第1章　オブジェクト指向プログラミング

担当者：瀧川一学（大規模知識処理研究室）
連絡先：部屋：情報科学研究科棟 6-16　内線：6470
E-mail: takigawa@ist.hokudai.ac.jp

1　はじめに

この演習では 作業 や 課題 を通して、Java 言語を用いた「オブジェクト指向プログラミング」の考え方と初歩 (**カプセル化、継承、多態性**の三要素) を学びます。各自、この実習書に沿って 作業 と 課題 をこなしてください。星印の 課題* は余裕のある人向けの発展課題なので飛ばしても構いません。分からないことが出てきたら随時、担当教員か TA に聞いてください。

1.1　実習の進め方

このテーマでの演習は全 6 日あります。下記は進め方の目安ですが、各自のペースで進めてください。

- （1 日目）1 〜 3 節：Java プログラミングの基本と Eclipse の使い方
- （2 日目）4 節：標準クラスライブラリの使い方 (クラスを使ってみる)
- （3 日目）5 節：オブジェクト指向入門 1 (クラスを作ってみる)
- （4 日目）6 節：オブジェクト指向入門 2 (継承と多態性)
- （5 日目）7 節：オブジェクト指向入門 3 (Interface と抽象クラス)
- （6 日目）レポート作成日

1.2　実習の Web サイト

この「実習書」と各サンプルの「ソースコード」等は下記の Web サイトからダウンロードできます。適宜、実習中にも訂正や追加資料を置くことがありますので定期的にチェックしてみてください。

```
http://art.ist.hokudai.ac.jp/~takigawa/csit_java/
```

1.3 レポート提出と〆切

すべての 課題 を実施し、作成したプログラム、実行結果、解析、考察をレポートにまとめて提出してください。 作業 はレポートに含める必要はありませんが、 課題 の回答には必要なものです。余裕のある人は発展課題として 課題* も行いレポートに含めてください。このレポートの内容と出席態度に基づき成績評価を行います。

レポートは A4 の用紙を使用し、実験テーマ名、氏名、学生番号を記した表紙を付け、左上をホチキスで留め、以下の期日までに提出すること。各課題のプログラム、実行結果、解析、考察などはレポートにまとめて提出すること。特別な事情がない限り、その後のレポート提出は認めない。

提出〆切: 2017 年 4 月 24 日 (月) 午後 5:00
提出先: 情報科学研究科 6F 6-16 室 (瀧川) ドアポストの提出口

- レポートの書式は特にありません。Word で作っても LaTeX で作っても良いです。
- ソースコードなど印刷すると長くなるものはレポートに電子的提出を行う旨を明記し、下記メールアドレスに提出すること[1]。レポート本体のメール提出は認めません。ソースコード以外の実行結果、解析、考察などの記載部分はレポート本体の方に入れること。

  宛先: `java@art.ist.hokudai.ac.jp`

1.4 作業・課題リスト

作業と課題 (必須)

3 節	作業 1.1	作業 1.2	作業 1.3	作業 1.4
	課題 1.1	課題 1.2		
4 節	作業 1.5	作業 1.6	作業 1.7	
	課題 1.3	課題 1.4		
5 節	作業 1.8	作業 1.9		
	課題 1.5	課題 1.6	課題 1.7	
6 節	作業 1.10	作業 1.11		
	課題 1.9	課題 1.10		
7 節				
	課題 1.12	課題 1.13		

発展課題 (余裕のある人のみ)

3 節		
4 節		
5 節		
	課題* 1.8	
6 節		
	課題* 1.11	
7 節		
	課題* 1.14	課題* 1.15

[1] 提出は管理の都合上、上記アドレスに限ります。瀧川や TA のメールアドレスに送っても提出とはみなされません。

2 Java プログラムと「クラス」

プログラミング言語 Java(以下、Java 言語) は、現在最も広く利用されているオブジェクト指向プログラミング言語の一つです。演習では Java 言語自体については最低限必要な範囲しか扱わないので、さらに詳しい情報は付録 1 を参考に自習することをお勧めします。ここでは、C 言語をはじめたときのように画面に「Hello, World」と表示される Java プログラムのソースコードを見てみましょう。

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

ソースコード 1.1: HelloWorld.java

このプログラムをコンパイルして実行するには、emacs、vim、nano 等のテキストエディタで上記のソースファイルを入力したテキストファイルを作成し、HelloWorld.java という名前で保存したのち、コマンドラインから以下のように入力します[2]。

```
$ javac HelloWorld.java
$ ls
HelloWorld.java   HelloWorld.class
$ java HelloWorld
Hello, World
```

「javac」コマンドが Java のコンパイラであり上記ソースファイルを実行ファイルに変換します。HelloWorld.class がコンパイルされた実行プログラムであり、これを実行するときには「.class」をつけないで「java」コマンドに指定します。このプログラムは C 言語と完全に異なる原理で実行されます。以下のようにして HelloWorld.class を実行してみます。

```
$ java -verbose:class HelloWorld
[Opened /usr/java/jdk1.8.0_74/jre/lib/rt.jar]
[Loaded java.lang.Object from /usr/java/jdk1.8.0_74/jre/lib/rt.jar]
:
[Loaded java.lang.Shutdown$Lock from /usr/java/jdk1.8.0_74/jre/lib/rt.jar]
```

この Hello, World を実行するために、java.lang.Object や java.lang.String などたくさんのものが「Load」されたようです。これらは「クラス (class)」と呼ばれる Java プログラムの基本構成単位です。先ほどの HelloWorld.java をコンパイルして作成されたファイル HelloWorld.class も、ソースコードの 1 行目やファイル名から類推できるように「class」の一つです。実際下記のコマンド[3]によると

```
$ java -verbose:class HelloWorld | grep Loaded | wc -l
     391
```

この HelloWorld という単純なプログラムを実行するのに実に 391 個ものクラスが関わっていることが分かります[4]。Java には標準で Java 言語の基本プログラミングや様々な応用プログラミングで必要となる多数のクラスが定義されています。

[2]行頭の「$」はシェルのコマンドプロンプト記号なので入力しないで下さい。
[3]`wc -l` の「`-l`」は「-1」ではなく「-l」(ハイフン・エル) です。
[4]grep コマンド, wc コマンド等の UNIX コマンドやシェルのパイプ処理については man コマンドや Linux/UNIX に関する書籍などで意味と使いかたを確認してください。

3 「クラス」を使ったプログラミング

本演習はこうしたクラスの定義や利活用とクラス間のやりとりでプログラミングを行う「オブジェクト指向プログラミング」への入門となっています。Hello, World を出力するだけだったらこんな周りくどいことをするメリットはありません。なぜ「クラスを色々作成しておいてクラス間のやりとりとしてプログラムを定義する」と良いのかを理解してもらうのがこの演習の一つの目標です。

まず、Java の開発における標準的開発環境である「Eclipse」を使った Java プログラムの開発、コンパイル、実行の仕方、を通して、Java の基本文法に慣れるところから始めましょう。なお、特に使い慣れているエディタがある人はそれを使っても良いでしょう。

以降の説明を読み下記の作業と課題に取り組んでください。

作業 1.1

ソースコード 1.1「HelloWorld.java」を Eclipse で作成し、実行せよ。

作業 1.2

ソースコード 1.2「HelloWorld2.java」のファイルを Web サイトからダウンロードし、Eclipse にインポートし、実行せよ。

作業 1.3

Web サイトのサンプル「TestArray.java」を書き換えて、array1 と array2 について、要素を全て足した値とすべて掛けた値を出力するようにせよ。

課題 1.1

Web サイトのサンプル「TestArray2.java」を書き換えて、要素を全て足した値を出力する public static メソッド「getSum」、及び、値が 0 以外の要素を全て掛けた値を出力する public static メソッド「getNonZeroProduct」を追加し、main メソッド中で配列 $(1, 2, \ldots, 10)$ と $(-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5)$ に対して適用するようにせよ。引数の配列の長さが変わっても使えるように定義すること。また、同様の「関数」を C 言語でどのように実現していたかについて違いを考察せよ。

作業 1.4

Web サイトからサンプル「TestArrayList.java」をインポートし実行せよ。

課題 1.2

実行時引数で「1 3 4 3 9 8 0」のように任意長の整数列を入力して、それを ArrayList を利用して入力と逆順で出力するプログラムを作成せよ。

3.1 Eclipse を使った Java プログラムの開発と実行

この演習のような小規模なプログラミングの場合は、テキストエディタで開発しても良いのですが、Java では多くのクラスを同時に扱うため、統合開発環境 (IDE, Integrated Development Environment) を使う開発が便利です。デバッグ、リファクタリング、テストといった開発に付随する作業も効率的にサポートしてくれます。この演習では、IBM によって開発された高機能ながらオープンソースの IDE である「Eclipse」を使ってみましょう。

Eclipse を使用して、先ほどの HelloWorld.java を作成してみます。Eclipse によるプログラム作成は、以下のような流れになります。Eclipse で生成される class ファイルの場所については付録 2 を参照してください。

Eclipse の起動

計算機室環境の Linux (CentOS7.2) で Eclipse を起動するには、左上の「アプリケーション」の「プログラミング」の「Eclipse Mars 2 (4.5.2)」を選択します。初回起動時にはワークスペースランチャーが起動し、主に開発で用いる「ワークスペース」(作業ディレクトリ) をどこにするか聞かれるかもしれませんが、そのまま「OK」を押し標準設定で進めて構いません。

プロジェクトの作成

Eclipse では開発するプログラムを構成するソースファイル群やそれらをコンパイルし、実行するための設定などをまとめたものを「プロジェクト」と呼びます。一つのプロジェクトで複数の異なるプログラムを管理することもできますが、**本実験では基本的に作業や課題ごとプロジェクトを作成する**ことにします。例えば課題 1 であれば「Kadai1」というプロジェクトを作成すれば良いでしょう。

プロジェクトの作成は、以下の手順で行います。

1. メニューから「ファイル」→「新規」→「プロジェクト」を選択する。
2. 「Java プロジェクト」を選択し、「次へ」を選択する。
3. プロジェクト名に適当な名前を入力し (例えば Kadai1 など)、「次へ」を選択する。
4. ビルド設定は特に変更せず、「完了」を選択する。(関連付けられたパースペクティブを開きますか？と聞かれる場合「はい」とするとメニューやエディタなどが Java 開発用に設定されます)

以上でプロジェクトの作成は完了です。次ページのような画面が開いていることを確認してください。もし「ようこそ (Welcome)」ウィンドウが表示されている場合にはこれを閉じ、Package Explorer (Eclipse ウィンドウ左側の欄) にプロジェクト名の項目が追加されていることを確認してください。

ソースコードの作成

次に、プロジェクトにソースファイルを追加し、そのファイルにプログラムを書いていきます。Eclipse ではソースコードの定型的な部分を自動生成する機能がありますが、以下の手順ではそれらを使用せず、一からソースファイルを作成してもらうことにします。

1. メニューから「ファイル」→「新規」→「ファイル」を選択する。

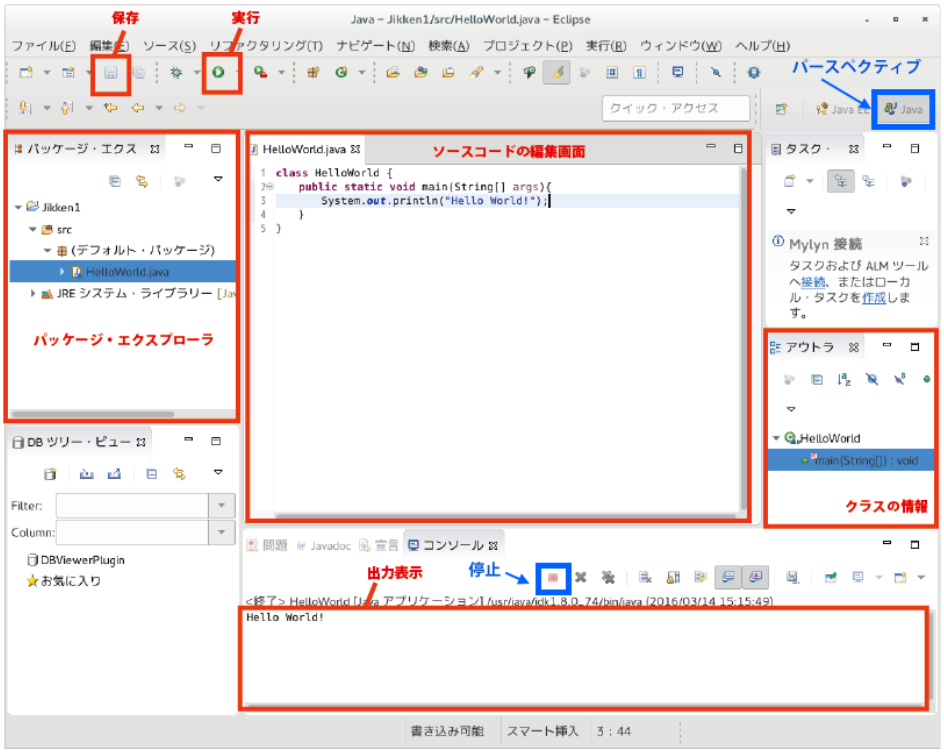

図 1.1: Eclipse の Java 開発用の基本画面 (どの画面をどう表示するかはカスタマイズできる)

2. 親フォルダとして、作成したプロジェクト・フォルダの中の「src」を選択[5]し、**ファイル名として、「ソースファイルの中で public とするクラス名」 + 「.java」を入力**する。今回は「HelloWorld.java」を入力する。最後に「完了」を選択する。

3. 作成したファイルの編集画面が表示されるので、ソースコードの内容を入力する。

4. メニューから「ファイル」→「保管」を選択 (もしくは Ctrl+S) し、ソースファイルの内容を保存する。(上部ツールバーの保存ボタンでも可)

