プログラミング言語各論
第3回講義ノート

2005年度担当: 中井央

2005年9月15日

1 パッケージ

パッケージとは1つのディレクトリに収められた関連するclassファイル群である。

Javaではコンパイルすると、クラス名.classというファイルが作成される。プログラムは規模が大きくなってくれば機能ごとにファイルをまとめたくなる。すでに開発されたプログラムは再利用したい。プログラムを多くの人で共有するには、そのプログラム (クラスファイル) が置かれている位置を指定する機能が必要となる。パッケージはこの機能を提供する。

1.1 パッケージの指定

これから書くソースプログラムがあるパッケージに属していることを示すにはそのソースプログラムの先頭に次のように記述する。

```
package パッケージ名;
```

パッケージ名はディレクトリ階層に対応している。aというディレクトリをパッケージとみなすならば、

```
package a;
```

のように記述する。

簡単な例を示そう。まず、パッケージaに対応するディレクトリを作成する。そして、ディレクトリaに移動し、プログラム (図1) を作成する。

```
uni% mkdir a
uni% cd a
```

コンパイルはディレクトリaの1つ上のディレクトリから次のように行う。

```
 1 package a;
 2
 3 class PackageDemo {
 4   public static void main(String[] args){
 5     System.out.println("Package Demo");
 6   }
 7 }
```

図 1: PackageDemo.java

```
uni% javac a/PackageDemo.java
```

実行するには、パッケージ名に続けてドットを打ち、mainメソッドを含むクラス名を指定する。

```
uni% java a.PackageDemo
```

演習 1

図1を入力し、コンパイル、実行せよ。

1.2 パッケージあるいはクラスの指定

上述したようにプログラムはパッケージ名をつけ、階層的に管理できる。Javaがインストールされると Javaに標準のクラス群はインストールされた状況にしたがって**あるディレクトリ**に格納される。ここではそれをパッケージルートディレクトリ、縮めてルートと呼ぶことにしよう。

標準で用意されているクラスを使う場合も必要とするクラスを見つけ出せるようにソースプログラムにその所在を記述する必要がある。

以前、LinkedListクラスを紹介した。それを使う際にはソースプログラムの先頭に次のように記述した。

```
import java.util.*;
```

これはルートから見て、java というパッケージ内の util というパッケージ内にあるすべてのクラスを以下で利用する可能性がある、という意味である。

パッケージは階層になっていて、パッケージの階層とディレクトリの階層は対応する。パッケージは階層をドット (.) で区切って表現する。ディレクトリは階層をスラッシュ(/) で区切って表現する[1]。

例えば、LinkedList を使うには本来的にはルートからのパッケージ階層をすべて記述して指定する必要がある。LinkedList l; のような変数の宣言は、本来は java.util.LinkedList l; のように記述しなければならない。しかし、これでは記述が煩雑になる。そこで import 文を使うことによりプログラム内では単にパッケージ名を取り除いたクラス名のみを使うことができるようになる。この際、必要となるクラス名は列挙する必要があるが、同じパッケージ内のものに限り、記号*を使ってまとめて指定することが可能である。すなわち、import java.util.*; という宣言は、java.util というパッケージ内のすべてのクラスを import 文で列挙したことに等しい。

ここで 1 つ注意が必要である。ファイルの場合、記号*を使えば、階層的に以下のディレクトリまでも指定したことになるが、パッケージの場合、*は階層的には適用されない。ディレクトリで言えば、同一ディレクトリ内にあるクラスファイルのみが指定の対象となる。そのディレクトリの中にさらにディレクトリがあってパッケージとなっている場合には別途、そのパッケージ (内のクラス) を指定する必要がある。

1.3 Java がパッケージを探す仕組み

さて、Java は導入された状態のまま利用していると、上述の Java がシステムとしてもっているパッケージのルートおよびカレントディレクトリからクラスファイルを探そうとする。

ところで、誰かが開発したライブラリもしくは自身が開発したライブラリを使用したい場合を考えよう。ここでは自分があるディレクトリ内でプログラムを開発し、パッケージにしてあるとしよう。そのプログラムを別のプログラム開発に使いたい場合、使いたいたびに開発しているディレクトリにコピーしてきて使用するのは手間がかかるし、ディスクも余計に必要になる。そこで、標準のクラス群とは別だが、どこかに設置し、いろいろな個所から使用したいクラス群 (パッケージ) があった場合、環境変数 CLASSPATH を使って指定する。

例えば、ユーザ nakai の下に nakaijava というディレクトリがあって他のユーザが使いたい場合、次のように設定する。

```
uni% setenv CLASSPATH /home/x/nakai/nakaijava
```

複数箇所を指定したい場合は、このテキストの最初に述べた環境変数 PATH と同様コロン:で区切って並べればよい。

1.4 名前空間

Java ではパッケージを用いることである考えのもとに設計されたクラス群をまとめることができるが、情報隠蔽の観点からたとえばパッケージの外からはアクセスができないようになっていた方がよいクラス (やそのメソッドなど) もある。

このようなアクセスを制御する機構が Java には備わっている。他のクラスからのアクセスの制御には次のものがある。

1. public： このアクセス修飾子がついていればどこからでもアクセスできる
2. protected： このアクセス修飾子がついている場合は次の 2 通りとなる
   (a) 同一パッケージ内からはアクセスできる
   (b) 他のパッケージ内のこのクラスのサブクラスからアクセスできる
3. デフォルト：すなわち特に指定がない場合、パッケージ内のすべてのクラスからアクセスができる
4. private： このクラスからだけアクセスができる

