プログラミング言語各論I
第2回講義ノート

2005年度担当: 中井央

2005年9月8日

1 Java API

C言語もそうだし、Javaもそうであるが、プログラミング言語として用意されている構文は非常にシンプルなものである。C言語ではライブラリと呼ばれるプログラムを組むのに便利なように用意された関数群がある。ほとんどすべてのプログラムではこのような用意されたライブラリ (の関数) を用いて作成される。例えば、画面表示を行う `printf` という関数はライブラリとして提供されているものである。`printf` などは標準的に使用するものであるため、明示的なライブラリの指定がなくとも利用できるが、数学のライブラリなどは明示的にライブラリを指定する必要がある[1] 。

すなわち、C言語として用意されているものは本来、関数を記述するための宣言、変数の宣言、`if` や `while` などの制御文、代入文、(演算) 式などである。

`fopen` や `fgets` などもライブラリとして用意されたものである。

さて、Javaも発想としては同様である。標準的に一般的に使用するものとして、あらかじめ言語処理系で用意しているクラスライブラリ (class library) がある。

Javaのドキュメントはインターネット上でも公開されている。Java 2 SDK Standard Editionの日本語ドキュメントは次のURLで参照できる[2]。

`http://java.sun.com/j2se/1.4/ja/docs/ja/`

[1] `-l` で始まるオプションを使う。数学のライブラリを使用する場合は`-lm` を指定する。

[2] 現在、uni上で利用できるJavaのバージョンは1.4であるため、この講義では1.4までの機能を用いることにする。

2 配列

Javaにおける配列はC言語における配列とは少し異なる。まず、図1のサンプルプログラムを見てみよう。(14行目は紙面の都合上、折り返している)

```
 1 class ArrayTest {
 2   public static void main(String[] args){
 3     int a [] = new int[10];
 4     int i;
 5
 6     for(i = 0; i<10; i++){
 7       a[i] = i;
 8     }
 9
10     for(i = 0; i<10; i++){
11       System.out.println(a[i]);
12     }
13
14     System.out.println("The length of a = "
                          + a.length);
15   }
16 }
```

図 1: `ArrayTest.java`

例えば `int` の配列を宣言したい場合、C言語では `int a[10]` のように宣言するが、Javaでは図1のように `new` を使ってオブジェクトとして割り当てをする。

このため、配列である変数だけを宣言しておき、別のところで `new` を実行することも可能である。

```
int a[];
...
a = new int[10];
```

2.1 配列の長さ

Java では配列をオブジェクトとして扱う。配列オブジェクトには length というフィールドがあり、その配列の長さが定義されている。

Java では実行時に配列の添え字がその値の範囲を超えていないかチェックされる。例えば、図 1 の 10 行目の 10 を 11 に変更し、添字の範囲を超えたアクセスを起こさせると次のようなエラーメッセージが表示される。(紙面の都合上、折り返している)

```
Exception in thread ‘‘main’’ java.lang.
        ArrayIndexOutOfBoundsException
        at ArrayTest.main(ArrayTest.java:11)
```

2.2 配列の初期化

C 言語と同様に配列を宣言すると同時に初期化することができる。例えば次のようになる。

```
int a [] = new int[] { 3, 1, 4, 1, 5, 9, 2, 6 };
```

また、図 2 の 3 行目のように初期化することもできる。3 行目の{と}ではさんだものを配列の初期化子と呼ぶ。初期化子を用いることで配列の要素数がわかるため、あえて new を書かなくてもよいわけである。

```
 1 class ArrayTest2 {
 2   public static void main(String[] args){
 3     int a[] = { 3, 1, 4, 1, 5, 9, 2, 6 };
 4     int i;
 5
 6     for(i = 0; i<a.length; i++){
 7       System.out.println(a[i]);
 8     }
 9   }
10 }
```

図 2: ArrayTest2.java

2.3 配列の配列

Java には多次元配列はない。しかし、配列の配列という考え方はある。この区別について以下に説明をする。

例えば、次のような宣言を考えよう。

```
int a[][];
```

この宣言では int を要素とする配列、を要素とする配列の宣言をしている。すなわち、a という配列の一個の要素は「int の配列」なのである。このことを示すプログラムを図 3 に示す。

```
 1 class ArrayTest3 {
 2   public static void main(String[] args){
 3     int i;
 4
 5     int a[][] = new int[][] {
 6       new int[]{0},
 7       new int[]{0, 1},
 8       new int[]{0, 1, 2},
 9       new int[]{0, 1, 2, 3}
10     };
11
12     for(i=0; i<a.length; i++){
13       System.out.println(a[i].length);
14     }
15   }
16 }
```

図 3: ArrayTest3.java

図 3 では 5 行目から 10 行目で配列の配列 a の宣言をしている。ここでは各要素として int の配列の実体を割り当てている。それぞれ違う長さの配列で初期化しているので、このプログラムの実行結果は、1, 2, 3, 4 が表示される。

さて、配列の配列の宣言の仕方としては 1 つの要素が配列であることがわかっているため、とりあえず何要素あるかだけ宣言しておき、後に必要に応じて各要素の実体を割り当てることも可能である。

```
int a[][] = new int[4][];
...
a[0] = new int[]{ 6, 0, 2 };
a[1] = new int[]{ 1, 4, 1, 4 }
...
```

配列の配列の扱いがわかれば配列の配列の配列といった (いくつ「配列の」がついても) 配列を要素とする配列の扱い方は同じである。

最後に注意を 1 つ。配列の宣言を行った場合、その変数名はその「配列というオブジェクト」を指し示し

ているに過ぎない。したがって図 4 のように指し示す先を変更しても問題はない。(それをどう使うかはプログラマ次第)

```
 1 class ArrayTest4 {
 2   public static void main(String[] args){
 3     int a[] = new int[] { 1, 7, 3, 2 };
 4     int b[] = new int[] { 2, 2, 3, 6, 0, 6 };
 5
 6     int i;
 7
 8     a = b;
 9
10     for(i=0; i<a.length - 1; i++){
11       System.out.print(a[i] + ", ");
12     }
13     System.out.println(a[i]);
14   }
15 }
```