プログラムの実行設定と実行

プログラムを実行するための設定は実行構成 (Run configuration) と呼ばれます。実行構成において最低限設定すべき項目は実行したいプログラムを管理しているプロジェクト名と、そのプログラムにおける main メソッドが定義されているクラス名です。 これらを設定し、プログラムを実行するまでの手順は以下のようになります。

1. メニューから「実行」→「実行構成」を選択する。

2. 左にある「Java アプリケーション」の中のプロジェクト名のアイコンをクリックする。

[5]Eclipse ではソースファイル (*.java) はこの画面左の「パッケージエクスプローラー」のプロジェクト名の下の「src」というところに置く必要があります。「main 関数がみつからない」のようなエラーがでる場合確認してみてください。

3. メイン・クラス の欄に main メソッドが定義されているクラス名 (今回は HelloWorld) が入力されていることを確認する。(そうでない場合、もしくは該当するクラスが複数あり変更したい場合は、入力欄の右の「検索」 を選択する。現在のプロジェクトで定義されているクラスのうち、main メソッドが定義されているクラスが 表示されるので、それを選択し「OK」を選択する)

4. 「実行」を選択する。

標準出力への出力結果は、一番下の欄の「Console」欄に表示されます。また Ctrl+F11 キーや上部の再生ボタンのアイコンをクリックで、直前に利用した実行構成を使ってプログラムを実行することができます。通常 実行構成の設定は一度のみ 行い、プログラムの作成途中では Run ボタンや Ctrl+F11 キーで実行するのが便利です。

Eclipse のワークスペースと生成ファイル

Eclipse は最初にワークスペースに指定したディレクトリにファイルを置きます。このディレクトリは初期設定では「`~/workspace`」になっています。ソースコードのファイル及び生成される実行ファイルはプロジェクトごとにディレクトリで管理されます。プロジェクト名が「Kadai1」の場合、「`~/workspace/Kadai1`」になります。ソースファイルが置かれる「src」や生成される実行ファイル (.class ファイル) が置かれる「bin」などのサブディレクトリが配置されます。

Eclipse の機能紹介

- ソースコードをドラッグで選択し、右クリック→ソース→フォーマットを選ぶと、文法に従って、見やすいようにソースコードを整形してくれます。インデントが崩れたら試してみてください。
- 入力された左丸括弧 '(' に対応する右丸括弧 ')' など、対応する括弧を自動的に挿入します。括弧の中の内容を入力したあと、TAB キーを押すと右丸括弧の次の位置にカーソルが移動します。また、自動挿入された括弧を無視して入力しても、余分な括弧は入らないようになっています。
- ソースコードを入力中、ソースコードの内容を解析し、問題がある箇所を指摘します。ある行に問題がある場合は、その行の左端にアイコン (警告を表す黄色いマーク、もしくはエラーを表す赤地に×のマーク) が表示されます。また、メソッド名やクラス名等のキーワードに入力間違いがあると考えられる場合は、そのキーワードの下部に赤い波線が表示されます。これらのアイコンや波線をマウスでポイントする (または、カーソルが置かれた状態で Ctrl+1 キーを押す) と、問題の詳細が表示されますので、それを参考にソースコードの修正を行い、再度ソースコードの保存を行ってください。
- ソースコードが保存されると自動的にソースコードのコンパイルを行います。よって、ソースコード保存時に問題が一つも指摘されなかった場合は、ソースコードのコンパイルが正常に終了したことになります。
- 現在のソースコードの文脈で利用できるコードを補完する Content Assist、赤線で示された 間違いの修正方法を提示する Quick Fix、メソッドのパラメータの型をポップアップで表示してくれる Parameter Hints など、効率よくソースコードを編集するための便利な機能が備わっています。これらの機能については、「Edit」メニューから呼び出し及びショートカットキーの確認ができます。

実行時引数とファイルインポート

下記のコマンドライン実行のような実行時引数を扱えるプログラム「HelloWorld2」のソースコードをダウンロードして実際に Eclipse で実行してみる方法を見てみましょう。

```
$ java HelloWorld2 this test
Args 1: this, 2: test
```

Eclipse で実行時にコマンドライン引数を指定するには、実行するまえの「実行構成」画面で、「引数」タブをクリックして、「プログラムの引数」というところに記述してください (上記の場合は例えば「this test」と入力する)。

```
1 public class HelloWorld2 {
2     public static void main(String[] args) {
3         System.out.printf("Args 1: %s, 2: %s\n",args[0],args[1]);
4     }
5 }
```

ソースコード 1.2: HelloWorld2.java

上記の HelloWorld2.java をサポート Web サイトからダウンロードして実行する手順は以下のようになります。例としてプロジェクト名を「TestImport」とします。

1. ファイル → 新規 → プロジェクト →Java プロジェクトで「TestImport」を作成する。
2. アプリケーション → インターネット →Google Chrome で Web ブラウザを起動し `http://art.ist.hokudai.ac.jp/~takigawa/csit_java/` を開く。
3. HelloWorld2.java を右クリックし、「名前を付けてリンク先を保存」を選択する。名前「HelloWorld2.java」のまま、デスクトップに保存する。
4. Eclipse のパッケージエクスプローラの「TestImport」の ▽ をクリックし src を表示させておき、デスクトップの HelloWorld2.java を src へドラッグ＆ドロップ
5. ファイル操作を聞かれるので「ファイルをコピー」を選んで「OK」を選択する。
6. Eclipse のパッケージエクスプローラの TestImport→src→ デフォルトパッケージに「HelloWorld2.java」が見えるので、ダブルクリックするとエディタ画面に内容が表示される。
7. そのまま実行すると「ArrayIndexOutOfBoundsException」というエラーが出る。これは引数が必要なプログラムなのに何も指定していないため。引数の指定のために、メニューから実行 → 実行構成を開き、引数のタブをクリック。「プログラムの引数」に例えば「this is test」と入力し、「実行」を選択する。
8. 画面下部のコンソールに「Args 1: this, 2: is」と出力される。

Java プロジェクト作成における注意 (まとめ)

1. 1 つの java ファイルに 1 つの「(public) class」とする。
2. ファイル名は「そのクラス名.java」とする。
3. クラス名は既存のクラス名と重複してはならない。
4. この演習で Eclipse を使う場合、1 つの「プロジェクト」に各作業や課題ごとの java ファイルを全て置くのではなく、作業や課題ごとに Kadai1 や Sagyou1 などの「プロジェクト」を作成すること。

上記注意の補足：

- Java では命名規則の慣例 (Naming Convention) として、クラス名 (ファイル名も) は大文字で `UpperCamelCase` のように、変数やメソッド名は最初のみ小文字として `lowerCamelCase` のように、値が変わらない定数は大文字とアンダースコアで `SNAKE_CASE` のようにつける。
- 既存のクラス名と重複してはならないと言ったって既存のクラス名なんて全部把握できんやん、、、という実情を反映して「パッケージ」(p.49) という仕組みが用意されています。
- 1 つのプロジェクトに複数の課題の java ファイルを置く場合のように、1 つのプロジェクトに main メソッドを持つクラスが複数存在するとどの main メソッドを実行するか選択肢が生じる。もし 1 プロジェクトで複数のクラスを管理する場合、「実行構成」で適宜実行する main メソッドを持つクラスを指定すること。

3.2 配列と for 文

プログラミングでは、たくさんの対象をまとめて扱うことが多いので、「配列」は基本的なデータ構造になります。Java 言語でどのように配列を扱うことができるか、Web サイトからサンプル「TestArray.java」をダウンロード、Eclipse へインポートし実行してみましょう。C 言語の配列が分っていれば動作は理解できるかと思いますが、ソースコードを読み、下記の差異に注意してください。

- Java では「`int` 配列」が「`int[]` 型」のように配列自体が型のように扱われます。従って、後の例で説明するように C 言語と異なり、配列を return することもできます。
- for 文に 2 通り書き方があったり、array.length で配列長が取れるのは C 言語にはない特徴です。
- **変数のスコープ**: 特に for 文の中で定義される変数の影響範囲に注意してください。以下のように変数の宣言文 (int i) の位置によって、変数が参照できる範囲 (変数のスコープ) が決まります。以下の例では変数 i も j も for 文の外では参照できません。

```
for(int i=0; i<array1.length; i++){ // for 文の中で変数 i が宣言 (int i) されている
    int j = i+1; // 変数 j の宣言文
    System.out.println(i);
    System.out.println(j);
}
System.out.println(i); // エラー
System.out.println(j); // エラー
```

- Java ではコメントは C 言語のように「`/* ... */`」ともかけますが、この例のように 1 行コメントであれば行頭に「`//`」とかけばコメント行になります (C++と同じです)。
- 通常 C 言語で配列用のメモリを確保する処理は、Java では new によって実現されていることに注意しましょう。実際には「`int[] a = new int[5]`」と書けば、最初に `int` 型の要素 5 つ分の配列がメモリ上に作られ、`int[]` 型の変数 `a` がメモリ上に作られ、変数 `a` に配列領域の先頭要素のアドレスが代入されます (つまり実質的にはポインタ的に動作します)。
- free をしないで大丈夫なのかと不安になる人もいるかと思いますが Java では基本的に確保した領域が Java プログラムから参照できなくなると自動的に片付け処理が行われます。この便利な仕組みはガベージコレクションと呼ばれます。ただし、要らなくなったら即座に領域が解放されるとは限らず、いつ片付けられるかは基本的にはシステムが勝手に決めます。

3.3 java.util.ArrayList

配列は言語仕様にある基本データ構造ですが、Java には List, Set, Queue, Deque, Map, HashMap など便利なデータ構造が「部品」つまり「クラス」として標準クラスライブラリに用意されています。Web サイトのサンプル「TestArrayList.java」では java.util.ArrayList の使用例を示しています。「<>」記法については付録 3 も参照してください。「配列」と似ていますが、「配列」は定義時にサイズを指定する必要があったのに対して、ArrayList は逐次いくつでも追加していくことが可能です。こうした「入れ物」はいろいろなプログラムを作成するときに利用できるとても便利な「部品」です。ArrayList を使うには C 言語の include 文に相当する「import 文」が必要となります。標準ライブラリについては次節でもう少し詳しく解説します。

参考: ArrayList はクラスであり要素に対するループの書き方が 4 種類もあります。入力を処理する Scanner クラスについては後述。

```
ArrayList<Float> a = new ArrayList<>();
Scanner scanner = new Scanner("0.12 0.23 -1.5");
while(scanner.hasNextFloat()){
    a.add(scanner.nextFloat());
}
// (1) 拡張 for
for(float x : a){
    System.out.println(x);
}
// (2) 標準 for
for(int i=0; i<a.size(); i++){
    System.out.println(a.get(i));
}
// (3) Iterator
Iterator<Float> it = a.iterator();
while(it.hasNext()){
    System.out.println(it.next());
}
// (4) forEach ラムダ式！
a.forEach( (x)->{System.out.println(x);} );
```

3.4 関数

C 言語を習った人が次に気になるのは「関数」ではないでしょうか? 関数は C 言語においてはプログラムの構造を分かりやすくする非常に便利な概念でした。実は Java には「関数」というものはありません。おや？「main」のところは main 関数で関数じゃないのか？と思うかもわかりませんが、これは Java では「メソッド」と呼ばれます。どうみても関数にしか見えないものをなぜ「メソッド」と呼ぶのかは「オブジェクト指向」の根幹に関わります。

その理由は「クラス」の説明まで先送りにして、ここでは、例えば、`double sqrt(double)` などの「関数」として用意しておくべき機能は Java では main と同じく「public static」をつけて定義する、と理解しておきましょう。Web サイトのサンプル TestArray2.java の例は「関数」的に用意された「public static メソッド」と活用例です。配列を return できることも理解しておきましょう。

参考: もし、「`double[] array2 = array;`」とした場合、基本的には `array2` を書き換えると `array` の内容も書き換わります。上のでは「`array`」が保存されているメモリ上の場所 (アドレス) が「`array2`」にコピーされるので、「`array2`」で参照すると結局「`array`」と同じところを見に行ってしまいます。

4 オブジェクト指向の恩恵：既存のクラスライブラリを使ってみる。

簡単に言えば、オブジェクト指向は「大規模な」プログラムを「複数人」で作るために生まれた「部品化」の作法の一つです。「大規模な」プログラムをいきなり作成するのは大変なので、そのプログラムに必要な部品を作成して、それらを組み合わせて実現するのです。ただし、オブジェクト指向言語で記述さえすればオブジェクト指向が実現できるわけではありません。何を「部品」として用意しておくかに腕とセンスが問われます。この節では、まずよく考えられて設計された汎用部品として標準で提供されている「部品」を使ってみることで、オブジェクト指向の恩恵を考えます。(本節と同じ課題を C 言語でやれ、と言われたらどうでしょうか?)

