、それらを派生クラスと呼ぶ。この場合、子クラスのみでなく、それらの子孫クラスも含める。

```
uni% java Inherit5
A
B
C
D
E
```

このように一様に扱えることがオブジェクト指向でのプログラミングの 1 つの特徴である。

演習 10

1. 図 18 を入力し、コンパイル、実行せよ。
2. 図 18 の A 以外のいくつかのクラスから print メソッドを削除し、コンパイル、実行してみよ。実行結果を考察せよ。
3. B, C, D, E に独自のメソッドを作り、a.[i].xxx()(xxx は作成したメソッド名) でアクセスするようにプログラムを作成し、コンパイル、実行してみよ。エラーメッセージが表示された場合は、なぜなのかを考えよ。
4. その他、ここで疑問を感じたら、それをプログラムにし、コンパイル可能かテストしてみよう。エラーメッセージが出た場合、そのメッセージを良く読み、どうしてエラーとなったか理解しよう。

4.10 static

static というキーワードをつけるとどうなるのかについては、プログラムが実行されているときのメモリの様子を知ることが理解の手助けとなる。ここではプログラムが実行されているときの環境 (実行時環境) について簡単に説明した後、Java における static の役割を示す。

4.10.1 実行時環境について少々

いま、図 19 の C 言語のプログラムがあり、コンパイル、実行することを考えよう。

このプログラムはコマンドラインから与えられた数字に対する階乗計算を行って出力する。いま、コマンドラインから 5 が与えられたとしよう。13 行目では与えられた文字 5 を数値の 5 に変換し、関数 fact に渡している。fact では 5 を受け取り、条件分岐で n<1 を満たさないため、さらに fact を引数 4 で呼び出す。これを引数 0 で呼び出すまで繰り返す。

```
 1 #include <stdio.h>
 2
 3 int fact(int n){
 4   if (n<1){
 5     return 1;
 6   }
 7   else {
 8     return n * fact(n - 1);
 9   }
10 }
11
12 main(int argc, char* argv[]){
13   printf("%d\n", fact(atoi(argv[1])));
14 }
```

図 19: 階乗 (factorial) 計算を行う C プログラム

引数 0 で呼び出されたら if 文の条件式を満たすので 1 を返す。返された方はその結果と自身が持っている n の値 (ここでは 1) をかけたものをさらに呼び出し元に返す。その呼び出し元は返された結果の 1 と自身が持っている n の値 (ここでは 2) をかけたものをさらに呼び出し元に返す。このようにして最終的に最初に fact を呼び出した main に値 120 が返される。

ここで一部の人は fact が fact を呼び出すと前に呼び出されたときの値はどこにいってしまうのか不思議に思うかもしれない。

関数の呼び出しにはスタックを用いて、各呼び出しに対して一時的な記憶場所を用意することになっている。

この様子を図 20 に示す。

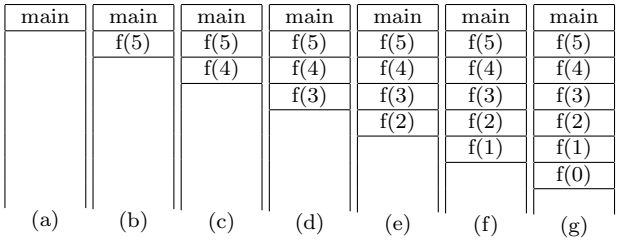

図 20: fact の実行の様子

図ではスタックは下向きに伸びている。図の (a) の状態がプログラムが実行されたところ (すなわち、main 関数が呼ばれたところ) を示している。main 関数内で宣言された変数はこの領域に置かれる。そして fact が呼ばれるとその 1 つ右隣の (b) の状態になる。この例では fact 内に変数宣言はないが、もしあればそれ

らの記憶場所はこの呼び出しで確保された領域に置かれる。この時点でさらに fact を呼び出すと図でもう 1 つ右隣の (c) の状態となる。このようにして fact(0) が呼ばれるまでスタックには記憶領域が積まれていく。逆に関数が計算を終えて呼び出し元に制御が戻る際にはスタックからその関数の領域が下ろされる。

もう 1 つ例を示そう。

```
 1 #include <stdio.h>
 2
 3 int hoge(){
 4   int x = 0;
 5
 6   return x++;
 7 }
 8
 9 main(){
10   int i;
11
12   for(i=0; i<10; i++){
13     printf("%d\n", hoge());
14   }
15 }
```