図 4: ArrayTest4.java

演習 1

1. 図 1 を入力し、コンパイル、実行せよ。
2. 図 2 を入力し、コンパイル、実行せよ。
3. 図 3 を入力し、コンパイル、実行せよ。
4. 図 3 を改造し、配列 a が持つ各配列をそれぞれ一行に一配列、その要素をコンマで区切って出力するようにせよ。ファイル名を ArrayTest3d.java とする。実行結果は次のようになる。

```
uni% java ArrayTest3d
0
0,1
0,1,2
0,1,2,3
```

5. 図 4 を入力し、コンパイル、実行し、何が起こったのか確認せよ。

3 Object クラス

Java ではすべてのクラスは Object という名のクラスを継承していることになっている。これは明示的な extends キーワードを用いず、暗黙のうちに行われている。図 5 はクラス Object を利用したサンプルプログラムである。ただし、このプログラムは正しく動かない。

```
 1 class Hoge {
 2   private String name;
 3
 4   Hoge(String name){
 5     this.name = name;
 6   }
 7
 8   public void print(){
 9     System.out.println(name);
10   }
11 }
12
13 class ObjectTest {
14   public static void main(String[] args){
15     Object x[] = new Object[5];
16
17     x[0] = "xyz";
18     x[1] = new Hoge("abc");
19     x[2] = new Integer(256);
20     x[3] = new Double(3.14159);
21
22     x[1].print();
23   }
24 }
```

図 5: ObjectTest.java

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。

```
uni% javac ObjectTest.java
ObjectTest.java:22: cannot resolve symbol
symbol  : method print  ()
location: class java.lang.Object
    x[1].print();
        ^
1 error
```

まず、図 5 を解説していこう。

1〜11: クラス Hoge の宣言。コンストラクタでは文字列を受け取り、フィールド name にそれを指させている。

13〜24: main メソッドを含む ObejctTest クラスの宣言。

15: Object 型の配列を宣言している。Object 型は全てのクラスのスーパークラスなので、どのようなクラスのインスタンスも保持することができる。

17〜20: 各種クラスのインスタンスを生成し、配列 x の要素として格納している。

22: x[1] はクラス Hoge のインスタンスを保持しているはず、と思い、そのメソッドを呼び出そうとしている。コンパイル時にはこの行に対し、エラーメッセージが表示されている。

22 行目でなぜエラーとなるのか??

一旦 Object 型として配列の要素に格納しているものはそれ以降も Object 型として扱われる。Object 型で宣言しているメソッドを呼び出す分には問題がないが、22 行目では Object 型としては所有していないメソッド print の呼び出しをしている。このため、コンパイル時にエラーとなる。

しかし、x[1] の実体は Hoge クラスのインスタンスであるから print の呼び出しができてもよいはずである。

クラス Object 型の変数を使うとどんなクラスのインスタンスであっても扱えるため、便利である一方、その変数が指している実際のクラスのメソッド (やフィールド) を利用したい場合、キャストが必要になる。

22 行目は次のように記述することで print を呼び出すことができるようになる。

```
((Hoge)x[1]).print();
```

なお、このことは Object 型に限った話ではなく、あるクラスのスーパークラスである A 型として扱っていて、しかしそのサブクラス以降で導入された (A にはない) メソッドを利用する時にはこのようなキャストが必要となる。なお、Java1.5 では、総称型が導入されたので、これまで Object クラスの変数として扱ってきていたものを実際の型にして扱える場合もある。この講義ではその詳細は省略する。

4 ラッパ (wrapper) クラス

これまで各クラスは暗黙でクラス Object を継承していることを述べた。このことは便利に使える場合がある。例えば、さまざまなクラスのオブジェクトをまとめて扱いたい場合である。

後にリンクトリストなどのデータ構造について触れるが、データ構造はそこに保持されるデータが何であろうと、その保持する要素を連結したものである。int のリスト、double のリスト、クラス A のリストと作り分けるより、入れるものは Object として扱ってしまうことで、データ構造を表すクラスを簡潔に実現することができる。この考え方とは別にそれぞれのデータ構造が持つ型をプログラム作成時に指定できるような仕組み (C++でいうテンプレート) についても議論があるが、ここでは扱わないことにする[3] 。

さて、Object 型として扱えるのはクラスとして宣言されたものだけであるため、基本型を要素としたい場合に困ることになる。これを解決するため、ラッパクラスが用意されている。

ここでは Double クラスを紹介する。

4.1 Double

ここでは Java のドキュメントから Double クラスの概要を紹介する。

Double クラスは、プリミティブ型 double の値をオブジェクトにラップする。Double 型のオブジェクトには、型が double の単一フィールドが含まれる。

このクラスには、次の定数が定義されている。

- double 型の正の最大値を表現する MAX_VALUE
- double 型の正の非ゼロ最小値を表現する MIN_VALUE
- double 型の非数[4] を表現する NaN
- 負の無限大値を表現する NEGATIVE_INFINITY
- 正の無限大値を表現する POSITIVE_INFINITY

コンストラクタには次の 2 つがある。

Double(double value) double 型の値を受け取り、それを保持する。

Double(String s) double 型を表す文字列を受け取る。

数あるメソッドからいくつかを抜粋して以下に記す。

double doubleValue()

このオブジェクトが持つ double 値を返す。

[3] 前節末にも記載したが、Java 1.5 からは総称型を利用できる。

[4] 数学的に定義できない演算の結果を表す。例えば 0 による除算の結果など。

static double parseDouble(String s)

与えられた文字列を構文解析 (parse) して、double の値を返す。

public String toString()

この Double オブジェクトが表す double 値を引数とした toString() の呼び出し結果を返す。要は、その数値を表現する String 型のオブジェクトを返す。

static Double valueOf(String s)