以降の説明を読み下記の作業と課題に取り組んでください。

作業 1.5

「java.lang.String」の API リファレンスを参照しながら、「TestString.java」を書き換えて、上記例で使われていない他のメソッドを色々と試してみよ。

作業 1.6

Web サイトから「TestCalender1.java」および「TestCalender2.java」をダウンロードして、API リファレンスを参照しながらソースコードを分析し、違いと何をやっているかを分析せよ。

作業 1.7

API リファレンスで java.text.SimpleDateFormat を調べ、「フィールド」や「メソッド」と並んで「コンストラクタ」が何種類か定義されていることを確認せよ。

課題 1.3

TestCalendar1.java や TestCalender2.java で使われているクラスの API リファレンスを調べながら、自分の誕生日が何曜日だったか」および「自分が還暦 (60 歳になる誕生日) まであと何日あるのか (それを時間換算すると何時間になるか)」を計算するプログラムを作ってみよ。

課題 1.4

課題 1.3 では「自分の誕生日」に対する「今日」の結果を表示するものであった。これを標準入力から日付を入力できるよう変更し、日付 Y 生まれの日付 X における経過時間を表示するようにプログラムを書き換えよ。それを用いて、1916 年 4 月 30 日生まれの Claude Shannon は、Microsoft 社が 1995 年 8 月 24 日に Windows 95 を発売したとき、何歳だったのか、を調べよ。入力の形式などは問わないので各自定めること。

4.1 部品と設計図：「クラス」と「インスタンス」

「オブジェクト指向」は「部品化」の作法であると言いました。たとえば、自分で組み立てる本棚では、パーツ A(天板):1、パーツ B(下板):1、パーツ C(横板):2、パーツ D(背板):1、パーツ E(棚板):4、ネジ A:16、ネジ B:8、などと部品が同梱されているでしょう。このとき、パーツ E を例にとると、そのパーツの雛形を「クラス」、実際にその雛形で作られた 4 つのパーツ (棚板) の実体を「インスタンス」と呼びます。オブジェクト指向では、様々な部品の「雛形 (クラス)」を用意して、適切な数の「部品 (インスタンス)」を実体化してそのやりとりであらゆる処理を実現します。

さて、自動車の設計において「ドア」「エンジン」「エアコン」などが既に「部品」として用意されており、これらを組み立てて「自動車」にしなければいけない係の人を想像しましょう。自動車の組み立て係は「エンジン」や「エアコン」などの各々の部品の動作原理まで理解しておく必要があるでしょう

か。どれが「エンジン」や「エアコン」で各々「どうやって使うか (どうやって繋ぎあわせるか)」だけ分かっていれば中身の原理は組み立てる作業には必要ないはずです。「エアコン」のスイッチは自動車の室内に配置すべきで、どれが on スイッチなのかは知っておく必要があります。また、「エアコン」部品の電源の線がどれでどのような型かも知っておく必要があります。しかし、on スイッチを押すとなぜ冷房されるのかは組み立てには特に必要がありません。なので、どういう部品があるのかと、それが提供している機能を「どうやって使うか」だけを調べられれば良いはずです。

4.2 標準クラスライブラリと API

Java に標準で用意されているクラスライブラリにはどんなクラスがあるのか、どうやって使うのか、をどのように調べれば良いのでしょうか。先ほどのたとえのように、使うだけなら「中身の仕組みを知らなくても良い (ソースコードをいちいち読まなくて良い)」ので、部品の「機能リスト」と各々を「どうやって使うか」の「API(Application Program Interface)」だけ文書化されていれば十分です。

一般に、OS や特定のシステムの機能をアプリケーションプログラムから利用できるようにするインタフェースを API と言いますが、Java のクラスライブラリの API はドキュメント化されています。たとえば、Java SE 8 の場合、以下で見られます。Java のバージョンによって標準ライブラリの仕様は異なるので、他のバージョンを使う際は「Java API リファレンス」にバージョン番号などを加えてインターネットで検索してみてください。

```
http://docs.oracle.com/javase/jp/8/docs/api/
```

4.3 API リファレンスの使い方

３つのパネルからなっていて、左上に「パッケージ」、左下に「クラス」、右に説明が出ます。Java のクラスライブラリは大まかな機能ごとに「パッケージ」という単位にまとめられています。明示しなくても使える標準機能は前述したように「java.lang」という名前のパッケージになっています。

左上のパネルで「java.lang」をクリックしてみましょう。すると左下の画面が java.lang に用意されている「クラス」一覧に変わります。「インタフェース」という特殊なクラスも表示されますがとりあえず置いておきましょう。インタフェースについては演習の最後に扱います。表示されたクラスの中で知りたいクラスをクリックすると右に説明が表示されると思います。例えば、「Math」をクリックしてみましょう。そのクラスの「フィールド」と「メソッド」の一覧が表示されると思います。この中でさらに、「sqrt」をクリックしてみましょう。すると「`public static double sqrt(double a)`」であることが分かります。つまり、sqrt は java.lang.Math クラスにおいて、public static なメソッド (3.4 節で確認したように C 言語の関数的なもの) として用意されているようです。これらの static なメソッドは「クラスメソッド」とも呼ばれるものです。

java.lang の場合、何もしなくても使えるので、これを使うには sqrt は Math クラスのメソッドであるということだけ明示すれば良く、例えば、「Math.sqrt(5.0)」のように使うことができます。

しかし、このような関数的な箇所のみであれば C 言語の math や stdlib とあまり変わらない感じがします。ここでは、標準ライブラリの中から様々なプログラムで用いられることの多いクラスをいくつか見ていきましょう。

4.4 java.lang.String クラス

さきほどの API リファレンスで「java.lang」パッケージから「String」クラスをクリックしてみましょう。これは文字列を扱うためのクラスであり、main メソッドの引数のところで我々はもう目にしてきました。

この String クラスを使いながら、「オブジェクト」「クラス」「インスタンス」「メソッド」「フィールド」「コンストラクタ」などのオブジェクト指向特有の概念に慣れていきましょう。Web サイトからサンプル TestString.java の例を見てください。

まず、4 行目で使われているメソッド「valueOf」は API リファレンスによると「public static」なメソッドであり、先ほどの Math クラスの例と同じように、java.lang の機能なので、「String.valueOf」として呼び出すだけで、引数の double 型の数字を文字列型に直してくれているようです。

一方、それ以降では static ではないメソッドが使われています。まず、String クラス「型」の「変数」str1、str2、str3 が定義されているようです。これらはもはや「型」ではなく、String クラスの実体例であり、String クラスの「インスタンス」と呼ばれます。つまり、変数 str1、str2、str3 が保持しているものは String クラスのメソッドが使える一つ一つ別々の実体 (オブジェクト) です。

オブジェクト指向とは「部品化の作法」と言いましたが、本来は「部品」というのは同じ型のものが複数必要なものです。組み立て机の説明書にネジ A が 4 個、ネジ B が 12 個、などとあるのを想像してください。ネジ A の型 (仕様) を決めるのが「クラス」で、実際の 4 個のネジ A がその「インスタンス (実体)」ということになります。ネジ A が 4、ネジ B が 12、と「部品化」しておけば、16 本のネジを 2 つの仕様を決めることで製造できるというわけです。

たとえば、13 行目、14 行目では、String クラスのインスタンスである str2 と str3 の両方に 5 文字目の文字を問う同じメソッド charAt(5) が呼ばれていますが結果が異なります。先ほどのネジの例えで言えば、ネジ A を 4 本作って、その 1 本を赤く着色したとしても、他 3 本は元の色のままだということです。つまり、このメソッド charAt は Math.sqrt や String.valueOf と違って、static(静的)=「引数が同じならいつも同じ挙動」ではありません。実際、String.chatAt(5) と仮に言えたところで「何の 5 文字目??」となるでしょう。

ただし、String.charAt(“abcdb”,3) みたいな「関数」のような形として public static にも定義はできることはできます。こうしないで、str2 や str3 のインスタンス自体に charAt(5) と聞いて答えが返ってきているところがミソなのです。「部品化」された同じメーカーの「エアコン」はいろいろな自動車にインスタンスとして搭載されますが、on スイッチは個々の自動車のドライバーが押すものであって、メーカーがコントロールするものではないのです。

これが「関数」みたいなものを「メソッド」と呼ぶ理由です。メソッドは (static なものをのぞき) それぞれの実体に対して動作を指定するものです。

4.5 java.lang 以外の例: java.util.Calendar

今までは何もしなくても使える java.lang パッケージのクラスばかり取り上げてきました。ここでは java.lang 以外の例として、java.util.Calendar クラスを取り上げることにしましょう。まず、Web サイトから「TestCalendar1.java」「TestCalendar2.java」の 2 つのプログラムを実行し、API リファレンスを頼りに、内容を確認してみてください。Java は文法は簡単ですが、いろいろな既存のクラス (部品) を利用するので、このように API リファレンスを参照し解読する作業がとても大事になります。

注意！: java.util.Calendar では月に対応する数字がひとつずれています。例えば、1 月は「0」、2 月は「1」などとなります。なので、プログラム中では定数を使ったほうがよいかもしれません。API リ

ファレンスで java.util.Calendar を調べると、最初のほうの「フィールド」に「static int」で定義された「英数大文字」の「定数」がいろいろとあるのがわかります。ここで、例えば「1 月」を参照するには「JANUARY」を見ます (正確には「java.util.Calendar.JANUARY」で参照するか次の説明のように「import」する)。クリックすると「関連項目」に「定数フィールド値」というのがあり、どういう値で定数が定義されているかみることができます。「定数フィールド値」のリストを見てみると、「JANUARY」の場合「0」となっているのがわかると思います。

import 文

TestCalendar1.java では java.lang 以外のクラスを用いているため、毎回、java.util.Calendar や java.text.SimpleDateFormat など、パッケージ名も含めた長い名前を指定しています。これでも問題はありませんが、名前が長いため、ソースコードの可読性が下がっており、見辛いことは否めません。

TestCalendar2.java では最初に「import 文」により import したクラスについてはクラス名のみで利用できるようになっています。あるパッケージのクラスをすべて import するには、たとえば、「import java.util.*;」等とします。java.lang だけはあまりに基本的な機能が多いため import なしで使えるようになっていますが、Java では java.lang 内のクラスは自動的に全て「import java.lang.*;」と import されているとも解釈できます。

new 演算子とコンストラクタ

もう一つ、SimpleDateFormat のインスタンスを「new」で作成しています。クラスのインスタンスを生成する際にはこのように「new」で新しいインスタンスを生成することを指示します。

ここで、new するときに、クラス名の中に引数を指定しています。これはインスタンスに挙動を指示するための「メソッド」ではなく、「コンストラクタ」と呼ばれるもので、クラスのインスタンスが最初に作成されるときのみに行われる処理が付与されているものです。SimpleDateFormat クラスを使う際には、いずれにせよフォーマットを指定するだろうから、作成時にどのようなフォーマットかを指定できると便利です。このようにインスタンスを作る際に初期化を行うことが「コンストラクタ」の役割です。なお、String クラスは利便性のため

```
String str = "abc";
```

という特殊な表記法ができるため、直感的にはコンストラクタがわかりづらいのですが、API リファレンスで java.lang.String クラスのところにあるように、コンストラクタで明示的に書けば

```
char data[] = {'a', 'b', 'c'};
String str = new String(data);
```

と同じです (ただ、毎回こう書くのは面倒くさいですよね)。

4.6 標準入力

さて、上のような課題に取り組むと、C 言語で scanf を使ったようにプログラム中に書かれた値について計算するだけではなく、キーボードから値を読み込んで計算する方法を使いたくなると思います。ここでは以下のように java.util.Scanner を用いる方法を記載しておきます。API リファレンスで java.util.Scanner クラスで使えるメソッドを確認してみてください。Java ではこの処理を行う方法がい

くつもあるのでもちろん API リファレンスや書籍を参考にして得た他の方法を用いても構いません。なお、標準入力だけではなくファイルを含めた入出力について付録 4 で解説しているので必要に応じて参考にしてください。特に入出力は外部のファイルや操作が関与するためエラーが起きやすいタスクであり、try-catch 構文による例外処理がよく併用されます。

```
1  import java.util.Scanner;
2  public class TestInput {
3      public static void main(String[] args) {
4          System.out.println("Input Your Name:");
5          Scanner scanner = new Scanner(System.in);
6          String name = scanner.nextLine();
7          System.out.println("Input Your Age:");
8          int age = scanner.nextInt();
9          System.out.printf("Your Name=%s (Age=%d)\n",name,age);
10         scanner.close();
11     }
12 }
```

ソースコード 1.3: TestInput.java

注意！`System.in` にひもづいた `java.util.Scanner` の close

```
Scanner s1 = new Scanner(System.in);
int i = s1.nextInt();
s1.close();
Scanner s2 = new Scanner(System.in);
int j = s2.nextInt();
s2.close();
```

上記は一見問題なさそうに見えますが、エラーになります。s1 の close() の段階で、s1 にひも付いている System.in も close() されてしまうので、標準入力が閉じてしまうためです。これに対する最も簡単な解決法は「System.in にひも付いた Scanner はもう標準入力が要らないのでない限り close しない」というものです。(しかも System.in, System.out, System.err は reopen できない) もっとちゃんとやるには、

- System.in を保護するラッパークラス (そのクラスの close が呼ばれても System.in の close が呼ばれないように明示的に書く) を作成し、Scanner はそれにひも付ける。
- System.in にひも付ける Scanner を複数作らないような設計とする。

などが考えられますが煩雑なので演習でこの問題に出会ったら close しない方針で構いません。Java ではその Scanner を保持するオブジェクトがどこからも参照できなくなったら自動的にこうしたオブジェクトは片付けられる仕組みがあります。

4.7 非標準クラスライブラリ

これまで Java に標準で準備されているライブラリのさまざまな API について学習しました。これらの標準ライブラリは様々な機能を網羅していますが、世の中には優れた機能を持つ非標準ライブラリもオープンなものから商用のものまで様々に存在します。例えば Java はよく Android アプリの開発に用いられますが、Android の機能にアクセスするクラスライブラリなども非標準です。実際には、自分で過去に作った部品も誰か友達が作った部品も就職した会社のプロジェクトで用意されている自社部品も、Java からしたら「非標準」なので、このように非標準のクラスを使う場合も多いです。

Java のライブラリは商用・無償を問わず星の数ほどありますが、例えば、以下のようなライブラリがあります。演習では非標準のライブラリを扱いませんが、基本を付録 5 に載せているので余裕があれば確認してください。

- Guava (Google Core Libraries for Java)
  `https://github.com/google/guava/`
- Apache Commons Lang
  `https://commons.apache.org/proper/commons-lang/`
- charts4j
  `https://code.google.com/p/charts4j/`

