情報処理技法 (Javaプログラミング)2

第2回
前期の復習(2)

人間科学科コミュニケーション専攻
白銀 純子

第2回の内容

- 情報処理技法(Javaプログラミング)1の復習(続き)

前回の出席課題の解答

- Javaプログラムを実行させる仕組みを、下記のキーワードを使って説明しなさい。
  - キーワード: ソースコード, Javaバイトコード, Javaコンパイラ, JVM

解答例:
人間がソースコードを書き、それをJavaコンパイラに入力する。そうすると、Javaコンパイラはソースコードを翻訳したJavaバイトコードを作成する。人間がJavaバイトコードをJVMに入力すると、JVMはプログラムを1行ずつ命令を機械語に通訳してコンピュータに伝え、コンピュータがその命令を実行する。

例外処理

プログラムで発生するエラー

- プログラムの実行時に発生する可能性のあるエラー
  - 配列で、利用可能な範囲外の添え字を使おうとしたとき
  - String型の値をint型に変換できないとき
    - Ex. 入力された文字列をint型に変換したいときに、「abc」という文字列が入力される、など
  - 入力しようとしたファイルが存在しないとき
  - 数を0で割ろうとしたとき
  - etc.

プログラムを実行してみなければ、発生するかどうかわからないエラー
＝コンパイル時には発見できないエラー

「例外」と呼ぶ

例外に対処するには?

- 例外が発生すると...
  - プログラムの実行がその時点で終了してしまう
- 例外を発生させないためには...?
  1. 例外が発生しないよう、プログラムを書いておく
     - 完全には難しい(入力データなどは実行時でないと判断不可)
  2. 例外に対処するための処理をプログラムに書いておく
     - 例外が発生しても、それなりの処理を行う

例外処理

例外処理の書き方(基本形)

```
try {
    例外が発生する可能性のある処理
}
catch (例外の種類を表すクラス名  変数名) {
    例外が発生したときに行う処理
}
```

- tryを書いたら、必ずcatchも書かなければならない
- try文の中にcatchを書いてはならない
- tryの「}」の後、catchの前には何も書いてはならない

try～catch

- try
  - 例外が発生する可能性のある処理を、「try{～}」の間に書く
- catch
  - tryの中の処理で例外が発生したときに、行われる処理を書く

tryの処理(1)

- 例外が発生する可能性のある処理
  - 標準入力の処理
  - ファイル入出力の処理

    Javaの文法上の規則として、例外処理を書かなければならないもの
    (書かなければコンパイルエラー)
  - 配列を扱う処理
  - 文字列をint型に変換する処理
  - 割り算の処理

    文法上の規則としては、例外処理を書く必要はないが、必要に応じて自分の判断で例外処理を書くもの
  - etc.

tryの処理(2)

- 例外が発生する可能性のあるポイント
  - tryで、例外が発生する可能性のあるポイントをきちんと囲む必要
    - このポイントを囲んでいなければ、例外処理の意味はなし
    - 標準入力やファイル入出力では、このポイントを囲んでいなければ、コンパイルエラー

例外処理の書き方(基本形)

```
try {
    例外が発生する可能性のある処理
}
catch (例外の種類を表すクラス名  変数名) {
    例外が発生したときに行う処理
}
```

例外にも様々な種類

catchの処理(例外の種類)

- 例外の種類を表すクラス名
  - 例外には、様々な種類が存在
    - 入出力に関係する例外(入出力ができなかった場合に例外が発生)
    - 配列の添え字に関する例外(利用可能な範囲外の添え字を使おうとしたときに例外が発生)
    - 割り算に関する例外(数を0で割ろうとしたときに例外が発生)
  - tryで発生する可能性のある例外の種類を指定
    - 適切な種類を指定しておかないと、例外処理ができない

例外の種類(IO)(1)

- IOException
  - 入出力に関する例外
    - 標準入力・ファイル入力で、入力ができない場合
      - 標準入力:プログラムをターミナルから起動していない場合などは、入力不可能
      - ファイル入力:読み込もうとしたファイルが、「読み込み」のアクセス権がない場合などは入力不可能
    - ファイル出力で、出力ができない場合
      - 書き込もうとしたファイルが、「書き込み」のアクセス権がない場合などは出力不可能

例外の種類(IO)(2)