引数に与えられた文字列が表現する double 型の値を保持する Double のオブジェクトを返す。

ラッパクラスのよいところは、基本型をオブジェクトとして保持する際、あったら便利と思われる文字列と値との変換を行うメソッドを用意してあることである。

各基本型に対するラッパクラスは次のとおり。

byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character

演習 2

1. コマンドラインの第 1 引数で与えられた整数を表す文字列を数値に変換するプログラムを作れ。クラス名を `atoi` とする。
2. コマンドライン引数に 1 つ以上の整数が与えられた場合、その合計を表示するプログラムを作れ。クラス名を `atoi2` とする。
3. コマンドラインの第 1 引数で与えられた実数を表す文字列を数値 (double 型) に変換するプログラムを作れ。クラス名を `atod` とする。

5 String

String は文字列を表現するクラスである。C 言語風に言えば、String は文字列へのポインタ型であるともいえる。なお、ポインタで指された先を変更することはできない。C プログラムでは図 6 のように記述するとポインタが指す先を変更できない。

```
1 #include <stdio.h>
2
3 main(){
4   const char* name = "abc";
5
6   name[2] = 'x';
7 }
```

図 6: `String1.c`

図 6 をコンパイルすると次のようにエラーメッセージが表示される。これはキーワード `const` をつけることによって、ポインタ変数 `name` が指す中身を定数とみなすように指示しているからである。

```
uni% cc String1.c
cc: "String1.c", line 6: error 1549: Modifiable
lvalue required for assignment operator.
```

ところで 6 行目を次のように変えたプログラムはコンパイルすることができる。つまり、ポインタ変数が指す中身ではなく、ポインタ変数の値自体は変更できるのである。

```
name ="xyz";
```

今度は 4 行目を次のように変えてみる。`const` の位置が変わっているのに注意。

```
char* const name = "abc";
```

この記述では `name` という変数を定数として扱うため、コンパイル時には次のエラーメッセージが出力される。

```
cc: "String1.c", line 6: error 1549: Modifiable
lvalue required for assignment operator.
```

さて、String クラスであるが、基本的には図 6 と同様に考えて文字列を扱っている。もちろん、クラスと

していろいろと機能を用意している。ここでは文字列の扱い方を簡単に述べるとする。

String 型の変数で文字列を指す場合、本来は次のように記述する。

```
String a = new String("abc");
```

これを省略して次のように記述できる。

```
String a = "abc";
```

また、文字列の連結には + 演算子を利用できる。文字列"abc"と"xyz"の連結は次のようになる。

```
... "abc" + "xyz";
```

なお、+ のオペランドの一方に String 以外が来る場合、そのオペランドの toString というメソッドがまず呼び出され、String 型へと変換されてから文字列として扱われる。例えば、次のような記述があると 3.14 の部分は"3.14"という文字列へ変換されてから連結される (3.14 は double 型のリテラル)。

```
... "abc" + 3.14;
```

この節の最後として文字列の比較について述べる。文字列を指し示している変数は String 型であるが、比較には注意が必要である。単純に String 型の変数どうしを==で比較するとアドレスの比較となる。すわなち、同一オブジェクトであれば真であるがそうでなければ偽となる。内容を比較するには String クラスの equals メソッドを利用する。この違いを示す例題プログラムを図 7 に示す。

4 行目と 6 行目でややこしいことをしているのには理由がある。単純に 4 行目で String y = "xyz"とすると同じアドレスを保持することになる。これはコンパイラが"xyz"という文字列の出現を管理していて複数回出てきてもメモリ上の一箇所にリテラルを割り当てるからである。

したがって、図 7 のように記述することで、違うアドレスに格納された同じ内容の文字列を作り出すことができる。

```
 1 class StringSample {
 2   public static void main(String[] args){
 3     String x = "xyz";
 4     String y = "x";
 5
 6     String z = y + "yz";
 7
 8     if (x == z){
 9       System.out.println("x == y");
10     }
11     else {
12       System.out.println("x != y");
13     }
14
15     if (x.equals(z)){
16       System.out.println(
           "x and y are the same string");
17     }
18     else {
19       System.out.println(
           "x and y are not the same");
20     }
21   }
22 }
```

図 7: StringSample.java

図 7 の 8 行目ではオブジェクトのアドレスの比較をしている。C 言語風に言えばポインタの比較である。一方、15 行目ではオブジェクトの中身の比較をしている。String クラスの場合、メソッド equals は引数に与えられたオブジェクトが持つ文字列と自身が持つ文字列を比較する。

なお、String クラスには intern というメソッドが用意されている。これを用いるとその String クラスのオブジェクトが保持する文字列はプログラム内部のある領域に格納されるのであるが、既に同じ文字列が存在する場合、同一オブジェクトを指すようになる。これによって文字列の比較をオブジェクトの比較に置き換えることもできる。

substring メソッドの使用

String クラスには多くのメソッドが用意されているが、この中で substring メソッドを使用する例を図 8 に示そう。substring やその他にどのようなメソッドがあるかはマニュアルを参照して欲しい。

実行結果は次のようになる。

```
 1 class SubstringTest {
 2   public static void main(String[] args){
 3     String a = "Java is great.";
 4     System.out.println(a);
 5     String b = a.substring(5);
 6     System.out.println(b);
 7     String c = a.substring(5, 7);
 8     System.out.println(c);
 9     String d = a.substring(5, a.length());
10     System.out.println(d);
11   }
12 }
```

図 8: SubstringTest.java

```
uni% java SubstringTest
Java is great.
is great.
is
is great.
```

メソッド substring はオーバーロードされていて、int 型の引数を 1 つとるものと 2 つとるものがある。引数を 1 つとるメソッドはその引数が指し示す文字位置以降の部分文字列を新たな String 型のオブジェクトとして返す。引数を 2 つとるメソッドは第 1 引数が指す文字位置から第 2 引数が指す文字位置までの間の部分文字列を新たな String 型のオブジェクトとして返す。