5 オブジェクト指向入門 (1)：クラスを作ってみる。

さて、ここまで、クラスライブラリとして用意された様々な便利な「クラス」を「部品」として用いてプログラミングをすることで C 言語単体でできるよりも幅広いプログラムが作成できることを見てきました。以降では、これらの「部品」を作るほうの立場で考えてみましょう。本演習のテーマである「オブジェクト指向プログラミング」では、プログラミングをこれらの「クラス」の設計と他のクラスを上手に利活用したプログラミングを安全に効率よく行うことができます。

本節の作業と課題はまとめて末尾の 5.6 節に記載しています。

5.1 データ構造としての「クラス」

詳細に入る前に次の例を実行してみましょう。クラス A は main メソッドを持っていないことに注意してください。この例で見られるように基本的には一つの「クラス」は一つの java ファイルに記述されます。従って、Java 開発では複数のファイルを管理する状況が一般です。

まず、class は「フィールド (値)」と「メソッド (操作)」を保持するデータ構造と見なせます。下記 Test0.java の x と y は同じクラス A のインスタンスですが、print() メソッドの結果は異なります。

```
enum Hand {ROCK, PAPER, SCISSORS}
```

ソースコード 1.4: Hand.java

```
public class A {
    boolean b;
    int i;
    Hand h;
    public void print(){
        System.out.println(b);
        System.out.println(i);
        System.out.println(h);
    }
}
```

ソースコード 1.5: A.java

```
public class Test0 {
    public static void main(String[] s){
        A x = new A();
        x.b = true;
        x.i = 30;
        x.h = Hand.ROCK;

        A y = new A();
        y.b = false;
        y.i = 40;
        y.h = Hand.PAPER;

        x.print();
        y.print();
    }
}
```

ソースコード 1.6: Test0.java

また「フィールド」にデフォルト値を指定するなどの処理は「コンストラクタ」を定義することで次の例のように実現できます。new する際にインスタンスが生成されることにも注意してください。

```
public class B {
    boolean b;
    int i, j;
    Hand h;
    public B(){ // コンストラクタ
        this.b = true;
        this.i = 10;
        this.j = 20;
        this.h = Hand.SCISSORS;
    }
    public void print(){
        System.out.println(b);
        System.out.println(i);
        System.out.println(h);
    }
}
```

ソースコード 1.7: B.java

```
public class Test1 {
    public static void main(String[] s){
        B x = new B();

        B y = new B();
        y.b = false;
        y.i = 20;

        x.print();
        y.print();
    }
}
```

ソースコード 1.8: Test1.java

クラスは Java においては「型」のようにも捉えられます。たとえば下記のように、あるクラスのインスタンスを要素に持つ配列を定義できますし、インスタンスを return するなども同様に可能です。

```java
public class C {
    // 変数を「カプセル化」
    private boolean flag;
    private int cnt1, cnt2;
    // コンストラクタ
    public C(){
        this.flag = true;
        this.cnt1 = 0;
        this.cnt2 = 0;
    }
    // メソッド
    public void call(){
        System.out.println("call");
        if(flag) this.cnt1++;
        else this.cnt2++;
    }
    public void lost(String msg){
        System.out.println(msg);
        this.flag = false;
    }
    public void print(){
        System.out.println(this.cnt1
            );
        System.out.println(this.cnt2
            );
    }
}
```

ソースコード 1.9: C.java

```java
public class Test3 {
    public static void main(String[] s){
        C[] obj = new C[2];

        obj[0] = new C();
        obj[1] = new C();

        for(int i=0; i<7; i++)
            obj[0].call();

        obj[0].print();
        obj[0].lost("changed");

        for(int i=0; i<3; i++)
            obj[0].call();

        obj[0].print();
        obj[1].print();
    }
}
```

ソースコード 1.10: Test2.java

なお、複数のファイルを javac でコンパイルする場合は関連するファイル名をすべて引数に与えて次のようにコンパイル、実行します。そうすると 3 つのクラスに対応する class ファイルが生成されます。一旦生成されていれば例えばもし A だけに変更を加えた場合、コンパイルは当該クラス (A.java) だけで十分です。

```
$ javac Hand.java A.java Test0.java
$ java Test0
```

ここでは、Eclipse で行う際の注意について簡単に述べておきます。Eclipse では一つの「プロジェクト」で関連する複数のソースコードのファイルを管理します。プロジェクトに加えられているソースコードは対応する class ファイルがなければ実行時に自動的に生成されます。ただしプログラムの起動はなんらかのクラスの main メソッドを呼ぶことで行うので「実行構成」のところで「実行」する際、どのクラスの main メソッドを呼ぶのかを指定しておく必要があります。ソースファイルが一つしかない場合は問題になりませんが、複数ある場合は注意してください (p.9 参照)。

5.2 ケーススタディ：モンテカルロ・シミュレーション

それでは、オブジェクト指向開発での「部品」である「クラス」を使う事例として、これ以降では次の問題を考えることにします。モンテカルロ・シミュレーションとは乱数を用いるシミュレーションであり、分かりやすさのため「じゃんけん」を題材に考えますが、いろいろな機械学習や統計学の確率モデルや、不確定な因子を伴うシミュレーションに共通するものです。

1. A さんと B さんでランダムに「じゃんけん」を行い、その勝ち負けの変化をトレースしたい。例えば、10 回じゃんけんを行うと何回くらい「引き分け」になるだろうか? あるいは、B さんはランダムではなくて、A さんに負けたときは、次に先ほど A さんが出した手を出すようにするとすると、この率は変化するだろうか? あるいは B さんは「グーしか出さない」とした場合はどうなるだろうか?
2. A さんと B さんと C さんで 3 人で「じゃんけん」を行い、同様なシミュレートを行いたい。
3. n 人でじゃんけんを行ったとき「引き分け」が起こる率はどれくらい上がっていくのだろうか? 3 人、5 人、7 人、9 人、で 100 回じゃんけんを行ってみてこの値をシミュレートしたい。
4. n 人でじゃんけんをして、勝者 1 人を決めたいとすると勝敗が決まるまでに何回くらいじゃんけんが必要なのだろうか? 人数が多くなると引き分けが増えるため、「もし場の手が 3 種類のときは手が多いグループの人たちが勝ちとする」という引き分け解消ルールを導入するとこの回数はどのようになるだろうか?

さて、これらをプログラムを作成して確かめたいとき、どのようにプログラミングを行っていけば良いでしょうか。まずは上記を C 言語で手続き的にプログラミングするにはどのようにするか考えてみてください。

5.3 オブジェクト指向による考え方

上の内容はじっくり考え始めると、なかなか大変そうではないでしょうか。しかし、もし人を任意の人数集めることができるならば、実際にじゃんけん大会を開いて、上記のことを確認してみることは簡単そうです。例えば、1 番目の点であれば、実際に二人集めてきて、10 回じゃんけんしてもらい、それを記録しておけば良さそうだし、後半の B さんが手を変えるケースでも A さんに「ランダムに手を出してください」、B さんに「このような規則で手を出して見てください」と伝えておき、もう一度 10 回じゃんけんしてもらえば良いだけです。2 番目の点は 3 人集めてくればよく、3 番目のときでも n 人を実際に集めてきて、じゃんけんしてもらえば良いだけそうです。

オブジェクト指向はプログラムを「クラス」とその「インスタンス」というオブジェクト部品で設計します。したがって、「ランダムにじゃんけんを行うじゃんけんプレイヤー」のクラスを設計し、これらのインスタンスを 2 つ生成して、実際に対戦させて見れば良さそうです。対戦結果を判定するために「審判クラス」も必要でしょうか。

まず Web サイト RandomJankenPlayer.java と Hand.java のように「ランダムじゃんけんプレイヤー」のクラスを見てみます。ここではじゃんけんの手グー (Rock)、チョキ (Scissors)、パー (Paper) を表現する Hand というクラスを enum として用意しています。enum(列挙型) の挙動は例を見れば分かると思いますが、グー=0, チョキ=1, パー=2 とかすると読みづらいし、どれがどれだったか途中で忘れてしまうと面倒なことになりそうなので明示的に名付けしておく感じです。

この単純な例を使って、まず、いくつかの用語について述べておきます。

クラス これ自体が部品 RandomJankenPlayer の雛型 (設計図) で「クラス」と呼ばれます。

インスタンス java で実行した場合に実行される main メソッドでは、この雛型に基づいて、player1 と player2 の 2 つの実体オブジェクトを生成しています。これらをこのクラスの「インスタンス」と呼びます。

メソッド このクラスの部品は命令 showHand() を備えています。これらはこの部品のインスタンスがどのように振る舞うかを規定しているもので「メソッド」と呼ばれます。メソッドの振る舞いは同じクラスであってもインスタンスごとに異なります。player1 や player2 の showHand() が各々独立であることを確認してください。

フィールド 実際にこれらをゲーム画面に表示することを考えるとプレイヤの名前くらいは必要でしょうか。このクラスは String name という文字列変数を備えており、このクラスのフィールドと呼ばれます。フィールドも勿論インスタンスごとに異なります。player1 と player2 は異なる name を持っていることを確認してください。

コンストラクタ name を設定するメソッドはなく、ここではクラス名と同じで返り値のない RandomJankenPlayer(String name) という特殊メソッドを定義しています。これはこのクラスのインスタンスが生成されるときにのみ実行されるメソッドでこのクラスの「コンストラクタ」と呼ばれます (construct=作る)。主にクラスの初期化や準備を記述します。

アクセス修飾子 メソッドやフィールドには public というキーワードが付与されています。これを「アクセス修飾子」と言い、各々へのアクセス制御を規定します。

オブジェクト指向では部品の仕様を「クラス」として定義して、それを雛型として「インスタンス」オブジェクトを必要な数生成して、それらのメソッドを呼び出すことによって、処理を行います。これは、最初の「じゃんけんをシミュレートしたい」という要求に対して、人を実際に呼び出せば簡単にできるのならば、各々の人やら審判やらの構成部品をプログラミングして整備することで、それらのオブジェクトをメソッドで操作しながら、やりたいタスクを実行することに相当します。

そのためオブジェクト指向は単にプログラミングの道具としてだけにとどまらず、最近ではソフトウェアにやらせたい複雑なことを整理する「モデリング」のための道具としても活用されています (要求分析などソフトウェア開発の上流工程で UML 図でもソフトウェア要求を記述する、など)。

5.4 カプセル化とアクセス修飾子

まだ、じゃんけんするために二人 (Yamada さんと Suzuki さん) を呼び出して 10 回手を出させただけなので、シミュレートするためには勝敗判定を行ったり、勝ち負けを記録したり、引き分け回数を記録したりと追加が必要です。

しかし、その前にオブジェクト指向の 3 本柱の一つ「カプセル化」の話をしておきましょう。今回のクラスではフィールドに「public」というアクセス修飾子が付いています。フィールドやメソッドは「アクセス制限」ができるのですが、この例で用いた「public」はもっともアクセス制限が緩い (アクセスを公—public—にしている) 状態です。

アクセス制御	名前	アクセス修飾子	アクセス可な範囲
緩い	public	public	すべてのクラス
	protected	protected	自分と同じパッケージに属するクラスか自分を継承した子クラス
	package private	(何も書かない)	自分と同じパッケージに属するクラス
厳しい	private	private	自分自身のクラスのみ

ここでは「public」のままで何の問題もなさそうですが、オブジェクト指向はそもそも「大規模な」ソフトウェアを「複数人で」作るための方針という点を少し考えておきましょう。このプログラムではコンストラクタで名前を設定しており、一旦設定された名前を後から変える必要はなさそうです。これをゲーム画面つきで完成させることを想像して考えるとプレイヤがゲーム開始時に設定したこのキャラクタ名を後から変更する必要があるかどうかということになります。そのような機能を追加するにせよ、キャラクタ生成時とその機能の発動時以外にこの名前フィールドを変更することはなさそうです。

しかし、「public」のままだと、じゃんけん 3 回目で「Suzuki」を「Yamada」に書き換えられてしまいます。そんなことはしないようにと覚えておけばよいじゃないかと言われればその通りなのですが、もっと複雑で大規模なソフトウェアを「複数人で」作っている状況ではそのようなことを「仕組み」として許さないようにしておかないと、訳のわからない深刻なバグの原因になりかねません。大規模なプロジェクトで参加者全員が すべての ソースコードの隅から隅まで読み設計意図を理解しておかなくてはならないような設計はどう考えても問題があるでしょう。アクセス修飾子による制限はそのための大切な仕組みです。大切なビデオ録画を (大切さをわかっていない) 家族に勝手に上書きされたり消されたりしてしまわないよう「上書き禁止」にしておきたい状況のようなものです。name 変数を private にしておけばこのクラスのメソッド以外からこの値が書き換えられる心配は「仕組み上」無くなります。家族に「あの録画ビデオは大事だから消さないでね」とお願いするのと、仕組みとして「上書き禁止」をセットしておくのとどちらが安全でしょうか。

この点が大切なことはオブジェクトが「部品」であるということから想像できるかもしれません。家を建てることを想像してください。家を建てる際には、電子配線やら水道の配管やらは家の住人からは普通隠されていると思います。それらがむき出しになっていると見た目が悪いということもあるでしょうが、電気配線やら水道配管を住人が普通に触れられるとすると、生活しているうちに何かの拍子で配線を切断してしまったり、配管を壊してしまって、家の機能全体に支障をきたすリスクがあります。むき出しになっていたほうが、業者の人は不良があった際に直しやすいですが、住人にとってはリスクこそあれ、嬉しいことはほとんどありません。部品は機能の発現に必要な部分だけ出しておき、中身の部分はできるだけ隠蔽すべきなのです。このことを**「カプセル化」**(情報をカプセルに入れて隠蔽してしまうイメージ) と呼びます。カプセル化はオブジェクト指向では部品の「機能」と「使い方」だけが周知されるため、ユーザは中身の仕組みを知らないという程で設計する必要があるため非常に重要な概念です。