- IOException
  - 分類されているパッケージ: java.io
    - 「import java.io.IOException」または「import java.io.*;」がなければコンパイルエラー
  - 例外が発生する可能性のあるポイントが、tryの中に書かれていなければ、コンパイルエラー
    - ポイント:readLine()メソッド,ファイルを開く処理など

例外の種類(IO)(3)

標準入力

```
String str;
BufferedReader br = new BufferedReader(new  InputStreamReader(System.in));
try {
   str = br.readLine();
}
catch (IOException e) {
}
```

ファイル入力

```
String str;
try {
  FileReader fr = new FileReader("入力するファイルの名前");
  BufferedReader br = new BufferedReader(fr);

  str = br.readLine();
  br.close();
}
catch (IOException e) {
}
```

例外が発生する可能性のあるポイント(実際に入力をしているポイント)

発生する例外は入出力関係、と指定

catchの処理(内容)(例2)

```
try {
   BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
   str = br.readLine();
   int num = Integer.parseInt(str);
}
catch(NumberFormatException e) {
   System.out.println("入力されたデータは数値ではないため、処理できません。");
}
```


複数種類の例外に対する処理

- 1つのtryの中に複数種類の例外が発生することも

```
String str;
int num;
try {
  FileReader fr = new FileReader("sample.txt");
  BufferedReader br = new BufferedReader(fr);

  str = br.readLine();
  num = Integer.parseInt(str);
  br.close();
}
```

FileNotFoundExceptionの可能性

IOExceptionの可能性

NumberFormatExceptionの可能性

例外処理の書き方(応用)

```
try {
   例外が発生する可能性のある処理
}
catch (例外の種類を表すクラス名1 変数名) {
   1の例外が発生したときに行う処理
}
catch (例外の種類を表すクラス名2 変数名) {
   2の例外が発生したときに行う処理
}
```

catchはいくつ分書いてもOK

例外処理の書き方(応用)(例)

```
String str;
int num;
try {
  FileReader fr = new FileReader("sample.txt");
  BufferedReader br = new BufferedReader(fr);

  str = br.readLine();
  num = Integer.parseInt(str);
  br.close();
}
catch(FileNotFoundException e) {
  System.out.println("このファイルは存在しません。");
}
catch(IOException e) {
  System.out.println("このファイルからデータを読み込むことはできません。");
}
catch(NumberFormatException e) {
  System.out.println("読み込んだデータを数値に変換することができません。");
}
```

catchを必要なだけ並べる

ファイル入力

書式～入力～(全体像)

```
try {
  String str;

  FileReader fr = new FileReader(入力するファイル名);
  BufferedReader br = new BufferedReader(fr);

  str = br.readLine();
  br.close();
  fr.close();
}
catch (IOException e) {
   System.out.println("指定されたファイルの入力ができません。");
}
```


書式～入力～(詳細2)

```
try {
  String str;

  FileReader fr = new FileReader(入力するファイル名);
  BufferedReader br = new BufferedReader(fr);

  str = br.readLine();
  br.close();
  fr.close();
}
catch (IOException e) {
     System.out.println("指定されたファイルの入力ができません。
");
}
```

どのファイルの内容を入力するか、ファイルの名前を決める

- 「入力するファイル名」は、単なる文字列でも、変数に代入された文字列でも良い
- 入力するファイルは、Javaプログラムと同じ場所に置いておく

書式～入力～(詳細4)

```
try {
  String str;

  FileReader fr = new FileReader(入力するファイル名);
  BufferedReader br = new BufferedReader(fr);

  str = br.readLine();
  br.close();
  fr.close();
}
catch (IOException e) {
     System.out.println("指定されたファイルの入力ができません。");
}
```

「読むために」ファイルを開く作業(「fr」の部分は、FileReaderの変数名を入れること)

書式～入力～(詳細5)

```
try {
  String str;

  FileReader fr = new FileReader(入力するファイル名);
  BufferedReader br = new BufferedReader(fr);

  str = br.readLine();
  br.close();
  fr.close();
}
catch (IOException e) {
     System.out.println("指
}
```

開いたファイルを1行ずつ(文字列として)読んでいく(「br」の部分は、BufferedReaderの変数名を入れること)