演習 3

1. 図 7 を入力し、コンパイル、実行せよ。
2. コマンドライン引数を受け取り、それを 1 行に 1 つずつ表示するプログラムを作れ。クラス名を StringTest とする。
3. コマンドライン引数を受け取り、各引数については、1 行に 1 文字表示し、引数と引数の間には空行を入れるプログラムを作れ。ただし、全引数の最後に空行が来ても良いとする。クラス名を StringTest2 とする。
4. コマンドラインの第 3 引数として与えられた文字列に対し、第 1 引数で与えられた数字の位置から第 2 引数で与えられた数字の位置までの部分文字列を表示するプログラムを作れ。位置は、第 1 文字目を 0 とすることに注意せよ。クラス名を StringTest3 とする。例えば、java StringTest3 2 4 StringBuffer と入力した場合、rin と返される。

   なお、与えられた整数を表す文字列を数値に変換するには Integer クラスで定義されたメソッドを利用するとよい。詳細はマニュアルを参照のこと。

6 toString

Object クラスには toString メソッドが定義されている。このメソッドの目的はそのオブジェクトに関連する文字列を返すことである。これまでに使用してきた println などのメソッドはクラス型の変数を受け取るとその toString メソッドを呼び出す。

例えば次のようなクラス MyName があったとしよう。

```
class MyName {
  private String name;

  ...
}
```

(ベタな例だが…) このクラスはコンストラクタで受け取った名前を保持する。このクラスに関して表示したいときは name フィールドの文字列を表示したい。したがって、次のように toString を (再) 定義 (オーバーライド) しておく。

```
public String toString(){
  return name;
}
```

もちろん、リンクトリストの要素にする、個人情報を表わす Person というクラスなら、名前、年齢 (生年月日)、身長、体重、といったそのクラスが持つフィールドを表示するための文字列を作り出し、それを返すようにすればよい。

演習 4

1. 個人を表すクラス Person を適当に作成し、toString メソッドを持つようにせよ。このプログラムのファイル名を PersonTest.java とし、main

メソッドを含む PersonTest クラスを作成し、その中で Person クラスのインスタンスを作り、それを System.out.println の引数に与えたらどうなるか、試してみよ。

7 さらに継承について

継承をする際、継承の仕方にいくつかオプションがある。ここではその機能を紹介する。

7.1 final

クラスの宣言の際、キーワード final をつけることで、そのクラスをさらに継承することを禁止することが出来る。

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

図 9: TestFinal.java

図 9 に例を示そう。1〜7 行目でクラス A を、9〜15 行目でクラス A を継承したクラス B を宣言している。ただし、クラス A は宣言時にキーワード final がつけられている。このプログラムをコンパイルすると次のようにエラーメッセージが表示される。

```
TestFinal.java:9: cannot inherit from final A
class B extends A {
                ^
1 error
```

また、クラス全体ではなくメソッドに対して final をつけることも出来る。こうすると、そのメソッドをオーバーライドすることを禁止したことになる。その例を図 10 に示す。

```
 1 class A {
 2   int x;
 3
 4   public final void print(){
 5     System.out.println("class A");
 6   }
 7 }
 8
 9 class B extends A {
10   int x;
11
12   public void print(){
13     System.out.println("class B");
14   }
15 }
16
17 class TestFinal2 {
18   public static void main(String[] args){
19     A a = new A();
20
21     a.print();
22   }
23 }
```

図 10: TestFinal2.java

このプログラムをコンパイルしようとすると次のエラーメッセージが出力される。

```
TestFinal2.java:12: print() in B cannot override
print() in A; overridden method is final
  public void print(){
              ^
1 error
```

継承という機能はスーパークラスを特殊化するものである。クラス A を継承してクラス B を作成する場合、クラス A の機能を受け継ぐが、必要に応じてクラス A の持つ機能 (たとえばあるメソッド) を上書きし、自分に適したように修正を施すわけである。こうする

ことでもとの機能を保ちつつ、自分自身のオリジナルな機能を加えることが可能となる。

しかし、継承したクラスは継承もとのクラスとして見かけ上振舞うことが出来る。つまり、クラス A の変数にクラス B の実体を指させることができる。そうするとオリジナルのクラスの機能を意図しない方向に拡張される可能性がある。このため、拡張によって意図しない振る舞いになることをさけるため、オーバーライドを禁止する機構が用意されている。

7.2 abstract

キーワード abstract はサブクラスを設けなければならないことを指示するために記述する。

abstract をメソッドの宣言につけた場合、そのメソッドの本体は記述できない。また、abstract を付与したメソッド宣言を 1 つでも含むクラスの宣言には abstract をつけなければならない。

図 11 に例を示そう。

```
 1 class A {
 2   public abstract void hoge(){
 3     System.out.println("hoge");
 4   }
 5 }
 6
 7 class TestAbstract {
 8   public static void main(String[] args){
 9   }
10 }
```

図 11: TestAbstract.java

図 11 をコンパイルしようとすると以下のエラーメッセージが出力される。1 つ目はクラス A には abstract を付けなければならない旨であり、もう 1 つは 2 行目のメソッド宣言は abstract がついているのだから、本体を持てない旨、である。

```
TestAbstract.java:1: A should be declared
abstract; it does not define hoge() in A
class A {
^
TestAbstract.java:2: abstract methods cannot
have a body
  public abstract void hoge(){
                       ^
2 errors
```

逆に図 12 はコンパイルに成功する。

```
 1 abstract class A {
 2   public void hoge(){
 3     System.out.println("hoge");
 4   }
 5 }
 6
 7 class TestAbstract2 {
 8   public static void main(String[] args){
 9   }
10 }
```

図 12: TestAbstract2.java

abstract というキーワードをどういうときに使うかについて [2] の 166 ページの例をもとにして説明しよう。

計算機の画面上に現れるウィンドウに関する Java のクラスがあるとしよう。スクロールバーやダイアログボックス、テキストフィールドなどウィンドウの中に現れるさまざまな「部品」が考えられる。これをそれぞれクラスと捉えたいが、これを抽象化してみると「部品」というクラスが考えられる。個々の部品は「部品」クラスをもう少し具体的にしたものである。すなわち、「部品」クラスを継承し、特殊化したものである。「部品」という概念そのものは実際にウィンドウの中に表示される「もの」ではない。