5.5 カプセル化、アクセサ、Getter/Setter

オブジェクト指向プログラミング言語の背景にある考え方の一つは、関連する機能に関与するデータと処理をフィールド・メソッドとして、クラスという「いれもの」に収容できるところにあります。このクラスという「いれもの」に入れたフィールドとメソッドをクラスの外からの干渉を受けないよう保護する仕組みが**「カプセル化 (encapsulation)」**です。

そのため、まず「private String name;」に直しておきましょう。これでこの部品の外からのこのフィールドへアクセスを遮断することができます。アクセスしようとするプログラムを作るとそもそもコンパイル時にエラーになります。ただし、じゃんけんの手を表示しているところで「player1.name」「player2.name」などと呼び出しているところも適切に直す必要があります。フィールド値を private にしてここではその値を読み出すだけのメソッド getName() を定義することにしましょう。ここではコンストラクタ以外でこのフィールドの値を書き換える必要がないので、必要ありませんが、もしそのような機能を提供したいなら、setName(String name) も作れば良いでしょう[6]。このようなメソッドはアクセサ (accessor) と言い、フィールド値を取得するメソッド・設定するメソッドをそれぞれ getter、setter と呼ばれます。

[6] クラスのインスタンスが持つフィールドの name は this.name で参照します (p.30 参照)。なので、引数にも同じ name という変数名が使えます。

getter や setter を通さないアクセスを仕組みとして禁止しておくことで途中で値が意図せず書き換えられてしまうことを防ぎ、バグの少ないプログラム作成の機能を提供しています。例えば、この場合のように Read only のフィールドを提供できます。

このように Getter/Setter アクセサを介してクラス内のフィールドへアクセスするようにしておく利点は情報の保護だけではありません。例えば、呼び出し側は必ずアクセサを経由するようにしておけば、クラス内でのフィールド値の格納の仕方を変えても呼び出し側のコードを変更する必要がありません。たとえば、アクセサの中でフィールド値が参照されるたびに参照回数をカウントしたり、setter において必要となる計算をキャッシュしておいたり、値が望ましいものになっているのかの正当性チェックを加えたりと、クラス内部の処理を加える自由度も提供しています。また、後述する「多態性」を利用することができるようになります。

Getter および Setter の名前は、慣習的に get(set) + 「フィールド名の先頭文字を大文字にしたもの」となっています。この定義は定型的なため、Eclipse にはこれらを自動生成する機能があります。ここでは、以下の手順に従って、Eclipse の機能を用いて Getter と Setter を定義します。

1. 編集中のソースファイルを Sample.java とし、入力カーソルを Sample クラスの定義範囲の中に入れておく。

2. メニューから「ソース」→「Getter および Setter の生成」を選択する。

3. フィールドの一覧が表示されるので、ゲッターとセッターを生成したいフィールドにチェックを入れる。変数の左側の三角形をクリックすることで、ゲッターとセッターのどちらかのみを生成するためのチェックボックスを表示する事ができる。ここでは全てのフィールドについて、ゲッターとセッターの両方を生成するので、右上の「すべて選択」を選択し、「OK」を選択する。

5.6 作業と課題

作業 1.8

Web サイトからサンプル Hand.java と RandomJankenPlayer.java をダウンロードし、実行せよ。2 つのファイルを一つのプロジェクトに import し、RandomJankenPlayer の main メソッドが実行されるように設定すること。また、RandomJankenPlayer の中身を読み、ランダムにグー・チョキ・パーの 3 択を行う部分についてよく確認しておくこと。

作業 1.9

RandomJankenPlayer.java のクラスが持つフィールドをカプセル化せよ。名前のフィールドなので「ReadOnly で保護する」ために、コンストラクタで初期化しアクセサは Getter のみ提供することとせよ。

さて、クラスのメンバであるフィールド値を private で保護し、アクセサを定義したら、いよいよ、審判クラスを定義することでまず 2 人対戦のじゃんけんのシミュレートができるようにしましょう。

課題 1.5

次の「ここを埋める」の箇所を適切に記述し、審判クラス Judge を完成させよ。

```
public class Judge {
    private String name;
    private RandomJankenPlayer player1, player2;
    public Judge(String _name){
        name = _name;
    }
    public void setPlayers(RandomJankenPlayer _player1,
                           RandomJankenPlayer _player2){
        // ここを埋める
    }
    public void play(int n){
        Hand hand1, hand2;
        int win1=0, win2=0;
        int lose1=0, lose2=0;
        int draw1=0, draw2=0;

        // ここを埋める

        System.out.println("Player1 : "+player1.name);
        System.out.println("Player2 : "+player2.name);
        System.out.println("Judge   : "+name);
        System.out.println();
        System.out.println("Results: "+n+" games");
        System.out.println(player1.name+" "+win1+" win, "+lose1+" lose, "+draw1
            +" draw");
        System.out.println(player2.name+" "+win2+" win, "+lose2+" lose, "+draw2
            +" draw");
    }
    public static void main(String[] args) {
        try{
            int num = Integer.parseInt(args[0]);
            RandomJankenPlayer player1 = new RandomJankenPlayer("Yamada");
            RandomJankenPlayer player2 = new RandomJankenPlayer("Suzuki");
            Judge judge = new Judge("Sato");
            judge.setPlayers(player1,player2);
            judge.play(num);
        }catch(Exception e){
            System.out.println("this requires an integer argument.");
        }
    }
}
```

ソースコード 1.11: Judge.java の雛形

ただし、この時点では 作業 1.9 のカプセル化の他には RandomJankenPlayer クラスへは変更を加えないものとし、上記の実行結果が以下のように表示されるようにせよ。また、play メソッドの中で Judge のインスタンス自体の name を参照するには this.name とすれば良い。

```
$ java Judge
This requires an integer argument.
$ java Judge 100
Player1 : Yamada
Player2 : Suzuki
Judge   : Sato

Results: 100 games
Yamada 35 win, 33 lose, 32 draw
Suzuki 33 win, 35 lose, 32 draw
```

課題 1.6

じゃんけんを 10 回やると平均何回くらい勝つのかに関する実データを集めたい。課題 1.5 プログラムを修正し、10 回じゃんけんを m セット行った時、各プレイヤの勝ち数、負け数、引き分け数の平均値を出力するようにせよ。m を 1,10,100,1000 と上げて、平均値が一定値に近づく様子を調べよ。

課題 1.7

プレイヤークラス RandomJankenPlayer、審判クラス Judge を書き換えて、以下の挙動をするプログラムを作成せよ。

- プレイヤーを Yamada, Suzuki, Tanaka の 3 人にする。審判クラス Judge はフィールドとして (2 人ではなく)3 人の RandomJankenPlayer インスタンスを持つようにする。)
- 対戦は毎回プレイヤーの中からランダムに 2 名を選び勝ち負けが出るまで対戦させ、この 1 対 1 マッチを n 回繰り返す。つまり 3 名の勝ち数の総和が必ずゲーム数 n と同じになるようにする。また、勝ち負けが出るまでの引き分け (draw) も記録しておく。
- RandomJankenPlayer クラス自身が勝ち数・負け数・引分け数の 3 つの値を持つようにフィールドを追加しコンストラクタで初期化する。各マッチごとに各々のプレイヤインスタンスの持つ値を変更するようにする。
- Judge では毎ゲームで 3 人からランダムに 2 人を選択し、勝ち負けが出るまで対戦し、引分けの際は両方のプレイヤの引分け数を+1 し、勝ち負けが出た場合、勝ちプレイヤの勝ち数を+1 し、負けプレイヤの負け数を+1 する。結果として、例えば 20 ゲーム実施した場合、以下のような出力が出るようにすること。

```
Player1 : Yamada
Player2 : Suzuki
Player3 : Tanaka
Judge   : Sato

Results: 20 games
1 vs 2 : 5 games
1 vs 3 : 7 games
2 vs 3 : 8 games

Yamada 6 win, 6 lose, 3 draw
Suzuki 5 win, 8 lose, 11 draw
Tanaka 9 win, 6 lose, 8 draw
```

課題 * 1.8

課題 1.7 では 3 人であったものを任意の m 人で作動するように拡張せよ。実際の main 部ではプレイヤを Yamada, Suzuki, Tanaka, Takahashi, Yamamoto の 5 人 ($m = 5$ の例) で作成し、100 ゲームの結果を同様なフォーマットで表示するものとする。

- 配列もしくは ArrayList などのデータ構造を用いる必要がある。
- m 人からランダムに 2 人を選び対戦させる必要がある。例えば 5 人の例では 0 から 4 の整数乱数を重複しないで二つ選ぶ。その方法としては例えば、まず num1 として、0 から 4 の整数乱数を一つ選ぶ。次に、num1 以外の残る 4 つの整数からもうひとつをランダムに選べば良い。つまり、$0, \cdots, \mathrm{num1}-1, \mathrm{num1}+1, \mathrm{num2}+2, \cdots, 4$ から一つ選べばよい。従って num2 は 0 から 3 までの整数乱数をとり、もし num1 の値と等しいか大きければ +1 すれば良い。

```
// 0 から 4 の整数から重複しない 2 つを選ぶ
java.util.Random gen = new java.util.Random();
int num1 = gen.nextInt(5);
int num2 = gen.nextInt(4);
if(num1<=num2) num2 += 1;
```

- 課題 1.7 の出力のように、5 人の誰と誰が何回対戦したかを出力する場合は、2 つの数字のペアの生起を記録する必要がある。例えば、2 つの数字のペアを java.util.HashMap を用いて、以下のように文字列として格納してしまう方法がある。あるいは、Apache Commons Lang に「Pair」というそのままズバリのクラスがあるので余裕がある人はそれを利用すると良い。

```
java.util.HashMap<String,Integer> pairs = new
java.util.HashMap<String,Integer>();

// ここでペア (num1,num2) を生成したとする。

String pair;
if(num1<num2){
    pair = String.format("%d vs %d",num1+1,num2+1);
}else{
    pair = String.format("%d vs %d",num2+1,num1+1);
}
int count = 1;
if(pairs.containsKey(pair)){
    count += pairs.get(pair);
}
pairs.put(pair,count);
```

6 オブジェクト指向入門 (2)：継承と多態性

これまでの作業・課題で、クラスの定義の仕方について演習し、２人対戦じゃんけんシミュレータのたたき台ができあがったので、再度、5.2 節を読み、本来やりたかった内容をまず確認しておきましょう。この節ではオブジェクト指向の概念を利用して、じゃんけんプレイヤの戦略を増やしてみます。

オブジェクト指向言語の定義は色々と有りえますが、一般には、**カプセル化、継承、多態性**の三要素を持つことが基本となっています。このうち、カプセル化については既に解説してきました。本節では、残っている花形機能である**継承**と**多態性**について見ていきます。

本節の作業と課題はまとめて末尾の 6.7 節に記載しています。

6.1 継承と多態性

今までは RandomJankenPlayer のみでの対戦でしたが、当然 RandomJankenPlayer だけではなく、色々とじゃんけんの戦略を変えて複雑な状況で勝率や引き分け率がどのように変化するかを調べたいということがあります。そのためにも、ランダムではない振る舞いをするじゃんけん Player を色々と定義しておきたいところです。じゃんけんの戦略も色々と考えられると思いますが、ここでは以下の 2 パターンを例にして、どのようにランダム以外のじゃんけんプレイヤーを定義していけば良いか、その設計にオブジェクト指向がどのように関わっているかを見ていきましょう。

(a) B さんは単なるランダムではなくて、A さんに負けたときは、次に先ほど A さんが出した手を出すようにするとすると、この率は変化するだろうか?

(b) B さんは「グーしか出さない」とした場合はどうなるだろうか?

ここで、(a) 型のじゃんけんプレイヤーを JankenPlayerTypeA、(b) 型のじゃんけんプレイヤーを JankenPlayerTypeB として、各々クラスとして実装することを考えてみましょう。戦略 showHand() の方針が異なるだけで、じゃんけんやセットアップに必要なそれ以外の部分は基本的に、ランダムであっても何か特別な戦略であっても変わりません。そこで RandomJankenPlayer クラスのコードをもとにして、新しいクラス JankenPlayerTypeA、JankenPlayerTypeB を用意することを考えます。すると、勝敗の判定を行っていた Judge クラスの次の部分

```
private RandomJankenPlayer player1, player2;
public void setPlayers(RandomJankenPlayer player1,
                       RandomJankenPlayer player2){
  :
}
```

のフィールドおよびメソッドも修正する必要が生じます。

6.2 メソッドのオーバーロード

Java では引数の型や数が違う場合、コンストラクタを含めたメソッドを同じ名前で複数定義することができます。これをメソッドの「オーバーロード (overload)」と言います。ここで、C 言語においては次のコードにおいて、関数 add2 の名前を add とするとエラーになることを思い出しておきましょう。

```c
#include<stdio.h>
int add(int x, int y){
  return x+y;
}
double add2(double x, double y){
  return x+y;
}
int main(){
  printf("%d\n",add(12,29));
  printf("%f\n",add2(12.80,29.12));
}
```

ソースコード 1.12: C 言語では例え引数の型が違っても同じ名前の関数は定義できない

しかし、Java では、例えば

```
public void setPlayers(RandomJankenPlayer player1,
                       RandomJankenPlayer player2){ ... }
public void setPlayers(JankenPlayerTypeA  player1,
                       RandomJankenPlayer player2){ ... }
public void setPlayers(RandomJankenPlayer player1,
                       JankenPlayerTypeA  player2){ ... }
```

のように引数の型や数が異なれば同じ名前をメソッド名に用いることができます。

従って、引数 player1 および player2 の各々が RandomJankenPlayer か JankenPlayerTypeA か JankenPlayerTypeB になるようにすべての組合せについてオーバーロードしたメソッドを用意し、対応するフィールドを管理すれば、JankenPlayerTypeA や JankenPlayerTypeB であっても、setPlayers(player1,player2) と同じ名前形式でメソッドに渡すことができます。