1.4.1 アーカイブについて

パッケージが階層化されて設計されている場合、実際に階層的にディレクトリを作成し、そこに開発したファイルを設置する必要がある。

これをディレクトリ階層を保ったまま移動させるにはアーカイブにすると便利である。アーカイブとはこ

[1] Windows 環境では \(日本語環境では￥マーク) が使われる。

こでは複数のファイルを 1 つにまとめることを言う。アーカイブには jar というコマンドを使用する。これは UNIX などで利用されているアーカイブのコマンド tar と非常によく似ている。ここではとりあえず、アーカイブの作り方について述べる。

上述のパッケージ a 以下のアーカイブを作るには a のあるディレクトリで次のように記述する (a を指定するので a の中ではないことに注意)。

```
uni% jar cvf a.jar a
```

これを実行すればアーカイブファイル a.jar ができる。なお、アーカイブの中身 (どのようなディレクトリ構成か) を知りたい場合は、次のようにする。

```
uni% jar tvf a.jar
```

また、アーカイブを展開したい場合は次のようにする。

```
uni% jar xvf a.jar
```

一連のファイル群からなるアプリケーションの場合、それらをアーカイブにしておいて、展開せず実行させるには次の手順を踏む。

1. wheretostart.txt というファイル (ファイル名は何でもよい) に次のような一行を入れて保存する。

   ```
   Main-Class: a.PackageDemo
   ```

   (a.PackageDemo の部分は main メソッドを含むクラス名)
2. jar cmf wheretostart.txt a.jar a を実行する。これでアーカイブが作成できる。
3. 実行する場合は、次のようにコマンドラインから打ち込む。

   ```
   uni% java -jar a.jar
   ```

演習 2

図 1 をここで紹介した方法により、実行可能なパッケージとして作れ。作成ができたら、ほんとうに実行可能かどうかチェックせよ。

1.5 パッケージ名とソースファイル名

Java の言語仕様ではパッケージ名はインターネットのドメイン名から作るべきだと述べている。例えば、図書館情報大学の中井研究室の場合、インターネットのドメイン名を逆順にしたものに最後に中井研を示す nakai をつけて次のような感じにする。

```
jp.ac.ulis.nakai
```

このようにすれば、世界中のさまざまな人が Java のプログラムを開発している状況で、同じ名前のパッケージになることをできるだけ防ぐことができる。

みんなが同様に myutil などと名前を付けると、誰の myutil なんだかわからなくなってしまう。少し大きなプログラムの開発をする場合、パッケージにまとめることは便利だし、それによっていろいろなところで使うこともできるようになる。その際、パッケージ名のつけ方には注意したい。

2 例外

Java における例外とは、実行時に何らかの問題が発生したときにそれを通知するために生成されるオブジェクトのことである。

この節では例外処理の方法についてまず述べ、プログラムを書く際、なんらかのエラー処理のために例外オブジェクトを発生させる方法について述べる。

2.1 例外処理

例外には Java が最初からシステムとしてもっているものとユーザが定義したものがある。最初は Java が定義している例外を使って例外処理を説明する。

まず、図 2 のプログラムを見てみよう。

図 2 のプログラムをコンパイルし、実行すると次のようなエラーメッセージが出力される (紙面の都合上折り返している)。

```
Exception in thread "main" java.lang.Arithmetic
Exception: / by zero at ExceptionTest.main
(ExceptionTest.java:5)
```

```
1 class ExceptionTest {
2   public static void main(String[] args){
3     int i=1, j=0, k;
4
5     k = i/j;
6
7     System.out.println(k);
8   }
9 }
```

図 2: ExceptionTest.java

このエラーメッセージはクラス ExceptionTest の main メソッドで 0 による除算をしたことを検知したというものである。通常、例外によるエラー処理を施していなければこのようにエラーメッセージを出力し、プログラムの実行を終了する。

例外を発生させるのは、期待しない動作 (もしくは入力など) を捕らえ、その処理を行うためである。この「捕らえる」という処理を図 3 で見てみよう。

```
 1 class ExceptionTest2 {
 2   public static void main(String[] args){
 3     int i=1, j=0, k;
 4
 5     try {
 6       k = i/j;
 7     }
 8     catch (Exception e){
 9       System.out.println("can't divide by 0");
10     }
11   }
12 }
```

図 3: ExceptionTest2.java

図 3 では割り算の実行を try 節で囲んでいる。それに catch 節が続いている。節というのはここでは { と } で囲んだブロックのことであると思ってよい。

一般的には 1 つの try 節に対し、1 つ以上の catch 節が続くか 1 つ以上の finally 節が続く。catch 節がある場合、finally 節はなくてもよい。catch 節がなく、finally 節のみが存在する場合もある。

図 3 では、例外が発生するかもしれない文を try 節とすることで、例外に対応している。catch 節ではキーワード catch の後ろに受け取る例外の種類を書く。

Java ではクラス Exception が定義されている。実際にユーザが定義する例外はこのクラスを継承して作る。実際的な場面では、ユーザは例外を定義し、図 3 で書いたように (Exception e) のようにはせず、その時点での具体的な例外クラスを記述する。

finally 節については後述する。

2.2 例外の検索

例外が起こると、まず、その例外が発生した文が try 文で囲われているかチェックされ、そうでない場合、その文を含んだメソッドを呼び出した位置で、try 文で囲われているかどうかチェックする。そのような try 文が見つかるまで呼び出し元の方へ順にスタックをたどっていく。