ウィンドウの中に部品を配置することを考えてみよう。add というメソッドがあったとき、個々の部品を追加するためにメソッドをいくつも用意するよりは、「部品」を追加する、というメソッドがあればよいことに気づく。そう、個々の部品は「部品」クラスのサブクラスなのだから。

そして、「部品」クラスだけではプログラムとして何の役にも立たない。つまり、具体的な「部品」のク

ラスを作って、その実体を生み出さない限り、表示すべきものは何もないからである。

こう考えると、あるクラスを abstract にするべきなのは次の 3 つの条件が満たされたときである。

1. サブクラスがいくつも出来る
2. すべてのサブクラスをスーパークラスのインスタンスとして操作したい
3. スーパークラスだけではオブジェクトとして意味をなさない

演習 5

1. 図 9、図 10、図 11、図 12 を入力し、コンパイルしてどうなるか確認せよ。

8 インタフェース

Java におけるインタフェースとは多重継承をたくみに回避するための方策である。

8.1 多重継承の問題点

プログラミング言語 C++では多重継承を取り入れたが、多重継承を行う場合、やっかいな問題が起こることがある。

C++は C 言語をベースとし、クラスの機能を導入することでオブジェクト指向プログラミング言語に C 言語を昇格させたものといってもよいだろう。

さて、いま、図 13 のようなクラス A とクラス B があるとしよう。

見てわかるとおり、両者の違いはメンバ関数 (C++ではそう呼ぶ)print が表示する内容として A か B かだけである。

さて、いかにもサンプルらしい例題で申し訳ないが、この両者を継承するクラス C を考えよう。

```
class C : A, B {
  ...
}
```

継承は親の機能を受け継ぐものであった。いま、例えば、クラス C の変数 c があったとして、c.x はクラス

```
class A {
  int x;

  public void print(){
    printf("class A : \n");
  }
}

class B {
  int x;

  public void print(){
    printf("class B : \n");
  }
}
```

図 13: 多重継承の問題の説明用のプログラム (C++風)

A の x だろうか、クラス B の x だろうか。c.print() はどうだろうか??

悩ましい問題である。

8.2 Java では??

もちろん、多重に継承はできないので多重継承のようだが、そうではないような工夫が必要となる。

ここで少し、クラスの設計について考えてみよう。クラスにはデータとその操作がある。「操作」を「機能」といってもよい。「挙動」といってもよいかもしれない。

クラスは継承することで「特殊化」すると述べた。機能あるいは挙動は特殊化されたそれぞれのクラスで異なるかもしれない。ただし、その機能の概念は同じである場合も多い。

例えばいま、整数を表すクラスを考えよう。実際に Java では基本型の整数をオブジェクトとして扱うためにラッパクラス (wrapper class) を設けている。整数は整数どうし比較できる。また、実数を表すクラスも考えることができる。実数どうしも比較できる。さらにソートを考えたとき、なんらかのクラスもお互い比較できなければならない。

「比較できる」という概念はそれぞれ共通でも、それぞれの比較は実装の方式が異なる。整数と実数のクラスだけなら Number というようなクラスを作って継承することも考えられるが、比較をもう少し広く一般的に捉えて、ユーザが定義したクラスにその概念を導

入したいとき、Number というクラスを継承するというわけにはいかなくなる。

比較という概念に必要なのは「比較するメソッド」である。具体的な実装は個々の特殊化されたクラスで考えればよいが、どのようなメソッドを持っておくべきかは比較という概念を表せればよい。つまり、比較するという意味合いの「compareTo」というメソッド名のみ定めておけばよいという発想になる。

ところで、外部とのやり取りをする窓口のことをインタフェース (interface) という。クラスのオブジェクトが外部とやり取りをするには通常メソッドを使う。つまり、オブジェクトの外部とのインタフェースはメソッドである。

少しややこしいが、そのインタフェースのみを定めたクラスもどきを Java ではインタフェースという。インタフェースは次のようにして定義する。

```
interface Hoge {
  ...
}
```

メソッド名とその引数および返り値の型を書き記したものをそのメソッドのシグネチャという。インタフェースで宣言できるのはメソッドのシグネチャと定数である。

インタフェースはある意味で制限を設けたクラスであると解釈できる。インタフェースでは、「継承する」替わりに「実装する」という言葉を使う。つまり、インタフェースではメソッドのシグネチャのみを定義しているため、そのメソッドを実際にどこかで実装しなければならない。インタフェースの実装は次のように記述する。

```
interface Hoge {
  ...
}

class Test implements Hoge {
  ...
}
```

継承では `extends` を使用したが、実装では `implements` を使用する。

インタフェースは複数実装することができる。継承では複数の親クラスを持つことはできない。

インタフェースを実装したクラスは継承と同じようにそのインタフェース型の変数で扱うことができる。

インタフェースの使用については後に具体的な例題で説明する。

9 さらに文字列

これまで文字列を扱うにはクラス String を利用してきた。しかし、このクラスのオブジェクトは与えられた文字列を保持するが、その文字列は固定されていて変更は加えることができなかった。文字列に対して、文字の削除、挿入、追加などを便利に行えるように StringBuffer クラスが用意されている。ここでは StringBuffer クラスを紹介する。

StringBuffer クラスのオブジェクトは内部にバッファを持っていて、そのため、文字列に対して文字の追加や文字の削除が行える。

StringBuffer クラスは `append` というメソッドを持つ。これはオーバーロードにより、様々な型を引数に取り、StringBuffer のオブジェクト (への参照) を返す。引数として定義されているのは `boolean`, `char`, `double`, `float`, `int`, `long`, `Object`, `String`, `StringBuffer` などである。基本型が与えられた場合はその値を表わす文字列が内部で作られ、それを現在保持している文字列の最後に追加した形の StringBuffer オブジェクト (への参照) が返される。実際には自身への参照が返される。具体的な例としては、文字列の連接をする際、コンパイラでは内部的に次のように扱っている。

```
x = "nakai"+" "+"hisashi"
```