しかし、この方法はそれほど見通しが良いとは言えないでしょう。例えば、JankenPlayerTypeC や JankenPlayerTypeD など、新しいじゃんけん戦略を加えるごとにすべての組合わせについてメソッドを定義しなければいけません。このような有りえそうな追加機能にすら対応が困難になるということは設計がまずそうです。

ここで本節のテーマの一つである**「継承 (inheritance)」**を用いることでこのような局面にうまく対処できることを見ていきましょう。

6.3 継承とメソッドのオーバーライド

既に存在するクラスを再利用して、機能の拡張・変更を行った新たなクラスを定義するための仕組みの一つに、**継承** *(inheritance)* があります。継承の特徴は元のクラス（**スーパークラス**）が持っていた機能（フィールド、メソッド）を完全に引き継ぎ、機能の追加・変更のみを行うという点にあります。このため、スーパークラスに存在していたフィールドやメソッドは継承によって拡張されたクラスにおいても確実に存在するため、継承によって新たに作成したクラス（**サブクラス**）はスーパークラスの機能を包含しています。この特徴により、あるクラスを引数に取るメソッドに、そのサブクラスを渡したり、あるクラスの配列にそのサブクラスの要素を混在させたりすることが可能になります。

あるクラス BaseClass を継承し、新しいクラス NewClass を定義するには、extends というキーワードを用いて

```
public class NewClass extends BaseClass{
    ...
}
```

のように書きます。ここでは、Eclipse のクラス生成機能を利用して、以下の手順で RandomJankenPlayer クラスを継承した JankenPlayerTypeB クラスを定義します。

1. メニューから「ファイル」→「新規」→「クラス」を選択する。
2. 「名前」の欄に「JankenPlayerTypeB」と入力する。
3. 「スーパークラス」の右の「参照」を選択します。
4. 型の選択欄に「RandomJankenPlayer」と継承したいクラスの名前を入力すると、名前が前方一致するクラスが表示される。ここでは自分で定義したデフォルトパッケージの「RandomJankenPlayer」を選択して「OK」を選択する。
5. 「完了」を選択する。

これにより、JankenPlayerTypeB クラスを定義する新しいソースファイルが作成され、画面に表示されます。この時点では JankenPlayerTypeB クラスで新たに追加・変更される内容はまだ何も書かれていないため、JankenPlayerTypeB クラスは RandomJankenPlayer クラスと全く同じものとなっています。まず、以下のようにメソッド showHand() を定義します。

```
1 public class JankenPlayerTypeB extends RandomJankenPlayer {
2     public Hand showHand(){
3         return Hand.ROCK;
4     }
5 }
```

ソースコード 1.13: JankenPlayerTypeB クラスの定義

これで、RandomJankenPlayer クラスから継承された showHand メソッドが、上記の新しい定義によって上書きされます。このように、サブクラスでスーパークラスのメソッドを定義し直すことを、メソッドの**オーバーライド** (override) と呼びます。RandomJankenPlayer クラスを継承しているため、getName() など RandomJankenPlayer クラスで定義されているメソッドは、新しいクラスにおいても呼び出すことができます。クラスを継承し、必要に応じてメソッドのオーバーライド・追加を行うことで、クラスの機能を容易に修正・追加できることが分かると思います。

次に、RandomJankenPlayer クラスと同様に、インスタンスの生成時にフィールドを初期設定できるよう、コンストラクタを定義します。コンストラクタは通常のメソッドと異なり継承されないため、必要な場合は、新たに定義する必要があります。ここで、コンストラクタを何も定義しなかった場合は、スーパークラスの引数無しコンストラクタを呼び出すだけの、最も単純なコンストラクタが自動的に定義されます。このため、スーパークラスにそのようなコンストラクタが定義されていない場合は、コンパイルエラーとなりますので注意しましょう[7]。

```
1 public class JankenPlayerTypeB extends RandomJankenPlayer {
2     public JankenPlayerTypeB(String name){
3         super(name);
4     }
5     ... //省略
6 }
```

ソースコード 1.14: JankenPlayerTypeB クラスのコンストラクタの定義

[7]「黙的スーパー・コンストラクターは未定義です。 別のコンストラクターを明示的に呼び出す必要があります。」というエラーが出ます。いま、コンストラクタとして String をとるものを定義しているため、引数がなにもないコンストラクタは作られなくなります。こういうクラス X を継承したクラス Y を考えたときクラス Y のインスタンスを作成するとき、親 X をまず引数なしのコンストラクタでインスタンス化する仕様になっています。なので、X に引数なしのコンストラクタが定義されてない場合、このようなエラーが出ます。空メソッド (処理は何もしないメソッド) で良いので定義してください。

ここで使用しているメソッド super は、継承したスーパークラスのコンストラクタを呼び出すためのメソッドです。ここでは、String クラスの引数 name を用いて RandomPlayerTypeB クラスのコンストラクタを呼び出し、名前の初期化をしています。

ここで、super(name) はスーパークラスのコンストラクタの呼び出しを意味しており、これを用いることで、スーパークラスの定義をそのまま用いるようにしています。このように、super キーワードを用いることで、サブクラスのメソッド定義の中でスーパークラスのメソッドを利用することができます。this や super の挙動について以下の例で把握してください。

```java
public class ClassA {
    private String message;
    public ClassA(String message){
        System.out.println("ClassA-Constructor:"+message);
        this.message = message;
    }
    public void printMessage(){
        System.out.println("Message:"+this.message);
    }
}
```

ソースコード 1.15: ClassA.java

```java
public class ClassB extends ClassA {
    private int value;
    public ClassB(String message,int value){
        super(message);
        this.value = value;
    }
    public void printMessage(){
        super.printMessage();
        System.out.println("Value:"+this.value);
    }
    public static void main(String[] args){
        ClassA a = new ClassA("ClassA instance");
        ClassB b = new ClassB("ClassB instance",100);
        a.printMessage();
        b.printMessage();
    }
}
```

ソースコード 1.16: ClassB.java

コンストラクタや private なメンバは継承されません。従って継承したクラスのコンストラクタを明示的に呼び出すにには super() を使います。private なメンバも継承されないため getter/setter を経由するか、protected などアクセス修飾子 (p.21) で制御します[8]。

```
$ java ClassB
ClassA-Constructor:ClassA instance
ClassA-Constructor:ClassB instance
Message:ClassA instance
Message:ClassB instance
Value:100
```

[8]getter/setter も継承先でオーバーライドするなら protected にする必要はあります。

6.4 継承に関する基本的注意事項

Java では、すべてのクラスは直接的あるいは間接的に java.lang.Object というクラスのサブクラスになることを補足しておきます。クラスを定義するときに extends キーワードを用いて明示的にスーパークラスを指定しなかったときは、暗黙的に java.lang.Object がスーパークラスとなります。API リファレンスを見ればクラスの継承関係は明示されています。ここで再度、以下から Java SE8 の API リファレンスを見てみましょう。

`http://docs.oracle.com/javase/jp/8/api/`

上のほうにある「階層ツリー」をクリックして「クラス階層」を見てみると、Java の標準クラスライブラリの各クラスが java.lang.Object から継承されてできていることが分かります。また、java.lang.Object には close() や getClass() などのメソッドが定められており、java.lang.Object を継承している全ての Java のクラスでこれらのメソッドが利用できます。

また、コンストラクタのオーバーロードについても確認してみましょう。java.lang.String クラスの API リファレンスを見ると、このクラスにはたくさんのコンストラクタがありオーバーロードされています。

6.5 多態性 (polymorphism)

継承によって定義した JankenPlayerTypeB に対して、最後に以下のように main メソッドを定義して動作を確認してみましょう。

```
1  public class JankenPlayerTypeB extends RandomJankenPlayer {
2      public JankenPlayerTypeB(String name){
3          super(name);
4      }
5      public Hand showHand(){
6          return Hand.ROCK;
7      }
8      public static void main(String[] args){
9          RandomJankenPlayer player1 = new RandomJankenPlayer("Suzuki");
10         RandomJankenPlayer player2 = new JankenPlayerTypeB("Yamamoto");
11         Hand hand1, hand2;
12         for(int i=0; i<10; i++){
13             hand1 = player1.showHand();
14             hand2 = player2.showHand();
15             System.out.println(i+")"+
16                                "["+player1.getName()+"]"+hand1+" vs "+
17                                "["+player2.getName()+"]"+hand2);
18         }
19     }
20 }
```

ソースコード 1.17: JankenPlayerTypeB.java

ここでのポイントは

```
RandomJankenPlayer player2 = new JankenPlayerTypeB("Yamamoto");
```

という部分です。JankenPlayerTypeB は RandomJankenPlayer を継承したものなので、RandomJankenPlayer クラスの実体としても扱うことができます。このようにすることで、

```
hand1 = player1.showHand();
hand2 = player2.showHand();
```

と同じ showHand メソッドを呼び出していますが、player1 はランダム戦略、player2 はグーしか出さな

い固定戦略と、異なる振る舞いをさせることができます。この仕組みは**「多態性 (polymorphism)」**と呼ばれ、継承を利用したプログラミングの基本となるものです。先ほどオーバーロードによる対処では組合わせ的にメソッドの追加が必要となりましたが、継承を用いれば、さきほど作成した審判クラス Judge の内容を全く変えることなく、この新しい戦略を組み込んだプレイヤーを参加させることができます。(先ほど、Judge クラスのメソッドをさらに加えていたとしても同様に動作すると思います)

下記のコードのように JankenPlayerTypeB は RandomJankenPlayer を継承しているため、RandomJankenPlayer player2 という変数にインスタンスを保持できます。もちろんその場合使えるメソッドは RandomJankenPlayer クラスのものに限定されますがここでは対戦に使う showHand しか使わないので問題ありません。多態性により player2 が保持するオブジェクトは JankenPlayerTypeB クラスのインスタンスなので showHand メソッドはランダムじゃんけんではないことに注意しましょう。

```
public static void main(String[] args) {
    try{
        int num = Integer.parseInt(args[0]);
        RandomJankenPlayer player1 = new RandomJankenPlayer("Yamada");
        RandomJankenPlayer player2 = new JankenPlayerTypeB("Yamamoto");
        Judge judge = new Judge("Sato");
        judge.setPlayers(player1,player2);
        judge.play(num);
    }catch(Exception e){
        System.out.println("this requires an integer argument.");
    }
}
```

ソースコード 1.18: Judge クラスを用いた main メソッドの定義

オブジェクト指向を既にご存知であれば、この継承の使い方は少々気になるかもしれませんが、後で設計は直すので、ここではこの道具としての「継承」の要点に着目してください。

今回の場合、もともと継承の基底クラスとなっていた RandomJankenPlayer クラスのメソッドはあまり多くなかったですが、すでにプレイのためのいろいろな補助メソッドを実装してあり、それはプレイヤの戦略によらず必要になるものであったら、継承を利用することでそうした重複部分の実装をしなくてすみます。

子クラスでは親クラスの機能を基本的に引き継ぐため、子クラスで追加したい、あるいは、今回のように挙動を変えたい「差の部分だけ」をプログラミングすれば良く、便利です。なおかつ、今回のように審判クラスへの修正も「多態性」を用いることで一切避けることができます。

このようにある部品の振る舞いの一部だけを書き換えたいとき、「継承」はとても便利な道具を提供しています。子クラスでは変更が必要な「差分のみ」をプログラミングすれば良いだけです。このようなアプローチを**「差分プログラミング」**と呼び古くから使われています。

しかし、一方でこのような差分プログラミングでは継承をするときには親クラスの仕様を意識する必要があります。実際はこの「差分プログラミング」の側面のみで継承を使っていくと、非常に読みづらいコードになりやすく注意が必要となります。この点については次節で考えることにして、他の戦略に従うじゃんけんプレイヤーの追加を行いましょう。

6.6 継承におけるコンストラクタの注意点

Java ではコンストラクタは継承されないため、子クラスのコンストラクタの最初で自動的に呼び出しされます。RandomJankenPlayer のように継承するクラスのコンストラクタに引数がある場合 は super で明示的に呼び出していました。従って、main メソッドではなくコンストラクタで名前をセットする場合、継承する RandomJankenPlayer クラスのコンストラクタに修正が必要となります。また、name

フィールドを子クラスで再定義すると、このフィールドを直接参照していたメソッドは機能しなくなるため、フィールドへのアクセスはクラス内であってもアクセサ経由にしておくべきです。参考として、InteractiveJankenPlayer のコンストラクタは引数なしで定義しておき、コンストラクタ内で標準入力から文字列の入力を取得して、名前フィールドをセットする一つのやり方をあげておきます。

```
public class RandomJankenPlayer {
    private String name;
    public String getName(){
        return this.name;
    }
    public void setName(String name){
        this.name = setName(name);
    }
    public RandomJankenPlayer(){}
    public RandomJankenPlayer(String name){
        this.name = name;
    }
    ...
    // 他のメソッドではprivate変数にはアクセサ経由にしておく
    public void report(){
        System.out.println(getName()+" "+getNWin()+" win, "+getNLose()+" lose, "+
            getNDraw()+" draw");
    }
    ...
}

public class InteractiveJankenPlayer extends RandomJankenPlayer {
    public InteractiveJankenPlayer(){
        System.out.print("Your Name? ");
        String scanner = new Scanner(System.in);
        String name = scanner.nextLine();
        this.setName(name);
        scanner.close();
    }
    ...
}
```

ソースコード 1.19: 子クラスのコンストラクタで名前を初期化する場合

6.7 作業と課題

作業 1.10

ClassA.java および ClassB.java を Web サイトからダウンロードし実行せよ。内容を読み、「継承」と super について挙動を確認せよ。