第 1 回目のテキストで述べた実行時環境について思い出そう。プログラムが実行されている間、関数呼び出しは呼出しごとにスタック上にメモリ領域を確保されているのであった。例外が発生した際は、その例外を**キャッチ**できる try 文があるところまでスタックを遡って調べていく。

例えば図 4 を見てみよう。

```
 1 class ExceptionTest3 {
 2   public static void div(){
 3     int i=1, j=0, k;
 4
 5     k = i/j;
 6   }
 7
 8   public static void main(String[] args){
 9     try {
10       div();
11     }
12     catch (Exception e){
13       System.out.println("can't divide by 0");
14     }
15   }
16 }
```

図 4: ExceptionTest3.java

このプログラムでは 0 による除算を行うメソッド div が定義されていて、div を呼び出す際に try 節を使っている。すなわち、div の中で例外が発生してもそれを捉えるのは main メソッドの中 (12〜14 行目) である。

2.3 finally 節

finally 節は例外が起きても起きなくても、例外がキャッチされてもされなくても実行される。finally 節は実行の後始末のために利用できる。try 節中に書いた一連の文を実行できたら、そのまま finally 節に実行が移る。try 節中に書いた一連の文の途中で例外が発生したら、そこから上述のように実行時スタックをたどって catch 節を探していく。

2.4 例外を使用したプログラムの例

ここで [1] の邦訳の p.231 にある例題プログラム (図 5) を紹介する。なお、紙面内に収まるように 11, 14, 17 行目は修正している。

```
 1 class Divider {
 2   public static void main(String args[]){
 3     try {
 4       System.out.println("Before Division");
 5       int i = Integer.parseInt(args[0]);
 6       int j = Integer.parseInt(args[1]);
 7       System.out.println(i/j);
 8       System.out.println("After Division");
 9     }
10     catch (ArithmeticException e){
11       System.out.println("Arithmetic"+
                            "Exception");
12     }
13     catch (ArrayIndexOutOfBoundsException e){
14       System.out.println("ArrayIndexOutOf"+
                            "BoundsException");
15     }
16     catch (NumberFormatException e){
17       System.out.println("NumberFormat"+
                            "Exception");
18     }
19     finally {
20       System.out.println("Finally Block");
21     }
22   }
23 }
```

図 5: Divider.java

このプログラムではコマンドラインから 2 つの数字を引数として受け取ることを想定している。このため、引数が 1 つもしくは 0 だった場合、引数が 2 つであっても数字でなかった場合には例外が発生する。また、内部では除算を行うので、第 2 引数が 0 の場合にも例外が発生する。

この例からわかることは、複数の異なる種類の例外をキャッチすることができることである。すなわち、try 節内に記述した文に対して発生しうる例外に関して catch 節を設けるべきである。

さて、このプログラムを動かした結果を示そう。この例では 3 つの例外をキャッチしようとしているから、それらが発生するような入力と正常な入力の 4 通りを試してみる必要がある。また、この例では引数のとり方によってさらに場合わけしてみる必要がある。このため、以下のように 6 通りを考える必要がある。このようなテスト項目は一般にプログラム開発の際、プログラマが考慮しなければならないことである。

```
1 [nakai@uni]% java Divider 4 2
  Before Division
  2
  After Division
  Finally Block

2 [nakai@uni]% java Divider 4 0
  Before Division
  ArithmeticException
  Finally Block

3 [nakai@uni]% java Divider 4
  Before Division
  ArrayIndexOutOfBoundsException
  Finally Block

4 [nakai@uni]% java Divider
  Before Division
  ArrayIndexOutOfBoundsException
  Finally Block

5 [nakai@uni]% java Divider 4 2 3
  Before Division
  2
  After Division
  Finally Block

6 [nakai@uni]% java Divider a 4
  Before Division
  NumberFormatException
  Finally Block
```

図 6: Divider.java を実行した結果

実行結果を図 6 に示す。各々簡単に説明しよう。説明のため、各入力ごとに番号を付けておいた。

1. 正常な入力に対する結果である。try 節の最初と最後に println 文があり、どちらも出力されている。また、finally 節の println 文でも出力が行

われている。

2. 0 による除算が発生するような入力に対する結果である。try 節の最初の println 文が実行され、i/j が実行された段階で例外が発生しているため、try 節の最後の println 文は実行されず、finally 節の println 文が実行されている。
3. コマンドライン引数を 1 つにした場合の結果である。main の引数である String 型の配列 args はモノが 1 つしか入っていない。つまり、args[0] には文字 (列)4 が入っているが args[1] には何も入っていないため、args の長さは 1 である。それにもかかわらず、図 5 の 6 行目で、args[1] にアクセスしようとしているため、例外 ArrayIndexOutOfBoundsException が発生している。
4. 上記と同様である。今度はコマンドライン引数を 1 つも与えていないので、図 5 の 5 行目の段階で例外が発生している。
5. コマンドライン引数として余計なもの (3) も与えているが、プログラム中では args[2] にはアクセスしないため、正常な実行となっている。
6. コマンドラインから与えるのは数字でなければならないのだが、ここでは a という文字を与えてみた。クラス Integer は、引数として数字を文字列として受け取り、数値に変換する parse というメソッドを持っているが、このメソッドは入力が数字でない場合、NumberFormatException という例外を発生させる。この例では図 5 の 5 行目でその例外が発生している。