に対し、

```
x = new StringBuffer().append("nakai").append(" ")
        .append("hisashi").toString();
```

のように内部で扱っている (紙面の都合上、折り返している)。

文字列`"nakai"`に文字列`" "`を追加し、最後に文字列`"hisashi"`を追加しているわけであるが、StringBuffer クラスはバッファ(C 言語で言う文字の配列) を持っていて、そこに文字列を追加していっている。最初 `new StringBuffer()` が呼ばれたときはまだバッファには何も入っていない StringBuffer オブジェクトができる。StringBuffer オブジェクトは `append` メソッドを持

つのでそれを使って"nakai"をバッファに追加する。このメソッドは自身 (StringBuffer オブジェクト) を返すので、その返ってきたオブジェクトに対し、append(" ") を実行する。すなわち、そのバッファに" "を追加する。もちろん、append が返すのはその StringBuffer オブジェクトだから、それに対してさらに"hisashi"を追加する。そして最後に toString オブジェクトを呼ぶことで、String 型のオブジェクトが得られ、それ (の参照) を x に代入する。

以下で、この他に StringBuffer クラスが持つメソッドをいくつか紹介する。

- char charAt(int index)
  index によって示される位置の文字を返す。index の値が 0 のとき、バッファ内の文字列の 1 文字目の文字を示している点に注意。なお、文字列の長さの範囲以外の値を与えた場合は IndexOutofBoundsException が発生する[5]。

- StringBuffer delete(int start, int end)
  バッファ内の文字列の start 番目の文字から end-1 番目までの部分文字列を削除する。返り値は自身である。

- StringBuffer deleteCharAt(int index)
  バッファ内の文字列の index 番目の文字を削除する。返り値は自身である。

- int indexOf(String str)
  与えられた文字列 str をバッファ内の文字列の先頭から探し、見つかればその位置を返す。見つからなかった場合は-1 を返す。なお、この機能はバージョン 1.4 から導入されている。

- StringBuffer insert(...)
  insert も append と同様に基本型と Object 型、String 型に対して用意されている。例えば、String 用のメソッドのシグネチャは StringBuffer insert(int offset, String str) である。offset は挿入位置である。offset はもちろん、もとの文字列の長さの範囲を超えてはならない。超えた場合には StringIndexOutOfBoundsException が投げられる。

[5] jdk1.4 の日本語版のドキュメントでは文字を「文字列」と間違えて記述している。

- int length()
  このオブジェクトが保持している文字の長さを返す。

- toString()
  バッファ内にある文字列を String 型のオブジェクトにコピーしてそれを返す。

演習 6

1. コマンドラインの第 1 引数で与えられた文字列が回文であるかどうか判定するプログラムを作れ。ここでは作成の条件として、受け取った引数に対し、それを反転した文字列は StringBuffer クラスを用いて表現すること、とする。クラス名を ReverseString とする。
2. コマンドライン引数で与えられた My name is Hisashi Nakai. のような英文に対し、is の後ろに not を入れた文 My name is not Hisashi Nakai. を出力するプログラムを作れ。ここでは文の形式は A is B. のようなものだけを扱うとする。なお、not が入った文を作成するに当たっては StringBuffer を使用することを条件とする。コマンドラインから文を入力するには次のようにする。

   ```
   % java InsertNot "My name is Hisashi Nakai."
   ```

   クラス名を InsertNot とする。

10 Java で使えるデータ構造

Java はその開発環境が提供する API によって、さまざまなデータ構造を扱えるようになっている。ここではそのような API を概観し、その中から LinkedList クラスについて [1] の例を用いて紹介する。

10.1 コレクション API の概要

データの集まりを扱うデータ構造については、昔から議論されてきて、いろいろなものが提案されている。そのようなものをベースに Java では標準 API として、そのデータ構造を扱うためのインタフェースおよびそれらを実装したクラスが提供されている。

そのインタフェースの主なものを次に挙げる。

Collection オブジェクトの集まりを表現するためのインタフェース

List Collection を継承した、線形に並んだデータの集まりを表現するためのインタフェース

Set Collection を継承した、集合 (重複要素を許さない) を表現するためのインタフェース

SortedSet Set を継承し、その要素は整列されている集合を表現するためのインタフェース

これらを実装したクラスとして次のものがある。

- ArrayList
- LinkedList
- Stack
- HashSet
- TreeSet
- LinkedHashSet

10.2 LinkedList

ここでは LinkedList クラスを取り上げる。

図 14 に示すのは Java の言語処理系で用意しているリンクトリストクラスを使った例である ([1] より)。

以下、簡単に説明をする。

1: 後の回で詳しく説明する

6: LinkedList クラスの変数の宣言と実体化

7: Object クラスの実体を作ってリストへ追加

8: 文字列`"Hello"`(のオブジェクト) を作ってリストへ追加

11: リストのイテレータを作成 (後述)

12 〜 13: イテレータによる繰り返しアクセスで各要素を表示

15: 指定した要素 (`"Hello"`(のオブジェクト)) がリスト内にあるかチェック

```
 1 import java.util.*;
 2
 3 public class LinkedListDemo {
 4   public static void main(String[] argv) {
 5     System.out.println(
         "Here is a demo of Java 1.2's" +
         "LinkedList class");
 6     LinkedList l = new LinkedList();
 7     l.add(new Object());
 8     l.add("Hello");
 9
10     System.out.println(
         "Here is a list of all the elements");
11     ListIterator li = l.listIterator(0);
12     while (li.hasNext())
13       System.out.println(li.next());
14
15     if (l.indexOf("Hello") < 0)
16       System.err.println(
           "Lookup does not work");
17     else
18       System.err.println("Lookup works");
19   }
20 }
```

図 14: `LinkedListDemo.java`

`indexOf` メソッドは、引数として受け取ったオブジェクトが登録されている場合はその位置を整数で返し、登録されていない場合 -1 を返す。

イテレータはデザインパターンの 1 つである。リンクトリストをはじめとして、データを保持するデータ構造はいくつもある。このようなデータ構造に対して、「順にアクセスする」という概念を抽象化したのが、イテレータである。