作業 1.11

JankenPlayerTypeB.java を Web サイトからダウンロードし、実行せよ。同じクラスの異なるインスタンスでは同じメソッドを読んでも結果が異なる「多態性」の挙動を確認せよ。

課題 1.9

対戦のたびに標準入力から手を人間が指定する InteractiveJankenPlayer を作成せよ。main メソッドでプレイヤの名前も最初に入力させ、その入力文字列を用いてコンストラクタを呼び出すこと。実際にランダムプレイヤと対戦してみよ。

注意! 文字列と文字列を比較する時「==」を用いると、同じ文字列だが異なるインスタンスのとき偽を返してしまうため、equals メソッドを用いること。詳細は API リファレンスを参照。

```
Your Name? Takigawa
Player1 : Takigawa
Player2 : Suzuki
Judge   : Sato

Results: 5 games
Your Hand? (R/S/P) R
Takigawa:ROCK vs Suzuki:ROCK
Draw...
Your Hand? (R/S/P) S
Takigawa:SCISSORS vs Suzuki:PAPER
Takigawa Win!
Your Hand? (R/S/P) R
Takigawa:ROCK vs Suzuki:SCISSORS
Takigawa Win!
Your Hand? (R/S/P) P
Takigawa:PAPER vs Suzuki:ROCK
Takigawa Win!
Your Hand? (R/S/P) S
Takigawa:SCISSORS vs Suzuki:PAPER
Takigawa Win!

Takigawa 4 win, 0 lose, 1 draw
Suzuki 0 win, 4 lose, 1 draw
```

課題 1.10

前回の勝敗と出した手を記録しておくような機能を追加し、基本的にはランダムにプレイするが、前回のプレイが負けだった場合は、相手が出した手 (負けた手) をそのまま出す showHand メソッドを持つ JankenPlayerTypeA クラスを作成せよ。

Judge クラスに変更を加え、JankenPlayerTypeA、JankenPlayerTypeB、RandomJankenPlayer を戦わせ、以下のように対戦の結果のトレースを出力するようにせよ。これによって、JankenPlayerTypeA のインスタンスでは、敗れた場合、前回敗れた手を出すようにプレイしていることを確認せよ。二回連続で敗れた時、手が変わっていることも合わせて確認すること。この例では Suzuki が TypeA 戦略 (Yamada はランダム) である。

```
Player1 : Yamada
Player2 : Suzuki
Judge   : Sato

Results: 10 games
Yamada:ROCK vs Suzuki:PAPER
Suzuki Win!
Yamada:SCISSORS vs Suzuki:PAPER
Yamada Win!
Yamada:ROCK vs Suzuki:SCISSORS
Yamada Win!
Yamada:ROCK vs Suzuki:ROCK
Draw...
Yamada:PAPER vs Suzuki:ROCK
Yamada Win!
Yamada:SCISSORS vs Suzuki:PAPER
Yamada Win!
Yamada:ROCK vs Suzuki:SCISSORS
Yamada Win!
Yamada:PAPER vs Suzuki:ROCK
Yamada Win!
Yamada:ROCK vs Suzuki:PAPER
Suzuki Win!
Yamada:PAPER vs Suzuki:PAPER
Draw...

Yamada 6 win, 2 lose, 2 draw
Suzuki 2 win, 6 lose, 2 draw
```

課題 * 1.11

じゃんけんプレイヤー A や B と同じ要領で「継承」と「多態性」を用いて色々な戦略をとる他のプレイヤを加えてみよ。相手の直前 n 回の履歴を覚えておき、直前 $n-1$ 回の相手の手に対して過去の履歴からもっとも出しやすい手に勝つ手を出す履歴学習型戦略を実装して、InteractiveJankenPlayer を用いて対戦してみること。ランダムプレイヤーと比べて強く感じるか検証してみること。

- でたらめ戦略 (RandomJankenPlayer)
- ものまね戦略 (JankenPlayerTypeA)
- 一筋戦略 (JankenPlayerTypeB)
- 履歴学習型戦略

7 オブジェクト指向入門 (3): Interface と抽象クラス

これまでの作業・課題で、様々な戦略の異なる 2 人対戦じゃんけんシミュレータができあがったので、再度、5.2 節を読み、本来やりたかった内容を確認しておきましょう。

いままでは基本的に 2 人対戦のじゃんけんのみを扱ってきました。本節では、仕上げとして、複数人でのじゃんけん対戦を行ってみます。多数でじゃんけんをやると引分けの確率が上がるということは経験的に知っていますが、様々な戦略をもつプレイヤが混ざっていると理論上の分析は困難になるので、これをシミュレートしてみましょう。そのために、本節では継承に関するより詳しい概念を見ていきます。具体的には、抽象クラス (abstract class) とインタフェース (interface) という概念を説明します。このためには、実装の継承と仕様の継承、クラスの継承 (is-a 関係) と委譲 (has-a 関係)、final 修飾子による継承保護、多重継承の問題、などを考える必要があります。

本節の作業と課題はまとめて末尾の 7.8 節に記載しています。

7.1 「差分プログラミング」の乱用の弊害

前節では「差分プログラミング」として、RandomJankenPlayer を継承することで、その全機能を引き継ぎながら、新しいメソッドを加えたり、親クラスのメソッドの挙動を一部変更したりして、「差分」のみをプログラミングすることで、重複した処理のコーディングを省略できることを見てきました。

しかし、この「差分プログラミング」を主な動機として「継承」を利活用するのは本末転倒であって、使い方によっては、非常にわかりづらいプログラムとなってしまいがちです。

例えば、class A が既に定義されていて、このクラスの一部機能だけが必要となる class B を作成しなければならないとします。このとき、重複をさけるため、class B extends A とすれば、B は必要の無い class A の機能をも多数継承してしまって、とても冗長なクラスとなってしまいます。複数人で開発しているとき、このような経緯を知らずに class B を呼び出したり、継承したりすることを考えると、もはや「部品 (class B)」の使用者がどの機能が本当に class B の機能の実現に必要で、どの機能はもともと必要ではなかったが class A に実装されていたことによって class B にも引き継がれている機能なのか皆目見当がつかなくなるでしょう。オブジェクト指向で「継承」を覚えると色々と使ってみたくなる花形機能なのですが、実際は継承はかなり考えて使わないと逆に多様なトラブルの原因にもなってしまいます。ただ、差分プログラミングのために継承してしまうと、子クラスのソースコードを読んだだけでは何をしているのかさっぱりわからない解読が難解なスパゲッティコード (こんがらがってしまったコード) になってしまいます。親の仕様も知らないと使えないのでは「部品」としてまずい設計でしょう。

7.2 「実装の継承」と「仕様の継承」

ここでは、継承を用いる時、「実装」を継承するのか、「仕様」を継承するのか、という側面から、RandomJankenPlayer を継承して、課題 1.10 で作成した JankenPlayerTypeA の例を再考してみましょう。

まず、JankenPlayerTypeA を定義する際に、getName() など既に RandomJankenPlayer に実装されているメソッドを活用することを考えて行う継承を「実装」の継承と言います。実装を子クラスに継承すれば、子クラスではわざわざその実装済みのメソッドを再コーディングする必要もなく、「部品の再利用」という点では効率の高いプログラミングが可能となります。

一方、継承によって生じる「多態性」によって、JankenPlayerTypeA、JankenPlayerTypeB、InteractiveJankenPlayer など異なるじゃんけん戦略を持つプレイヤーを RandomJankenPlayer クラスのインスタンスでもあるかのように扱って、Judge のソースコードに修正を加えることなくじゃんけんの勝敗判定を行うことができました。この際にはいわば、JankenPlayerTypeA は RandomJankenPlayer クラ

スに「キャスト」されたかのように振る舞います。この場合は、「実装」の再利用のために継承していると言うよりは、showHand() 等、一連の規定されたメソッドを備えた RandomJankenPlayer クラスとして、新しいクラスも扱える「仕様」とします、という「仕様」の継承になっています。親の showHand() クラスは子クラスでオーバーライドするため、この親クラスの showHand 自体の実装は再利用されていません。が、Judge の引数に渡して、showHand() メソッドを呼び出すことによて、多態性により、様々なじゃんけん戦略を使うことができます。

以降ではこのように「仕様」の継承 (振る舞いに関する仕様の雛型) としての継承機能を活用して、さきほどのじゃんけんプレイヤーの設計を再考することにします。通常は実装を「空」にして定義することは勿論できないため、仕様の継承のための単なるダミーであったとしても親クラスのメソッドはなんらかの処理 (何もしないという処理も含めて) を実装してある必要があります。そういった仕様の継承のためのクラスを継承するときには、子クラスでは「これこれのメソッドは子クラスで実装することを念頭にカラになっているので必ずオーバーライドしてください」ということを徹底しておく必要があります。誤って、そのようなメソッドをオーバーライドしないで子クラスを定義してしまうと、重要なメソッドが空処理のままになってしまいます。

Java ではこのようなことが起こらないように、空処理になっており明示的に子クラスでかならずオーバーライドすることを念頭にしたメソッドを伴う「仕様の継承」のためのクラスを定義できます。そのための機構として、「抽象クラス (abstract class)」と「インタフェース (interface)」が用意されています。これらのクラスは通常のクラスと異なり、未実装で定義だけのメソッドを含めることができるため、逆にインスタンス化することはできません。仕様の継承はオブジェクト指向の概念ではもっとも重要な考え方の一つであり、この「抽象クラス」「インターフェース」も上手に併用して、安全なオブジェクト指向を考える必要があります。

7.3 抽象クラス

前節のやり方では、RandomJankenPlayer を継承して、負けた時の手を出す JankenPlayerTypeA や、ずっとグーを出し続ける JankenPlayerTypeB を定義しました。この際には RandomJankenPlayer のじゃんけん戦略を規定している showHand メソッドをオーバーライドすることで、ランダムにグー・チョキ・パーを出す以外の振る舞いをさせていました。また、多態性により、JankenPlayerTypeA や JankenPlayerTypeB は、同じ名前のメソッド showHand を呼び出すだけで各々の固有の戦略行動を取れます。これらのクラスは親クラス RandomJankenPlayer のインスタンスとしても扱うことができるため、Judge クラスを変更せずに勝敗判定をそのまま以前のプログラムで行うことができました。

```
RandomJankenPlayer player1 = new RandomJankenPlayer("Yamada");
RandomJankenPlayer player2 = new JankenPlayerTypeA("Suzuki");
Judge judge = new Judge("Sato");
judge.setPlayers(player1,player2);
```

のような例です。しかし、よく考えると、

```
RandomJankenPlayer player2 = new JankenPlayerTypeA("Suzuki");
```

で定義される変数 player2 は RandomJankenPlayer なので、親クラスの実装の詳細をあまり把握していない末端利用者からは、ランダムにじゃんけんをするプレイヤとして読めてしまうでしょう。これではあまり良い設計とは言えなそうです。このような点は単に「差分プログラミング」を目的として継承を行ってしまった弊害でもあります。

そこで、ちゃんと Java の「抽象クラス」という機能を用いて、「仕様の継承」のためだけの空処理の showHand メソッドを持つ基底クラス JankenPlayer を定義して、RandomJankenPlayer、JankenPlayerTypeA、JankenPlayerTypeB、InteractiveJankenPlayer 等は、そのクラスを継承して showHand を実装することで実現するように設計方針を変えましょう。

抽象クラスでは `abstract` 修飾子によって未実装な部分を含むことを明示します。例えば、showHand() を未実装にする場合は、class と showHand に abstract を付与して次のように定義すれば良いでしょう。それ以外のメソッドについては RandomJankenPlayer を継承したときと同様、実装を入れておけばそのままの機能が継承されるので、この子クラスであれば必要となる機能で子クラスでは変更が生じなさそうなメソッドを実装しておくことができます。

```
1 public abstract class JankenPlayer {
2     ... // 省略
3     public abstract Hand showHand();
4     ... // 省略
5 }
```

ソースコード 1.20: 抽象クラス JankenPlayer

このように抽象クラスとして定義しておけば、呼び出し側でも

```
JankenPlayer player1 = new RandomJankenPlayer("Yamada");
JankenPlayer player2 = new JankenPlayerTypeA("Suzuki");
Judge judge = new Judge("Sato");
judge.setPlayers(player1,player2);
```

となり、player2 は単なる JankenPlayer の一種ではある点が明示でき、Random な戦略を取るなどという誤解も生じず、より自然で誤解のないコードとなります。

この抽象クラスを継承することで各々の固有の戦略をもったじゃんけんプレイヤーを定義すれば良いでしょう。例えば、JankenPlayerTypeB は以下のように簡潔に定義することができます。

```
1 public class JankenPlayerTypeB extends JankenPlayer {
2     public JankenPlayerTypeB(String name){
3         super(name);
4     }
5     public Hand showHand(){
6         return Hand.ROCK;
7     }
8 }
```

ソースコード 1.21: 抽象クラス JankenPlayer を継承して定義された JankenPlayerTypeB クラス

7.4 インタフェースと多重継承の問題

Java では基本的に一つの親クラスからしか継承できません。例えば、`SongWriter` クラスで定義されている作曲機能と、`Singer` クラスで定義されている歌手機能と、両方を備えた `SingerSongWriter` クラスを定義したい場合を考えてみましょう。単純に考えると、これら二つを親に持つクラスを定義すれば良さそうです。

```
class SingerSongWriter extends Singer, SongWriter {
   ...
}
```

しかし、このような「多重継承」と呼ばれる二つ以上の親クラスからの継承は Java では禁止されています。なぜ、禁止する必要があるかと言えば、もし Singer クラスと SongWriter が名前が同じだが実装のことなるメソッドを持つ際に、実装が衝突しうるため、子クラスでどちらの実装を採用するかが不定になるからです。(C++など、そのような場合はコンパイルエラーとして多重継承を許容できるプログラミング言語も存在します)