- 1つ目の「br.readLine()」で1行目を読む
- 2つ目の「br.readLine()」で2行目を読む
- ……

※ただし、2行目を読んだあとに1行目をもう一度 読んだり、2行目をとばして3行目を読む、ということはできない

書式～入力～(詳細6)

```
try {
  String str;

  FileReader  fr = new  FileReader(入力するファイル名);
  BufferedReader  br = new  BufferedReader(fr);

  str = br.readLine();
  br.close();
  fr.close();
}
catch (IOException e) {
      System.out.println(
}
```

ファイルを読み終わった後、ファイルを閉じる作業
(プログラムでは、開いたファイルは必ず閉じる作業をしなければならない)
(「br」の部分は、BufferedReaderの変数名、「fr」の部分は
FileReaderの変数名を入れること)

ファイル入力の例

プログラム:

```
try {
  String str;

  FileReader fr = new FileReader("Sample.txt");
  BufferedReader br = new BufferedReader(fr);

  str = br.readLine();
  System.out.println(str);
  br.close();
  fr.close();
}
catch (IOException e) {
      System.out.println("指定されたファイルの入力ができません。");
}
```

「Sample.txt」の中身:

情報処理技法
サンプルファイル

出力結果:

情報処理技法

何行も書かれているファイル

- ファイルを最初から最後まで全部読むには?
  - 「br.readLine()」をファイルの行数分書く?
  - ファイルの行数が多いときは?
  - ファイルの行数が何行かわからないときは?

「ファイルが読み込み可能」
という条件でwhile文を使う

ファイルの最後の行まで読み込んでしまうと、その次の行を
読み込もうとしたときに「読み込み不可能」という結果が返ってくる

「読み込み可能」という条件

```
String str;
while (br.ready()){
  読み込んだ文字列に対する処理
}
```

➢「br.ready()」で、ファイルの中にまだ読み込んでいない行が存在するかをチェック
  ✓読み込んでいない行が存在すれば、「true」という結果
  ✓読み込んでいない行が存在しなければ、「false」という結果
    ファイルの終わりに達していない限り、whileの中身を実行する

while文の中身は...

- while文の中身(読み込んだ1行1行の扱い)は、プログラムの目的に応じて書く
  - strの内容を配列に代入する
    →ファイルの1行1行を配列として扱う
  - strの内容を数値に変換する(そして配列に代入する)
  - strの内容を細かく分解する(例えばスペースで区切って単語に分解する)
  - etc.

ファイル出力

書式～出力～(全体像)

```
try {
  FileWriter fw = new FileWriter(出力するファイルの名前);
  BufferedWriter bw = new BufferedWriter(fw);
  PrintWriter pw = new PrintWriter(bw);

  pw.println(書き込む内容);
  pw.close();
  bw.close();
  fw.close();
}
catch (IOExceptione) {
      System.out.println("指定されたファイルに出力ができません。");
}
```


書式～出力～(詳細2)

```
try {
  String str;
  FileWriter fw = new FileWriter(出力するファイルの名前);
  BufferedWriter bw = new BufferedWriter(fw);
  PrintWriter pw = new
  pw.println(書き込む内容);
  pw.close();
  bw.close();
  fw.close();
}
catch (IOException e) {
      System.out.println("指定されたファイルに出力ができません。");
}
```

どのファイルに書き込むか、ファイルの名前を決める

- 「出力するファイル名」は、単なる文字列でも、変数に代入された文字列でも良い
- 出力するファイルは、Javaプログラムと同じ場所に作成される

書式～出力～(詳細4)

```
try {
  String str;
  FileWriter fw = new FileWriter(出力するファイルの名前);
  BufferedWriter bw = new BufferedWriter(fw);
  PrintWriter pw = new PrintWriter(bw);

  pw.println(書き込
  pw.close();
  bw.close();
  fw.close();
}
catch (IOException e) {
      System.out.println("指定されたファイルに出力ができません。");
}
```

「書き込むために」ファイルを開く作業(「fw」の部分はFileWriterの変数名、「bw」の部分はBufferedWriterの変数名を入れること)

書式～出力～(詳細5)

```
try {
  String str;
  FileWriter fw = new FileWriter(出力するファイルの名前);
  BufferedWriter bw = new BufferedWriter(fw);
  PrintWriter pw = new PrintWriter(bw);

  pw.println(書き込む内容);
  pw.close();
  bw.close();
  fw.
}
catch
      S
}
```