演習 3

1. 図 3 を入力し、実行せよ。
2. 図 4 を入力し、実行せよ。
3. 図 5 を入力し、実行して、上記を確認せよ。

2.5 例外の発生のさせ方

これまで見た例では、システムで定義された例外が発生していた。例えば、0 による除算をシステムが検知し、システムが実行時にその例外を発生させていた。try-catch 文を使うことで、例外を捉え、処理することができた。

システムで用意している例外クラスを継承し、独自の例外クラスを作成することもできる。また、既存の例外クラスのオブジェクトを作成し、必要に応じて例外を発生させることもできる。ここではこれらのことについて述べる。

2.5.1 例外クラスの種類

Object クラスを継承した Throwable と言うクラスがある。例外オブジェクトは作成されたあと、throw 文によって「投げる」ことで「例外の発生」となる。catch 節は投げられたものの「受け手」であり、適切な受け口 (catch のあとの引数部分) により受け取られる。

例外を投げるには次のようにプログラム中に記述する。

```
throw new MyException();
```

MyException の部分には自分で定義したクラスのコンストラクタもしくはシステムで用意しているクラスのコンストラクタを記述する。

例外クラスのクラス階層は次のようになっている。

1. Object クラスを継承した Throwable クラス

2. Throwable クラスを継承した Exception クラス

Throwable クラスのコンストラクタには次の 2 つがある[2]。

Throwable()

Throwable(String m)

すなわち、引数なしか引数として文字列を受け取るものかである。後者は例外処理時に「説明」をつけるための文字列を与えるものである。この文字列を取り出すための String getMessage() というメソッドも用意されている。

システムが用意する例外クラスは Exception クラスを継承したものである。この中には図 5 で見た算術演算を評価した際に発生させられる ArithmeticException

[2] JDK1.4 からは新機能が追加されているがここでは扱わない。

や、配列に対するアクセスが実行される時、配列の添え字として配列のサイズを超えた値が使用された場合に発生させられる ArrayIndexOutOfBoundsException などがある。

2.5.2 例外を投げる

あるメソッドにおいて例外を投げる可能性があるならば、そのメソッドの宣言において、その旨を示す必要がある。具体的な記述方法の例を示そう。何も返さず、引数もないメソッド a が MyException という例外を投げる場合、次のように記述する。

```
void a() throws MyException {
  ...
}
```

すなわち、そのメソッドでは内部で例外を投げることを宣言する。

2.6 例外のまとめ

プログラムを記述するとき、特に重要なことはエラーをどう処理するかということである。個人レベルでプログラムを書いている場合は、プログラマ＝ユーザであり、入力として正当なものが何かを知っているため、入力として誤ったものが与えられることもほとんどない。そのため、エラーチェックのコードを省いてしまうことも多々ある。

しかし、真に実用的なプログラムを記述するには、エラー処理にこそ力を注ぐ必要がある。例えば、階乗計算を行うプログラムを記述するメソッドを考えよう。

```
int fact(int n){
  if (n == 1){
    return 1;
  }
  else {
    return n * fact(n - 1);
  }
}
```

上記のプログラムでは 0 以下の入力があった場合、プログラムは無限の再帰呼び出しをしようとして、システムに止められる。

チェックをどのようにすべきかということはプログラミング言語とは関係なく、プログラマが考慮すべき問題である。

Java ではエラーの処理を行うための仕組みとして、「例外」とその処理構文 (try-catch 文) が用意されている。

図 7 に示すのは階乗計算をするメソッドを含んだ例外の例題プログラムである。

```
 1 class UnderZero extends Exception {
 2   UnderZero(String m){
 3     super(m);
 4   }
 5 }
 6
 7 class ExceptionTest4 {
 8   public static int f(int n) throws UnderZero {
 9     if (n < 1){
10       throw new UnderZero("Use more than 0!");
11     }
12     if (n == 1){
13       return 1;
14     }
15     else {
16       return n * f(n - 1);
17     }
18   }
19
20   public static void main(String[] args){
21     int i = Integer.parseInt(args[0]);
22
23     try {
24       System.out.println(f(i));
25     }
26     catch (UnderZero uz){
27       System.out.println(uz.getMessage());
28     }
29   }
30 }
```

図 7: ExceptionTest4.java

このプログラムを簡単に説明しよう。1〜5 行目では階乗計算に対し、マイナスの値が与えられた場合に発生させる例外クラスの宣言である。8〜18 行目で階乗計算をするメソッドを宣言しており、9〜11 行目で引数の値をチェックし、0 以下であれば例外を投げる。

この例では main メソッドで f を呼び出したところまで実行時スタック上を引き戻し、例外処理が行われる。C 言語などで同じように最初に f を呼び出したところまで戻ってエラー処理を行うようにするにはどのようなコードを書けばよいであろうか。このことについてはみなさんに考えてみてもらいたい。