図 21: notstatic.c

```
 1 #include <stdio.h>
 2
 3 int hoge(){
 4   static int x = 0;
 5
 6   return x++;
 7 }
 8
 9 main(){
10   int i;
11
12   for(i=0; i<10; i++){
13     printf("%d\n", hoge());
14   }
15 }
```

図 22: static.c

図 21 と図 22 はほとんど同じだが 4 行目に static があるかないかが違う。それぞれコンパイルし、実行すると図 21 のプログラムでは 0 を 10 回出力する。図 22 のプログラムでは 0 から 9 までの値を出力する。

static をつけるかつけないかでこのように結果が変わる。static の意味は「静的」であり、これの対となる言葉は「動的」である。関数呼び出しの実行は動的に行われ、その関数が呼ばれるごとにその関数内の変数の領域がスタック上に確保される。ところで、キーワード static をつけることでその変数をスタック上に動的に取るのではなく、静的に確保することができる。図 20 でいえば main 用にとられた領域のひとつ前 (上) に動的な呼び出しとは関係ない静的な変数の領域が取られていると思えばよい。すなわち、関数呼び出しとは無関係にその変数の領域は確保されているため、関数の実行が終わってもその領域は解放されない。したがって、値をいつまでも残すことが可能となる。つまり、どの呼出しからも共用される。

演習 11

1. 図 21 と図 22 を入力し、コンパイル、実行してみよ。
2. 両者の違いを考察せよ。

4.10.2 C 言語の static

さて、C 言語で開発をする際、一般的には複数のファイルにプログラムを分割する。いまファイル A.c(図 23) とファイル B.c(図 24) の 2 つに分けてプログラムが作成されているとしよう。

```
 1 #include <stdio.h>
 2
 3 int x;
 4 int y;
 5
 6 void hoge(){
 7   printf("hoge!\n");
 8 }
 9
10 void print(){
11   hoge();
12 }
```

図 23: A.c

この 2 つのプログラムをコンパイルし、実行することはできる。次のようになる。

```
 1 #include <stdio.h>
 2
 3 void hoge();
 4 void print();
 5
 6 extern int x;
 7 extern int y;
 8
 9 int main(int argc, char* argv[]){
10   x = 3;
11   y = 4;
12
13   hoge();
14   print();
15
16   printf("x(%d) + y(%d) = %d\n", x, y, x+y);
17 }
```

図 24: B.c

```
uni% cc A.c B.c
A.c:
B.c:
uni% a.out
hoge!
hoge!
x(3) + y(4) = 7
```

A.c で宣言された変数 x，y は B.c の中で外部で宣言されたことを extern をつけて宣言することで参照できるようになる。また、A.c で宣言された関数 hoge と print はそれぞれ B.c から呼び出すことができる。

ここで x の宣言と hoge の宣言にそれぞれ static を付けるとどうなるか。

x や hoge はファイルの外部から参照できなくなる。すなわち、アクセスが制限される。

演習 12

1. 図 23 と図 24 を入力し、コンパイル、実行してみよ。
2. x の宣言の前に static を入れ、コンパイルしてみよ。
3. 同様に hoge の宣言の前に static をつけたらどうなるか。
4. 起こったことを考察せよ。

4.10.3 Java における static

Java ではキーワード static を用いるとそのクラスに対して「1 つだけ」を意味する。

データフィールドに対する static

あるクラス内で、データフィールドに static を冠して宣言をした場合、そのプログラムが実行されている間は、たとえ、そのクラスのインスタンスが複数存在したとしても、そのデータはただ 1 つだけしか存在しない。言い替えれば、複数のそれらのインスタンスでそのデータを共有しているともいえる。

```
 1 class Hoge {
 2   static int x;
 3 }
 4
 5 class StaticTest {
 6   public static void main(String[] args){
 7     Hoge h1 = new Hoge();
 8     Hoge h2 = new Hoge();
 9     Hoge h3 = new Hoge();
10
11     h1.x = 3;
12     System.out.println(h2.x);
13
14     Hoge.x = 4;
15     System.out.println(h3.x);
16    }
17  }
```

図 25: StaticTest.java

図 25 ではクラス Hoge には 1 つのフィールドがあり、static を冠している。

クラス StaticTest では main メソッドを定義している。Hoge の実体を 3 つ作っている。11 行目では変数 h1 を介してクラス Hoge の static なフィールドに値を代入している。12 行目では h2 を介してその値を表示している。また 14 行目ではクラス名 Hoge を指定し、そのフィールドに値を代入している。15 行目では h3 を介して変更された値を表示している。