開いたファイルに内容を書き込んでいく(「pw」の部分は、PrintWriterの変数名を入れること)
- 「pw.println(…)」では、書き込んだ後に改行が入る
- 「pw.print(…)」では、書き込んだ後に改行が入らない

→目的に応じて使い分けること

書式～出力～(詳細6)

```
try {
  String str;
  FileWriter fw = new FileWriter(出力するファイルの名前);
  BufferedWriter bw = new BufferedWriter(fw);
  PrintWriter pw = new PrintWriter(bw);

  pw.println(書き込む内容);
  pw.close();
  bw.close();
  fw.close();
}
catch (IOException e)
      System.out.println
}
```

ファイル書き込みが終わった後、ファイルを閉じる作業
(プログラムでは、開いたファイルは必ず閉じる作業をしなければならない)
(「pw」・「bw」・「fw」の部分は、それぞれPrintWriter・BufferedWriter・FileWriterの変数名を入れること)

ファイル出力の例

プログラム:

```
try {
  String str;
  FileWriter fw = new FileWriter("Sample.txt");
  BufferedWriter bw = new BufferedWriter(fw);
  PrintWriter pw = new PrintWriter(fw);

  pw.println("情報処理技法");
  pw.print("サンプルファイル");
  pw.close();
  bw.close();
  fw.close();
}
catch (IOException e) {
      System.out.println("指定されたファイルに出力ができません。");
}
```

実行結果
「Sample.txt」の中身:

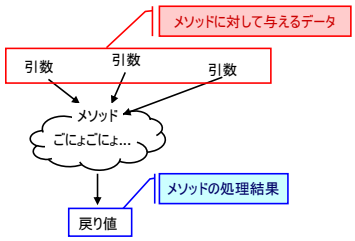
```
情報処理技法
サンプルファイル
```

メソッドの作り方・使い方

メソッド

- プログラム中で行われる処理の手順をまとめたもの
  - 複数の処理をまとめて、1つの名前を付けたもの
- メソッド名、引数、戻り値(返り値)という構成
  - メソッド名: メソッドの名前
  - 引数: メソッドに渡す情報(計算等の処理の材料にするデータ)
    - 処理の材料にするデータのみ、引数として定義
  - 戻り値(返り値): メソッドの内容を実行したときの処理結果
    - 多くの場合、戻り値を変数に代入して利用する
    - 「変数名 = メソッド」で、変数に戻り値が代入される

メソッドのイメージ



メソッドを作る(1)

- 定義するのは ごにょごにょ… の部分

メソッドを作るときのお約束
- ただし、いつも必ず「public static」であるとは限らない
- この授業では違う書き方をする場合も

```
public  static  戻り値のデータ型  メソッド名(引数) {
   メソッドでの処理内容
   return  処理結果;
}
```


メソッドを作る(2)

- 定義するのは ごにょごにょ… の部分

戻り値は1つだけ

```
public  static  戻り値のデータ型  メソッド名(引数) {
   メソッドでの処理内容
   return  処理結果;
}
```


メソッドを作る(3)

メソッドの名前(名前の付け方は、クラス名や変数名と同じ)

```
public  static  戻り値のデータ型  メソッド名(引数) {
   メソッドでの処理内容
   return  処理結果;
}
```

Javaの命名規則としては…
- 先頭の単語は動詞にする
- 複数の単語を連結するときは、先頭の単語はすべて小文字、2つ目以降の単語は先頭の文字のみ大文字、あとは小文字
  - 変数と同じ

メソッドを作る(4)

- 「引数のデータ型　引数の変数名」と書く
- 引数はいくつあっても良い(「,」で区切る)
- データ型はそれぞれ異なってもかまわない

```
public　static　戻り値のデータ型　メソッド名(引数) {
   メソッドでの処理内容
   return　処理結果;
}
```


メソッドを作る(5)

処理内容は何を書いても良い
(if, for, while, …)

```
public　static　戻り値のデータ型　メソッド名(引数) {
   メソッドでの処理内容
   return　処理結果;
}
```

※「処理内容」の部分では、「引数」で定義した変数を
通常の変数として利用できる

```
public int sum(int num1, int num2) {
   int result;
   result = num1 + num2
   return result;
}
```


メソッドを作る(6)