演習 4

1. 図 7 を入力し、コンパイル、実行せよ。例外が発生するような入力を与えること。
2. 次を満たすプログラムを作れ。ソースファイル名を ExceptionTest5.java とし、main メソッドを含むクラス名を ExceptionTest5 とする。
   (a) 新しく例外クラス MyException を定義する。(特に中身はなくてもよい)
   (b) MyException を拡張して、MyException0 から MyException9 までの 10 個のクラスを作る。
   (c) ExceptionTest5 クラスに static メソッド ex を作成する。このクラスは MyException 例外を投げる。引数は 1 つの整数型の値を受け取る。中ではその値にしたがって値が 0 なら MyException0 を、1 なら MyException1 を、…、9 なら MyException9 を投げるとする。値がそれ以外の場合、MyException を投げるとする。

      場合わけには if 文を使用してもよいが、図 8 に示す switch 文を使うと記述がきれいになる。main メソッドでは try-catch 文で ex の呼び出しをし、例外を受け取る準備をする。なお、引数がない場合、引数が数字ではない場合にも例外が発生するのでそれらも受け取り処理をするようにしなければならない。

```
switch (変数){
  case 0:
    変数の値が 0 だった場合の処理
  case 1:
    変数の値が 1 だった場合の処理
  ...
  case 9:
    変数の値が 9 だった場合の処理
  default:
    上記以外の処理
}
```

図 8: switch 文

3 Java による入出力

Java による入出力は面倒くさいといわれる。また、オブジェクトとして入出力 (の流れ) を扱うにはこのようにクラスを構成しておかなければならず、その点からすればこの構成こそがよいのだという意見もある。いずれにせよ、プログラムを作る際、入出力のないプログラムは役に立たないのだから、それを扱っていかなければならない。ここでは入出力について述べていく。

3.1 Java による入出力を学ぶ前に

UNIX などの OS ではパイプライン (略してパイプ) という考え方がある。2 つまたはそれ以上のコマンドを組み合わせてあたかも 1 つのコマンドのように使うことである。

例を示そう。UNIX 上には w というコマンドがある。実行例を図 9 に示す。

この中からユーザ名の一覧だけを取り出したいとしよう。awk というコマンドを使うとスペースで区切られた塊を 1 つのフィールドとみなしてそのうちの 1 つを表示することができる。次に示すのは各行の最初のフィールドをプリントする awk の例である。

```
awk '{ print $1 }'
```

いま、w の結果が w_res というファイルに入っているとすると以下のように実行することでユーザ名のフィールドだけを取り出すことができる。

```
awk '{ print $1}' w_res
```

画面への出力をファイルに入れるにはリダイレクト機能を使う。C シェルの場合、>という記号により出力をファイルに入れることができる。w コマンドの結果をファイルに格納するには次のようにする。

```
uni% w > w_res
```

w の結果をファイルに入れて、そのあと awk を実行するというのはわずらわしいので、パイプでつないでやることで一時的なファイルを作らず、目的の結果を得ることができる。

```
[nakai@uni]% w
  4:57pm  up 13 days,  8:18,  21 users,  load average: 2.27, 1.97, 1.87
User     tty           login@  idle   JCPU   PCPU  what
momo123  pts/ta        3:08pm                      emcws sample-7.xsl
momo260  pts/1         3:14pm    51      7      7  emcws rep5.xml
kuri279  pts/2         4:43pm    12              telnet clover
momo157  pts/3         4:49pm     6      1      1  emcws report.xsl
nakai    pts/tb        7:03pm284:43               emcws ExceptionTest.java
momo260  dtremote      3:13pm 45:46               -
momo123  dtremote      6:02pm 45:46               -
momo181  pts/4         5:00pm143:56               emcws sample-4.xsl
momo103  pts/tc        4:55pm     2               -csh
momo243  pts/td        3:10pm                     emcws sample-4.xsl
momo157  dtremote      4:49pm 45:46               -
kuri279  dtremote      4:42pm 45:46               -
momo181  dtremote      4:59pm 45:46               -
momo192  pts/te        3:51pm                     emcws sample-3.xml
momo225  pts/tf        4:49pm                     emcws rep5.xml
nakai    pts/9         2:55pm            2      2  w
nakai    pts/tn        5:48pm 45:06     12     12  tcsh
satoru   pts/17        5:08pm311:01               tcsh
kaki208  pts/29       10:31am 77:30               -
osaka    pts/32        4:02pm 95:31 111:02 111:02  top -s 10
kuri126  pts/10        3:34pm                     emcws rdpa6.c
```

図 9: w の実行結果例

```
uni% w | awk '{ print $1 }'
```

UNIX では標準入力、標準出力、標準エラー出力という考え方がある。これは UNIX 以外の OS でも採用されている場合が多く、Java のような言語処理系でもこの考え方に基づいて入出力が考慮されている。

標準入力とは、標準の入力元である。こういうとわかりにくいと思うが、一般に何も指定しない場合、キーボードが割り当てられる。cat というコマンドは 1 つ以上のファイル名を受け取り、それらを連結した結果を標準出力に出力する。ファイル名を指定しなかった場合はファイルの代わりに標準入力からの入力を受け取る。すわなち、単にコマンドラインから cat と打つと入力待ちの状態となる。

パイプは標準出力と標準入力をつなぐ。

標準出力は標準の出力先である。これもわかりにくいが一般に何も指定しない場合、端末の画面が割り当てられる。

さて上で述べた `w | awk '{ print $1}'` について考えてみよう。`awk '{ print $1 }'` だけを実行するとキーボードからの入力待ちとなる。パイプでつなぐことで w の出力が awk の入力になる。