Java では jdk1.2 から標準 API でイテレータを提供している。これはインタフェースの形で提供されている。`Iterator` インタフェースが持つメソッドは次の 3 つである。

- `boolean hasNext()`
  さらに要素がある場合、`true` を返す
- `Object next()`
  次の要素を返す
- `void remove()`
  最後に返された要素を削除する

LinkedList クラスでは Iterator インタフェースを拡張した ListIterator を提供する。ListIterator は次のメソッドを持つ。

- `void add(Object o)`
  指定された要素をリストに追加する。

- `boolean hasNext()`
  リストを順方向にたどったとき、さらに要素がある場合に `true` を返す。

- `boolean hasPrevious()`
  リストを逆方向にたどったとき、さらに要素がある場合に `true` を返す。

- `Object next()`
  リスト内の次の要素を返す。

- `int nextIndex()`
  次に next を呼び出したときに返されることになる要素のインデックスを返す。

- `Object previous()`
  リスト内の直前の要素を返す。

- `int previousIndex()`
  次に previous を呼び出したときに返されることになる要素のインデックスを返す。

- `void` remove()
  next または previous によって返された最後の要素をリストから削除する。

- `void set(Object o)`
  next または previous によって返された最後の要素を、指定された要素に置換する。

演習 7

プログラムを作成する上で重要な役割を果たすものはデータ構造である。これまでのコンピュータサイエンスの歴史上、さまざまなデータ構造が考案されてきた。Java の API ではこのようなデータ構造をクラスライブラリとして提供している。リンクトリストとハッシュはよく使われるデータ構造の 1 つであり、これらをベースとした派生的なデータ構造のクラスも提供されている。

ここではこれらを利用した簡単なプログラミング課題をあげる。

1. Java API のドキュメントを参照しながら、LinkedList クラスを使用して、コマンドライン引数で与えられた文字列 (複数) をリストの要素として順に登録していき、最後にイテレータを使用して、1 行に 1 つずつ表示するプログラムを作れ。クラス名を `LinkedListTest` とする。
2. Java API のドキュメントを参照しながら、Hashtable クラスを次のように利用するプログラムを作れ。Hashtable をうまく利用すると文字列をインデックスとする配列のようなものを作ることができる。まず、コマンドライン引数で与えられた文字列をキーとし、その長さを値としてハッシュテーブルに登録することを文字列の数だけ繰り返す。次に各文字列をキーとして、ハッシュテーブルに登録されている値を取り出し、一行にその文字列の後ろにスペースで区切った 1 つのコロンを置き、得られた長さを置くように出力する。これを文字列の数だけ繰り返す。ソースプログラムのファイル名を `HashtableTest.java` とし、作成するクラス名を `HashtableTest` とせよ。
3. 前問では Hashtable クラスを利用したが、今度は HashMap を利用せよ。ソースプログラムのファイル名を `HashMapTest.java` とし、作成するクラス名を `HashMapTest` とせよ。
4. LinkedHashSet クラスを使う例題を次のように作成せよ。Set は数学の集合を模擬するクラスであり、リストクラスなどと違うのは同じ要素を重複して含まない点である。LinkedHashSet はその一実装である。詳細は Java API のドキュメントを参照せよ。なお、登録する要素は Object 型の値として扱われる。このため、同じ内容を持っていても別要素として扱われる可能性があり、オブジェクトが持っている内容によって同一かどうかを決定するためには工夫が必要となる。
   まず、LinkedHashSet に登録するオブジェクトのクラスを考えよう。このクラス名を Hoge とする。Hoge にはさまざまなフィールドがあっても良いが、ここでは String 型のフィールド name を持つとし、その内容が同じものは LinkedHashSet に二重に登録されないようにしたい。このためには、Hoge クラスで hashCode メソッドと equals メソッドを (Object クラスのものから) オーバーライドする必要がある。

hashCode メソッドは name.hashCode() の値を返す。equals メソッドは与えられたオブジェクトの持つ name の内容と自身が持つ name の内容を equals で比較した結果を返す。
さて、例題プログラムには、まず、クラス Hoge を宣言し、フィールドとして String 型の name を持たせる。
LinkedHashSetTest クラスは main メソッドを持ち、コマンドライン引数で与えられた文字列を次々と LinkedHashSet オブジェクトへ登録していき、登録が終ったらイテレータを使って、各要素を標準出力へと出力する。
この段階でコマンドライン引数から同じ文字列を 2 つ以上与えた場合、どのような結果になるか実行してみよ。
次に上述の hashCode と equals の両メソッドを Hoge に実装し、コマンドライン引数から同じ文字列を 2 つ以上与えた場合、どのような結果になるか実行してみよ。

11 解答例

演習 1

ArrayTest3d.java を図 15 に示す。

```
 1 class ArrayTest3d {
 2   public static void main(String args[]){
 3     int i, j;
 4
 5     int a[][] = new int[][] {
 6       new int[]{0},
 7       new int[]{0, 1},
 8       new int[]{0, 1, 2},
 9       new int[]{0, 1, 2, 3}
10     };
11
12     for(i=0; i<a.length; i++){
13       for(j=0; j<a[i].length-1; j++){
14         System.out.print(a[i][j] + ",");
15       }
16       System.out.println(a[i][j]);
17     }
18   }
19 }
```

図 15: ArrayTest3d.java

演習 2

1. 解答例を図 16 に記す。

```
 1 class atoi {
 2   public static void main(String args[]){
 3     if (args.length != 1){
 4       System.out.println(
          "put only one command
           line argument as a number");
 5     }
 6     else {
 7       System.out.println(
          Integer.parseInt(args[0]));
 8     }
 9   }
10 }
```