- 「処理結果」を返すという意味(「変数名 = メソッド名(引数)」と書くと、「変数名」の中に処理結果が代入される)
- この文でメソッドの内容が終わる(この後には文を書かないこと)
- 処理結果のデータ型と戻り値のデータ型は同じもの

```
public　static　戻り値のデータ型　メソッド名(引数) {
   メソッドでの処理内容
   return　処理結果;
}
```


return文

- 戻り値のデータ型とreturn文で返すデータ型は同じでなくてはならない

```
public int sum(int num1, int num2) {
   int result;
   result = num1 + num2;
   return result;
}
```

○ データ型が同じ

```
public String sum(int num1, int num2) {
   int result;
   result = num1 + num2;
   return result;
}
```

× 違うデータ型

メソッドの作成例

処理結果を「double」型で返す　　引数の定義

引数に与えられた3つの数の平均を求めるメソッド

```
public static double average(int num1, int num2, int num3) {
   double result;
   result = (double) (num1 + num2 + num3) / 3;
   return result;
}
```

処理内容

「return」で、「結果を返す」という意味
(「return」の後に書く処理結果のデータ型は、戻り値のデータ型と同じにすること)

メソッドを作る場所

mainの上か下に作る
(どちらに作っても良い)

```
public class ファイル名 {
   [          ]
   public static void main(String[] args) {
   }
   [          ]
}
```

メソッドの作成例

```
public class Sample {
 public static double average(int num1, int num2, int num3) {
  double result;
  result = (double) (num1 + num2 + num3) / 3;
  return result;
 }
 public static void main(String[] args) {
 }
}
```


作ったメソッドを使う

- メソッドを使う部分は、「public static void main」の中に書く
  - 「メソッド名(引数の値)」でメソッドを呼び出す

作ったメソッドを使う(例)(1)

メソッドの定義

```
public class Average {
 public static double average (int num1, int num2, int num3) {
  int sum;
  double result;
  sum = num1 + num2 + num3;
  result = (double) sum / 3;

  return result;
 }
 public static void main(String[] args) {
  double result;
  result = average(10, 20, 30);
 }
}
```


作ったメソッドを使う(例)(2)

```
public class Average {
 public static double average (int num1, int num2, int num3) {
  int sum;
  double result;
  sum = num1 + num2 + num3;
  result = (double) sum / 3;

  return result;
 }
 public static void main(String[] args) {
  double result;
  result = average( 10, 20, 30 );
 }
}
```

引数には具体的な値または変数を書く
(引数の順番は、メソッドを作ったときの順番と同じに)

averageというメソッドを呼び出す

作ったメソッドを使う(例)(3)

```
public class Average {
 public static void main(String[] args) {
  double result;
  result = average(10, 20, 30);
 }

 public static double average (int num1, int num2, int num3) {
  int sum;
  double result;
  sum = num1 + num2 + num3;
  result = (double) sum / 3;

  return result;
 }
}
```

処理の流れ
1. mainメソッドで、averageメソッドの引数に10, 20, 30を指定する
2. 指定された10, 20, 30というデータがaverageメソッドの引数の変数「num1」、「num2」、「num3」にそれぞれ代入される
3. averageメソッド内で引数の値を使って計算され、変数resultに結果が代入される
4. averageメソッドの変数resultの値がmainメソッドの変数resultに代入される

やってみよう!(1)

- 24時間表記の時間を入力し、それを12時間表記の時間として出力するプログラム
  - 24時間表記の時間は標準入力で入力すること
  - 24時間表記の時間を引数とし、12時間表記の時間を戻り値とするメソッドを作ること
  - Ex.
    - 入力:5 →出力:5 am.
    - 入力:23 →出力:11 pm.

※必要と思われる例外処理をしておくこと

やってみよう!(2)

- これまでに作成したJavaファイルを読み込み、ファイルの行数を数えるプログラム
  - Javaファイルの名前は、標準入力で入力すること
  - ファイルの行数は、標準出力で出力すること
- 0～1000までの乱数を100個作成し、その乱数をファイルに書き込むプログラム
  - 例えば、「random」というint型の変数がある場合、
    random = (int) (Math.random() * 1000);
    で、変数「random」に乱数が1つ入る
  - 乱数は、「,」やスペースで区切ってファイルに書き込む

※必要と思われる例外処理をしておくこと