出力には標準出力と標準エラー出力の 2 つがある。これは通常の結果の出力とエラーメッセージの出力を系統として分けるために存在する。何もしなければどちらも同じ画面上に出力される。通常 C シェルでは、パイプには記号|を使用するが、標準エラー出力をパイプに送る場合は|&を使う。通常 C シェルでは、リダイレクトは>を使用するが、標準エラー出力をリダイレクトする場合は>&を使う。

ここまでのまとめ

パイプラインという考え方を知ることで、単純なコマンドを組み合わせて複雑な処理を行うことができるようになる。w の結果からユーザ ID だけを awk で取り出し、それをソートして、重なっているものを 1 つだけにするには次のようにする。

```
uni% w | awk '{ print $1}' | sort | uniq
```

これは 4 つのコマンドをつないだものである。それぞれのコマンドがもつ機能を組み合わせてあたかも 1 つのコマンドのように扱えるということに着目してほしい。

また、入力と出力については標準入力、標準出力、標準エラー出力という考え方があることを覚えておい

てほしい。もちろん、通常のファイルを入力、出力とする場合もある。

3.2 まずは簡単な例から

とりあえず、ファイル名を受け取って、書き込む例、読み込む例を示そう。これらのプログラムは [1] の日本語版の 11 章からのものである。図 10 に書き込みのサンプルプログラムを、図 11 に読み込みのサンプルプログラムを示す。

```
 1 import java.io.*;
 2
 3 class FileWriterDemo {
 4   public static void main(String args[]){
 5     try {
 6       FileWriter fw = new FileWriter(args[0]);
 7
 8       for (int i = 0; i < 12; i++){
 9         fw.write("Line "+i+"\n");
10       }
11
12       fw.close();
13     }
14     catch (Exception e){
15       System.out.println("Exception: "+e);
16     }
17   }
18 }
```

図 10: FileWriterDemo.java

まず、図 10 を見てみよう。このプログラムは書き込み用のファイル名を 1 つ受け取り、そこに 12 行書き込むというものである。FileWriter というクラスは文字をファイルに書き出すためのクラスである。

次に図 11 を見てみよう。こちらはコマンドライン引数の最初に与えられた名前のファイルを開き、その内容を読み込んで標準出力に出力するプログラムである。

演習 5

1. java のドキュメントで FileWriter の write メソッドについて調べよ。
2. java のドキュメントで FileReader の read メソッドについて調べよ。
3. 図 10 と図 11 を参考にして第 1 引数で指定されたファイルの内容を第 2 引数で指定されたファイルへコピーするプログラムを作成せよ。解答例は図 19 に示す。

```
 1 import java.io.*;
 2
 3 class FileReaderDemo {
 4   public static void main(String args[]){
 5     try {
 6       FileReader fr = new FileReader(args[0]);
 7
 8       int i;
 9       while((i = fr.read()) != -1){
10         System.out.print((char)i);
11       }
12
13       fr.close();
14     }
15     catch (Exception e){
16       System.out.println("Exception: " + e);
17     }
18   }
19 }
```

図 11: FileReaderDemo.java

3.3 バッファつき文字ストリーム

ストリーム (stream) とは「流れ」のことであるが、データの流れを意味する抽象的な概念としてここでは使用する。ファイルとのやり取りをする際、一旦バッファと呼ばれる領域に読み込みもしくは書き込みのデータをためることで、物理的なデバイスへのアクセス回数を減らすことができれば効率が良くなる。これは例えるならば、何らかの理由で台所の水道の蛇口から一滴の水を得て、風呂に入れることを繰り返して満杯にすることよりも、バケツにある程度貯めて、風呂場へ持っていく方が効率が良いというのと同様である。

この目的でバッファつきの文字ストリームクラスが用意されている。

この例を図 12 と図 13 に示す (プログラムは [1] の日本語版より)。図 10 と図 11 とそれぞれ比較してみよう。FileReader もしくは FileWriter のオブジェクトにかぶせるように BufferedReader や BufferedWriter を設置している。それ以外はほぼ同じになっている。なお、BufferedWriter では flush() というメソッドが用意されている。これはバッファに貯められた情報を出力先に吐き出す役目をする。場合によっては明示

```
 1 import java.io.*;
 2
 3 class BufferedWriterDemo {
 4   public static void main(String args[]){
 5     try {
 6       FileWriter fw = new FileWriter(args[0]);
 7
 8       BufferedWriter bw =
                          new BufferedWriter(fw);
 9
10       for (int i=0; i<12; i++){
11         bw.write("Line " + i + "\n");
12       }
13
14       bw.close();
15     }
16     catch (Exception e){
17       System.out.println("Exception: "+e);
18     }
19   }
20 }
```

図 12: BufferedWriterDemo.java

```
 1 import java.io.*;
 2
 3 class BufferedReaderDemo {
 4   public static void main(String args[]){
 5     try {
 6       FileReader fr = new FileReader(args[0]);
 7
 8       BufferedReader br =
                            new BufferedReader(fr);
 9
10       String s;
11       while((s = br.readLine()) != null){
12         System.out.println(s);
13       }
14
15       fr.close();
16     }
17     catch (Exception e){
18       System.out.println("Exception: " + e);
19     }
20   }
21 }
```