すなわち、Hoge の各実体とは独立にただ 1 つだけフィールド x のための領域がメモリ上に取られる。プログラムの記述としては各実体を参照する変数 (h1 など) を介してそのフィールドにアクセスすることもできるし、クラス名を指定してそのフィールドにアクセ

スすることもできる。static フィールドへのアクセスを明示するためにクラス名を指定したフィールドへのアクセスを記述するのが望ましい。

この static なデータフィールドは C で static を冠した関数内のローカル変数と同様に捉えることができる。

演習 13

1. 図 25 を入力し、コンパイル、実行してみよ。
2. 2 行目の static をはずすとどうなるか試してみよ。
3. h1, h2, h3 のいずれかを Hoge にしたらどうなるか試してみよ。
4. 同様に h1, h2, h3, Hoge に関して適当に変更してみることで、static の効果を考察せよ。

メソッドに対する static

メソッドの場合もデータフィールドの場合と同様である。static をつけてメソッドを宣言すると各インスタンスに作用するメソッドではなくクラス全体として提供するメソッドとなる。

例えば、これまで何の説明もなく System.out.println(...) を使用してきたが、これは out クラスの println メソッドを呼び出している。このメソッドは static に宣言されているのでクラスの実体を作る必要はない (System などドットで区切って並べる表現については別途説明をする)。

本来、オブジェクト指向の考え方からいけば、データを考えるときその操作もあわせて考え、まとめるという発想になるのだが、プログラムを記述する際、なんらかの汎用的に使える便利な機能 (メソッド) も提供できたほうがよい。static メソッドはその機能を提供する。例えば、println のように表示するという機能 (関数というかメソッドというか) だけが欲しい場合、いちいちオブジェクトを生成し、そのメソッドを呼び出すのではなく、汎用的に使えるメソッドとしての println があると便利である。

```
 1 class Point {
 2   int x;
 3   int y;
 4
 5   int getX(){
 6     return x;
 7   }
 8
 9   int getY(){
10     return y;
11   }
12
13   double distance(){
14     return java.lang.Math.sqrt(x*x+y*y);
15   }
16 }
17
18 class PointTest3 {
19   public static void main(String args[]){
20     Point p0 = new Point();
21
22     p0.x = 3;
23     p0.y = 4;
24
25     System.out.println("distance ="
                          + p0.distance());
26   }
27 }
```

図 26: PointTest3.java

5 演習の解答例

演習 1, 2

省略

演習 3

1. 省略
2. 図 26 に解答例を示す。

演習 4, 5

省略

演習 6

図 27 に解答例を示す。

```
 1 class Person {
 2   private String name = "nobody";
 3   private int height = 170;
 4   private int weight = 70;
 5
 6   Person(){
 7   }
 8
 9   Person(String n, int h, int w){
10     name = n;
11     height = h;
12     weight = w;
13   }
14
15   void print(){
16     System.out.println("name   = " + name);
17     System.out.println("height = " + height);
18     System.out.println("weight = " + weight);
19     System.out.println();
20   }
21 }
22
23 class PersonTest {
24   public static void main(String args[]){
25     Person p1 = new Person();
26     Person p2 = new Person("nakai", 170, 75);
27     Person p3 = new Person("kousei", 190, 100);
28     Person p4 = new Person("ayumi", 160, 63);
29
30     p1.print();
31     p2.print();
32     p3.print();
33     p4.print();
34   }
35 }
```