図 16: atoi.java

2. 解答例を図 17 に示す。

```
 1 class atoi2 {
 2   public static void main(String args[]){
 3     if (args.length == 0){
 4       System.out.println(
           "put only one command line argument
            as a number");
 5     }
 6     else {
 7       int k = 0;
 8       for(int i = 0; i<args.length; i++){
 9         k += Integer.parseInt(args[i]);
10       }
11       System.out.println(k);
12     }
13   }
14 }
```

図 17: atoi2.java

3. 解答例を図 18 に示す。

演習 3

1. 省略
2. 解答例を図 19 に記す。
3. 解答例を図 20 に記す。
4. 解答例を図 21 に記す。

```
 1 class atod {
 2   public static void main(String args[]){
 3     if (args.length != 1){
 4       System.out.println(
           "put only one command line argument
            as a number");
 5     }
 6     else {
 7       System.out.println(
           Double.parseDouble(args[0]));
 8     }
 9   }
10 }
```

図 18: atod.java

```
 1 class StringTest {
 2   public static void main(String[] args){
 3     for(int i=0; i < args.length; i++){
 4       System.out.println(args[i]);
 5     }
 6   }
 7 }
```

図 19: StringTest.java

```
 1 class StringTest2 {
 2   public static void main(String[] args){
 3     for(int i=0; i < args.length; i++){
 4       for(int j=0; j<args[i].length(); j++){
 5         System.out.println(args[i].charAt(j));
 6       }
 7       System.out.println();
 8     }
 9   }
10 }
```

図 20: StringTest2.java

演習 4

解答例を図 22 に示す。

演習 5

略

```
 1 class StringTest3 {
 2   public static void main(String[] args){
 3     int i, j;
 4
 5     i = Integer.parseInt(args[0]);
 6     j = Integer.parseInt(args[1])+1;
 7
 8     System.out.println(args[2].substring(i,j));
 9   }
10 }
```

図 21: StringTest3.java

```
 1 class Person {
 2   private int height, weight;
 3   private String name, address;
 4
 5   Person(int  height, int weight,
 6          String name, String address){
 7     this.height  = height;
 8     this.weight  = weight;
 9     this.name    = name;
10     this.address = address;
11   }
12
13   public String toString(){
14     return "name    = " + name + "\n" +
15            "height  = " + height + "\n" +
16            "weight  = " + weight + "\n" +
17       "address = " + address + "\n";
18   }
19 }
20
21 class PersonTest {
22   public static void main(String args[]){
23     Person p = new Person(170, 70,
                             "Nakai", "kasuga");
24     System.out.println(p);
25   }
26 }
```

図 22: PersonTest.java

演習 6

1. 解答例を図 23 に記す。
2. 解答例を図 24 に記す。

演習 7

1. 解答例を図 25 に示す。
2. 回答例を図 26 に記す。
3. 回答例を図 27 に記す。

```
 1 class ReverseString {
 2   public static void main(String args[]){
 3     StringBuffer sb = new StringBuffer();
 4
 5     for(int i = args[0].length()-1; i>=0; i--){
 6       sb.append(args[0].charAt(i));
 7     }
 8
 9     if (sb.toString().equals(args[0])){
10       System.out.println(args[0] +
                            " is a circular.");
11     } else {
12       System.out.println(args[0] +
                            " is not a circular.");
13     }
14   }
15 }
```

図 23: ReverseString.java

```
 1 class InsertNot {
 2   public static void main(String args[]){
 3     int i = args[0].indexOf(" is ")+4;
 4     StringBuffer sb =
        new StringBuffer(args[0].substring(0, i));
 5
 6     sb.append("not ");
 7     sb.append(args[0].substring(i));
 8
 9     System.out.println(sb);
10   }
11 }
```

図 24: InsertNot.java

4. hashCode, equals をオーバーライドしたものを図 28 に示す。

参考文献

[1] 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).

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


```
 1 import java.util.*;
 2
 3 class LinkedListTest {
 4   public static void main(String args[]){
 5     int i;
 6     LinkedList ll = new LinkedList();
 7     for(i=0; i<args.length; i++){
 8       ll.add(args[i]);
 9     }
10
11     Iterator it = ll.iterator();
12     while(it.hasNext()){
13       System.out.println((String)(it.next()));
14     }
15   }
16 }
```

図 25: LinkedListTest.java

```
 1 import java.util.*;
 2
 3 class HashtableTest {
 4   public static void main(String args[]){
 5     Hashtable ht = new Hashtable();
 6
 7     int i;
 8
 9     for(i=0; i<args.length; i++){
10       ht.put(args[i],
                new Integer(args[i].length()));
11     }
12
13     for(i=0; i<args.length; i++){
14       System.out.println(args[i] + " : " +
                    ((Integer)ht.get(args[i])));
15     }
16   }
17 }
```

図 26: HashtableTest.java

```
 1 import java.util.*;
 2
 3 class HashMapTest {
 4   public static void main(String args[]){
 5     HashMap hm = new HashMap();
 6
 7     int i;
 8
 9     for(i=0; i<args.length; i++){
10       hm.put(args[i],
               new Integer(args[i].length()));
11     }
12
13     for(i=0; i<args.length; i++){
14       System.out.println(args[i] + " : " +
                   ((Integer)hm.get(args[i])));
15     }
16   }
17 }
```

図 27: HashMapTest.java

```
 1 import java.util.*;
 2
 3 class Hoge {
 4   String name;
 5
 6   Hoge (String n){
 7     name = n;
 8   }
 9
10   public int hashCode(){
11     return name.hashCode();
12   }
13
14   public boolean equals(Object x){
15     boolean tmp = name.equals
                     (((Hoge)x).toString());
16
17     return tmp;
18   }
19
20   public String toString(){
21     return name;
22   }
23 }
24
25 class LinkedHashSetTest {
26
27   public static void main(String args[]){
28     int i;
29     LinkedHashSet lhs = new LinkedHashSet();
30
31     for(i=0; i<args.length; i++){
32       lhs.add(new Hoge(args[i]));
33     }
34
35     Iterator it = lhs.iterator();
36     while(it.hasNext()){
37       System.out.println(it.next());
38     }
39   }
40 }
```

図 28: LinkedHashSetTest.java