図 13: BufferedReaderDemo.java

的にこのメソッドを呼び出さないと書き込みが完了しない場合がある。

演習 6

図 12 と図 13 を参考にしてコマンドラインの第 1 引数に受け取ったファイルの中身をコマンドラインの第 2 引数に受け取ったファイルへ書き出すプログラムを作成せよ。ただし、書き出しの際、行番号を付し、行番号の領域は常に半角 5 文字分とることとし、行番号と実際の行との間は半角 1 つ分空けるとする。

解答例を図 20 に記す。

3.4 バイトストリーム

これまではテキストデータを扱う例を記してきた。ここではバイナリデータを扱う例を示そう。

基本型のところで整数型 int は 32 ビット符号付きの 2 の補数表現であることを述べた。整数を人間が読めるように文字列に変換すればテキストデータとなるが、メモリ内部で保持しているデータ形式 (ビット列) のまま外部に保存したい場合もある。このようなときバイトストリームを利用する。また、画像や音声などのデータはバイナリ形式で保存されている。

バイトストリームを扱うには InputStream, OutputStream クラスを利用する。ここではファイルに対して入出力を行うためにこれらのサブクラスである FileInputstream, FileOutputStream を使用した例をあげる ([1] の日本語版より)。図 14 が出力用のプログラムであり、図 15 が入力用のプログラムである。

まず、図 14 について説明しよう。6, 7 行目でコマンドライン引数で指定したファイルを開いている。言い替えると指定したファイルに対応するバイトストリームオブジェクトを作成している。このオブジェクトに対して行える操作の 1 つに `void write(int b)` がある。これは一応整数として値を受け取るが実際には下位 1 バイト (8 ビット) 分だけがストリームに対して出力される。この例では 0〜11 までの値を書き込んでいる。

次に図 15 について説明する。6, 7 行目はファイル名を引数とした入力ストリームオブジェクトの作成である。このオブジェクトには `int read()` というメソッドがある。これはストリームから 1 バイトのデータを読み込む。読み込んだ値は int として扱われる。ファイルの終わりに達した場合は −1 を返す。10〜12 行目では `read` により 1 バイトずつ読み、ファイルの終わりに達していなければその値を画面に出力する。

```
 1 import java.io.*;
 2
 3 class FileOutputStreamDemo {
 4   public static void main(String args[]){
 5     try {
 6       FileOutputStream fos =
 7         new FileOutputStream(args[0]);
 8
 9       for(int i = 0; i<12; i++){
10         fos.write(i);
11       }
12
13       fos.close();
14     }
15     catch (Exception e){
16       System.out.println("Exception: "+e);
17     }
18   }
19 }
```

図 14: FileOutputStreamDemo.java

```
 1 import java.io.*;
 2
 3 class FileInputStreamDemo {
 4   public static void main(String args[]){
 5     try {
 6       FileInputStream fis =
 7         new FileInputStream(args[0]);
 8
 9       int i;
10       while((i = fis.read()) != -1){
11         System.out.println(i);
12       }
13
14       fis.close();
15     }
16     catch (Exception e){
17       System.out.println("Exception: "+e);
18     }
19   }
20 }
```

図 15: FileInputStreamDemo.java

演習 7

1. まず、図 14 の 9 行目を次のように変更する。

   ```
   for(int i=48; i<=57; i++){
   ```

   そして、プログラムを実行し、書き込まれた結果を cat などで見てみよ。
2. 書き込まれた結果を図 15 のプログラムに与えてみよ。

3.5 標準入力からの入力

標準入力から文字データを受け取る方法を記す。サンプルプログラムは図 16 のようになる。このプログラムは図 13 とほとんど同じである。違うところは 6, 7 行目では InputStreamReader クラスのオブジェクトを作成していて、コンストラクタで標準入力を表す System.in を渡しているところだけである (細かい変数名の違いなどを除く)。

入力に関して注意することはエンコーディング (文字コード) の問題である。詳細はここでは省略するが、InputStreamReader はコンストラクタでエンコーディングの指定をすることができる。指定がなければその環境のエンコーディングを使用する。

uni 上の場合、X 端末からログインした場合は通常、エンコーディングとして EUC-JP となっており、Windows 環境では Shift_JIS となっている。uni へ telnet などでログインした場合は ISO8859_1 となっている。telnet などの環境でログインした際に X 端末でログインしたときと同じにしたければ次のように実行すればよい。

```
uni% setenv LANG ja_JP.eucJP
```

なお、現在のエンコーディングが何になっているかを判定するプログラムを図 17 に示す。

3.6 Java の入出力のまとめ

この節では Java による入出力について紹介した。入出力の対象は大きくテキストとバイナリに分けることができる。入出力に関連するクラスは 60 種類以上あると言われている。

コンピュータ上のデータは基本的にはバイトの列である。そのうちの一部のものを人間が見てわかるもの＝文字として扱っていて、そのようなデータ (バイト) の列をテキストと呼んでいるわけである。データの入出力の基本はバイト列であるという考え方からそれを扱うクラスとして InputStream が用意されている。バイトの列にも種類があり、ファイルから扱う、オブジェクトを (メモリ内部の形式のまま) 扱う、など、それぞ

```
 1 import java.io.*;
 2
 3 class ReadConsole {
 4   public static void main(String args[]){
 5     try {
 6       InputStreamReader isr =
 7         new InputStreamReader(System.in);
 8
 9       BufferedReader br =
           new BufferedReader(isr);
10
11       String s;
12       while((s = br.readLine()) != null){
13         System.out.println(s.length());
14       }
15
16       isr.close();
17     }
18     catch (Exception e){
19       System.out.println("Exception: "+e);
20     }
21   }
22 }
```