図 27: PersonTest.java

演習 7〜演習 9

省略

演習 10

1, 2, 4 は省略。

3. について解説を少々。a はあくまでも A としてインスタンスを扱っている。なので外から見た目は A であるから A のもつフィールドやメソッドしかアクセスできない。継承には 2 つの役割があり、1 つは差分プログラミングのための機能、もう 1 つは多態性を実現する機能である。前者は機能拡張のために存在する。この意味で使う時は、図 18 のような使い方はしない。いま、算術式を与えられると内部で木を構成し、その木の根に対して、計算せよというメッセージを送ると木 (算術式) 全体の値を計算するプログラムを考えてみよう。木は節 (ノード) によって接続されたデータ構造であり、子ノードを持たないノードを葉という。子ノードを持つノードを内部ノードという。この例の場合、内部ノードは演算子を表現するノードであり、葉ノードは数値を表現するノードである。ノードをクラスにすることを考えると、まず、ノードというものを表現する Node クラスを作る。これは 3 つのフィールドを持ち、それぞれ、値、左の子ノード、右の子ノードとする。また、計算をする eval というメソッドも持つ。実際にはこれを継承した Plus, Minus などのクラスで eval を再定義する。これらのクラスでは eval は、左の子ノードと右の子ノードを用いた演算結果を返すように定義する。実際は、左の子ノードの eval を呼んだ結果と右の子ノードの eval を呼んだ結果を用いて演算する。また、葉ノードに相当する Num というノードクラスも Node を継承して作る。このクラスの eval は単にその数値を返す。ここで重要なのは一番上のクラス Node で eval というメソッドを定義し、継承したクラスでは、それぞれが独自の eval を再実装していることである。また、これらの使われ方であるが、いずれのインスタンスにアクセスする場合でも Node クラスとしてアクセスし、実態に応じて別々の eval が実行される。このサンプルを図 28 に示す。super というメソッドは親クラスのコンストラクタを意味する。null は C 言語における NULL と同様に考えて良い。

演習 11〜演習 13

省略

Java のテキストについて

書店へ行くとこれでもかというほど Java のテキストが並んでいる。プログラミングの学び方はさまざまあるが、中井が書籍を選ぶ際の基準を次にあげる (順不同) ので、参考になればと思う。

1. 基本的な要素の説明が載っている ([5], [1], [4])
2. API の使用方法が載っている
3. 実用的な例題プログラムが載っている ([6], [7], [2], [3])

具体的な書籍名は以下の参考文献リストを参照されたい。

参考文献

[1] Arnold, K., Gosling, J., and Holmes, D.: *The Java Programming Language, Third Edition*, ADDISON WESLEY LONGMAN, a Pearson Education Company, 2000.
(邦訳) 柴田芳樹: 『プログラミング言語 Java 第 3 版』, ピアソン・エデュケーション (2001).

[2] Darwin, I. F.: *Java Cookbook*, O'RIELLY, 2002.
(邦訳) 豊福剛 (宇野浩司監訳): 『Java クックブック』, オライリージャパン (2002).

[3] Felleisen, M. and Friedman, D. P.: *A Little Java, A Few Patterns*, MIT Press, 1998.
(邦訳) 庄司速人／庄司裕子: 『Java とピザとデザインパターン』, SOFTBANK (1998).

[4] O'Neil, J.: *Teach Yourself Java*, McGraw-Hill Companies, Inc., 1999.
(邦訳) トップスタジオ (武藤武志監修): 『独習 Java』, 翔泳社 (1999).

[5] van der Linden, P.: *just JAVA 2 FOURTH EDITION*, Sun microsystems, 1999.
(邦訳) 中田秀基: 『just JAVA 2』, ASCII (2000).

[6] 丸の内とら: *10* **日でおぼえる** *Java* **実践教室**, 翔泳社, 2002.

[7] 結城浩: *Java* **言語で学ぶデザインパターン入門**, SOFTBANK Publishing, 2001.


```
 1  class Node {
 2    Node l=null, r=null;
 3    int val = 0;
 4
 5    Node(int v){
 6      val = v;
 7    }
 8
 9    Node(Node l, Node r){
10      this.l = l;
11      this.r = r;
12    }
13
14    int eval(){
15      return 0;
16    }
17  }
18
19  class Plus extends Node {
20    Plus(Node l, Node r){
21      super(l, r);
22    }
23
24    int eval(){
25      return l.eval() + r.eval();
26    }
27  }
28
29  class Minus extends Node {
30    Minus(Node l, Node r){
31      super(l, r);
32    }
33
34    int eval(){
35      return l.eval() - r.eval();
36    }
37  }
38
39  class Num extends Node {
40    Num(int v){
41      super(v);
42    }
43
44    int eval(){
45      return val;
46    }
47  }
48
49  class NodeTest {
50    public static void main(String args[]){
51      Node n1 = new Num(12);
52      Node n2 = new Num(23);
53      Node n3 = new Num(34);
54
55      Node p = new Plus(n1, n2);
56      Node m = new Minus(p, n3);
57
58      System.out.println(m.eval());
59    }
60  }
```

図 28: NodeTest.java