図 16: ReadConsole.java

```
 1 import java.io.*;
 2
 3 class OSTest {
 4   public static void main(String args[]){
 5     OutputStreamWriter osw =
         new OutputStreamWriter(System.out);
 6
 7     System.out.println(osw.getEncoding());
 8   }
 9 }
```

図 17: OSTest.java

れの用途に対して、InputStream クラスのサブクラスが用意されている。出力についても同様である。

ところで、我々人間はテキスト情報を扱いたい。この目的のために Reader クラスと Writer クラスが設けられている。バイトを扱う InputStream (とそのサブクラス) のオブジェクトを受け取り、内部で処理を施すことでテキストとして扱う InputStreamReader やその出力版である OutputStreamReader が用意されている。

また、読み書きの際、バッファにまとめて読み書きすることで効率を上げるという目的で BufferedReader や BufferedWriter が設けられている。

これらの使い方を振り返ると基本となる仕事をこなすオブジェクトに更なる機能を持つオブジェクトをかぶせた形になっている。具体的なコードの形で言えば、基本となるオブジェクトを引数として更なる機能を持つオブジェクトのコンストラクタが呼ばれている。

これらは冒頭で述べた複数の単機能を組み合わせることで求める機能を得ることと似ている。

ここではよく使いそうな入出力パターンの例を示した。上述のように Java では数多くの入出力のクラスが用意されている。ここで述べた基本的なパターンをベースにし、各自の必要に応じて Java のドキュメントを参照してほしい。

4 演習問題の解答例

演習 1

省略

演習 2

省略

演習 3

省略

演習 4

1. 省略
2. 図 18 に解答例を示す。

演習 5

図 19 を参照のこと。

演習 6

図 20 を参照のこと。

```
 1 class MyException extends Exception {
 2 }
 3
 4 class MyException0 extends MyException {
 5 }
 6
 7 class MyException1 extends MyException {
 8 }
 9
10 class MyException2 extends MyException {
11 }
12
13 class MyException3 extends MyException {
14 }
15
16 class MyException4 extends MyException {
17 }
18
19 class MyException5 extends MyException {
20 }
21
22 class MyException6 extends MyException {
23 }
24
25 class MyException7 extends MyException {
26 }
27
28 class MyException8 extends MyException {
29 }
30
31 class MyException9 extends MyException {
32 }
33
34 class ExceptionTest5 {
35   public static void ex(int i)
           throws MyException {
36     switch (i){
37     case 0:
38       throw new MyException0();
39     case 1:
40       throw new MyException1();
41     case 2:
42       throw new MyException2();
43     case 3:
44       throw new MyException3();
45     case 4:
46       throw new MyException4();
47     case 5:
48       throw new MyException5();
49     case 6:
50       throw new MyException6();
51     case 7:
52       throw new MyException7();
53     case 8:
54       throw new MyException8();
55     case 9:
56       throw new MyException9();
57     default:
58       throw new MyException();
59     }
60   }
61
62   public static void main(String args[]){
63     try {
64       ex(Integer.parseInt(args[0]));
65     }
66     catch (MyException e){
67       System.out.println(e);
68     }
69     catch (NumberFormatException e){
70       System.out.println(
           "please put a number as argument");
71     }
72     catch (ArrayIndexOutOfBoundsException e){
73       System.out.println(
           "please put a number as argument");
74     }
75   }
76 }
```

図 18: ExceptionTest5.java

演習 7

参考文献

[1] 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).

```
 1 import java.io.*;
 2
 3 class BufferedCopy {
 4   public static void main(String args[]){
 5     if (args.length != 2){
 6       System.out.println(
          "Usage: java Buffered copy src dest");
 7       System.exit(1);
 8     }
 9
10     try {
11       FileReader fr = new FileReader(args[0]);
12       BufferedReader br = new BufferedReader(fr);
13
14       FileWriter fw = new FileWriter(args[1]);
15       BufferedWriter bw = new BufferedWriter(fw);
16
17       String s;
18       int i = 1;
19       String tmp;
20       int j, k;
21
22       while((s = br.readLine()) != null){
23         tmp = Integer.toString(i);
24         j = tmp.length();
25         for(k=0; k<5-j; k++){
26           bw.write(' ');
27         }
28         bw.write(tmp, 0, j);
29         bw.write(' ');
30         bw.write(s, 0, s.length());
31         i++;
32         bw.newLine();
33         bw.flush();
34       }
35
36       fr.close();
37       fw.close();
38     }
39     catch (Exception e){
40       System.out.println("Exception: " + e);
41     }
42   }
43 }
```

図 20: BufferedCopy.java

```
 1 import java.io.*;
 2
 3 class FileCopy {
 4   public static void main(String args[]){
 5     if (args.length != 2){
 6       System.out.println(
           "Usage: java FileCopy src dest");
 7       System.exit(1);
 8     }
 9
10     try {
11       FileReader fr = new FileReader(args[0]);
12       FileWriter fw = new FileWriter(args[1]);
13
14       int i;
15       while((i = fr.read()) != -1){
16         fw.write(i);
17       }
18
19       fr.close();
20       fw.close();
21     }
22     catch (Exception e){
23       System.out.println("Exception: " + e);
24     }
25   }
26 }
```

図 19: FileCopy.java