しかし、実装が衝突しない場合に限れば、このような多重継承を許容できると便利です。Java では、いかなる実装をも持たない抽象クラスとして「インターフェース (interface)」という特殊クラスが用意されています。これはあくまで「仕様の継承」のためであって、すべてのメソッドは子クラスで「実装されなければならない」物理的制約として機能するものです。インターフェースで規定されたメソッドが子クラスで実装されていないとコンパイルエラーになります。

抽象クラスとインターフェースは混同しやすい概念ですが、両者の違いは、抽象クラスはメソッドの実装を持てるのに対してインターフェースは持てないことであって、前者は実装も継承できるのに対して、後者は「仕様の継承」のための「型定義」にのみ特化しているものであるということです。

インターフェースは実装を持たないため、多重継承しても衝突が起こりえません。従って、Java でもこの interface の場合は多重継承が可能です。すなわち、Java では、二種類の継承を分けて、仕様の継承のみ多重にできるようにしています。このようにして、データ構造の衝突やクラス階層の複雑化などを回避しているのです。

インターフェースはあくまで「型定義」のように「仕様」を決めて、子クラスで共通化させる用途に使います。例えば、JankenPlayer も Judge も getName で名前を問い合わせることができました。画面つきのゲームアプリケーションとして開発するのならば、各々キャラクタ用の顔画像ファイルが紐付いているかもしれませんし、それを画面に描画するメソッドを持っているかもしれません。

「名前を問い合わせることができる」クラスを「NameAvailable」というインターフェースとして定義するには以下のように記述します。

```
1 public interface NameAvailable {
2     public String getName();
3 }
```

ソースコード 1.22: NameAvailable.java

このインターフェースを継承するには extends ではなく implements を用います。

```
1  public class TestInf implements NameAvailable {
2      private String name;
3      public String getName(){
4          return this.name;
5      }
6      public TestInf(String name){
7          this.name = name;
8      }
9      public static void main(String[] args) {
10         TestInf t = new TestInf("hoge");
11         System.out.println(t.getName());
12     }
13 }
```

ソースコード 1.23: TestInf.java

インターフェースは実装を持たないので継承/拡張する (extend する) というより、インターフェースで規定された仕様を実装する (implement する) というニュアンスでしょうか。なお、通常は省略しますが、これらは未実装な部分であるので、正確には abstract 修飾子をつけたものと等価です。

```
1 public abstract interface NameAvailable {
2     public abstract String getName();
3 }
```

ソースコード 1.24: インターフェース NameAvailable を実装したクラス定義の例 (abstract 付与版)

Java ではインターフェースに実装を含むことは許されていないため、それ単体で提供しても実装者の負担になることが多いです。そこで、インターフェースと一緒に骨格実装クラスを提供することがよくあります。骨格実装クラスは AbstractInterface と呼ばれ、Java の主要な Collection Framework ではそれが提供されています。例えば、 java.util.List インターフェースに対しては java.util.AbstractList が提供されています。API リファレンスを確認してみてください。骨格実装が用意されているようなほとんどの場合は、骨格実装を継承してインターフェースを実装することが推奨されています。また、自ら骨格実装を作る場合、それが継承して使われることを想定して実装する必要があります。

7.5 クラスの継承 (is-a 関係)、委譲 (has-a 関係)、インタフェース (can-do 関係)

オブジェクト指向の「継承」とそれに付随する「多態性」は花形機能であり、非常に強力なプログラミング機能を提供しています。しかしながら、無計画にただ「差分プログラミング」のみを意識して継承を利用すると、すぐスパゲッティコード化する要因にもなりがちです。

一般に、クラスの継承は、そのクラスが表現している対象がサブクラスになるに従って「特化」していくようにし、スーパークラスになるに従って「汎化」していくように設計するのが望ましいとされています。RandomJankenPlayer や InteractiveJankenPlayer はそれらを含む概念である JankenPlayer の一種になっています。一方、JankenPlayerTypeB は RandomJankenPlayer の一種にはなっていなかったので、最初の継承関係はあまり望ましいものではなかったということです。これらの関係はしばしば「is-a 関係」と呼ばれます。

- RandomJankenPlayer is a JankenPlayer
- InteractiveJankenPlayer is a JankenPlayer

となっていますが、「JankenPlayerTypeB is a RandomJankenPlayer」ではないため、前者は良く、後者は悪いということになります。この「is-a 関係」になっているかはクラス継承を用いるかどうかの一つの判定指針になります。

一方、単にあるクラス A の機能をクラス B で使いたいだけであれば、何も継承する必要はなく、クラス A のインスタンスを一つ生成して、クラス B でクラス A 型のフィールドに保持しておくという形もあり得ると思います。これをオブジェクト指向では**「委譲 (delegation)」**と呼びます。継承と並んで、良く使われるパターンになっています。委譲が適している場合は、「is-a 関係」ならぬ「has-a 関係」になっていると言われます。例えば、ArrayList に String name フィールドを付与したいだけであれば ArrayList を継承して子クラスを作成し、そこに name フィールドを持たせるのではなく、ArrayList のインスタンスと String name をフィールドに持つクラスを設計すればよいということです。この場合、「the class has a ArrayList」というわけです (英語的には an ですが...)。

一方、継承関係は「is-a 関係」、委譲関係は「has-a 関係」という意味でなぞらえれば、インタフェースは「can-do 関係」と言われます。インタフェースでは、単に実装しなければいけないメソッドの仕様を規定しているだけで、これらのメソッドが持つ機能を “can do” するととらえることができます。

7.6 参考：クラス継承となりすまし、final 修飾子による継承の保護

RandomJankenPlayer のところで見たように、RandomJankenPlayer を継承した子クラスのインスタンスは RandomJankenPlayer のインスタンスとして振舞うことができます。オブジェクト指向の 3 本柱の一つである多態性もこうして実現されていました。しかし、これは場合によってはこの仕組みによって意図しないサブクラスが作成されて不正な使われ方を許容してしまう恐れがあります。

Java にはこの意図しないサブクラスが作成されないようにする指示が用意されています。もし、RandomJankenPlayer として振舞うことが可能な子クラスの生成を禁止する場合には、final 修飾子をクラス定義に付与することでこのクラスの継承を禁止することができます。

```
public final class RandomJankenPlayer extends JankenPlayer {
   ...
}
```

クラス継承に基づく Java の多態性の機能は強力なプログラミング手段ですが、継承を無制限に許したままのクラスを放置しておくと意図しないサブクラスが作成されて不正に利用されるリスクがあるということを覚えておきましょう。実際にクラスの継承を許すか否かはケースバイケースであり、それぞれのクラスの用途、重要度などを考慮して慎重に決める必要があります。この演習では継承を禁止しなければならない例を扱いませんが、以下のウェブページを参照してこの点を把握しておいてください。

クラス継承となりすまし
`https://www.ipa.go.jp/security/awareness/vendor/programmingv1/a03_04.html`

7.7 参考: javadoc によるドキュメンテーション

自分で作成したクラス群も再利用や後からの見返しのために API リファレンスを用意しておく方が安全です。設計時にはわかっていたこともしばらくすると忘れてしまいますし、自分以外の第三者が再利用する機会があるなら API リファレンスが非常に重要であることは、標準ライブラリ・非標準ライブラリのクラスを使ってみる実習で感じたことと思います。

Java では決まった形でコメントを残しておくことによって、自動的にあの形式の API リファレンスを生成できる javadoc という機能が用意されています。そのフォーマットについて逐一ここで記載することはしませんが、例えば以下の Web サイトや書籍などを参考に自分で作った課題プログラムの API リファレンスを javadoc で生成してみても良いでしょう。

- Javadoc 入門
  `https://ssl.aiosl-tec.co.jp/java-start/chap14.html`
- いまさら聞けない「Javadoc」と「アノテーション」入門
  `http://www.atmarkit.co.jp/ait/articles/1105/19/news127.html`

7.8 作業と課題

本節の演習は、いままでのじゃんけんプログラムの仕上げになります。次の課題でまずは継承に関する設計の見直しを行いプログラムを改善しておきましょう。

課題 1.12

抽象クラス JankenPlayer を作成し、それを継承することで、RandomJankenPlayer, JankenPlayerTypeA, JankenPlayerTypeB, InteractiveJankenPlayer を再定義せよ。前節で自作のじゃんけんプレイヤーを作成した場合は同様に再定義せよ。また、それぞれのクラスおよび Judge クラスには getName メソッドを受け付ける NameAvailable インタフェースを実装して、ゲームの開始時に「Member List? (Y/N))」と聞き「Y」を応答した場合、審判も含めて、その場のメンバーの一覧をアルファベット順で表示するようなメソッドを作成せよ。

課題 1.13

本節の目標である複数人対戦を実現するサンプルとして Results.java および Judge2.java を Web サイトからダウンロードし、内容を解析せよ。どのように複数人対戦を実現しているか考察し簡潔に概要をまとめよ。

また、この例を元にして Java プログラムを作成し、以下の点を検証せよ。ただし、プレイヤ名は Player1, Player2, ... などとし、表示させる必要はない。

- n 人でじゃんけんを行ったとき「引き分け」が起こる率はどれくらい上がっていくのだろうか? 3 人、5 人、7 人、9 人、で 100 回じゃんけんを行ってみてこの値をシミュレートしたい。

課題 * 1.14

本演習および Java プログラミング・オブジェクト指向プログラミングについて感想を自由に述べよ。(難しかった点の情報は今後の授業の改善に役立てるため歓迎)

課題 * 1.15

Judge2.java を解析し参考にすることで、さらにじゃんけんの勝敗について自由に Java プログラムを作成し解析せよ。例えば以下のような点を検証するような Java プログラムを作成せよ。

- n 人でじゃんけんをして、勝者 1 人を決めたいとすると勝敗が決まるまでに何回くらいじゃんけんが必要なのだろうか? その値は n に応じてどのように増加するだろうか?
- 人数が多くなると引き分けが増えるため、「もし場の手が 3 種類のときは手が多いグループの人たちが勝ちとする」という引き分け解消ルールを導入するとこの回数はどのようになるだろうか?
- n 人のじゃんけんで、非ランダムな戦略のプレイヤを導入するとじゃんけんの勝敗の傾向はどのようになるだろうか?

8 まとめとレポート作成について

お疲れ様でした。以上で終わりです。最初のページにある通り、いままでの課題についてレポートにまとめて提出してください。**カプセル化、継承、多態性**は理解できましたか？

Java は世の中で最もよく使われているプログラミング言語の一つです。また必要になったとき、この演習を思い出して復習してください。さらに理解を深めるにはまず以降の付録を参考にしてください。最後までつきあってくれてありがとう。また、どこかで。

付録１：Java とは

プログラミング言語 Java(以下、Java 言語) は、現在最も広く利用されているオブジェクト指向プログラミング言語の一つです。本節では、Java プログラムの基本的な構造を理解すること、また、Eclipse の基本的な使用方法を習得し、C 言語の知識を元にして簡単なプログラムを作成できることを目標とします。この演習書のみで Java 言語を学習することは難しいので、前節で述べたようにここでは必要最小限の解説にとどめています。適宜、前節の情報、Web サイト、参考書籍などを参考にしてください。

連載「Eclipse ではじめるプログラミング」(改訂版)
`http://www.atmarkit.co.jp/fjava/index/index_java5eclipse.html`

参考となる本やプログラム等

Java 言語の基本部分は C 言語と共通しているため、この指導書では例を通したごく簡単な説明にとどめています。Java の全体像を理解するためには、市販の入門書を入手 (購入あるいは図書館で借用) して、必要に応じて自学することを推奨します。Java の入門書は数多く出ているので実際に手にとって自分の好みに合うものを探して下さい。例えば、以下の本があります。

- 中山清喬・国本大悟 (著), スッキリわかる Java 入門 第 2 版 (スッキリシリーズ). インプレス, 2014. (ISBN-10: 484433638X)
- 中山清喬 (著), スッキリわかる Java 入門 実践編 第 2 版 (スッキリシリーズ). インプレス, 2014. (ISBN-10: 4844336770)
- 小森裕介 (著), なぜ、あなたは Java でオブジェクト指向開発ができないのか—Java の壁を克服する実践トレーニング. 技術評論社, 2004. (ISBN-10: 477412222X)

公式な Java のチュートリアルや詳しい説明は以下のサイトで利用できます。最新情報は英語ですが少し前のものは日本語版があるものもあります。

Java 言語自体についてわからないことがあればこの公式ドキュメントにあたるのがもっとも確実です。(技術英語の勉強をかねて?) 英語の公式チュートリアルを見てみるととても理解が深まると思います。Java のバージョンが変わったときなど、書籍やインターネットで検索できる情報は古くなっていたり、現在はもっと良い書き方があったりしますので、今後も Java とつきあっていくにはぜひ覚えておいて欲しいリソースです。

- Java 公式ドキュメント
  `https://docs.oracle.com/javase/tutorial/`
- 日本語の Java 公式ドキュメント
  `http://www.oracle.com/technetwork/jp/java/index.html`
- 連載「Eclipse ではじめるプログラミング」(改訂版)
  `http://www.atmarkit.co.jp/fjava/index/index_java5eclipse.html`

Java の名前とバージョンについて

Java は Java 言語自体と Java SE、Java EE、Java ME 等の分野に応じた標準ライブラリから成り立っています。また必要に応じてオープンソースや有償のライブラリを使って開発が行われることも多いです。

Java の開発・実行環境は 1996 年のリリース当初、JDK と呼ばれ、1.0、1.1 とマイナーアップデートが進みました。1.2 からは Java 2 と呼ばれ略称として J2SE 1.2 と表記されていました。1.2、1.3、1.4 とアップデートがあり、J2SE 5.0 では、バージョン番号の最初の 1 が外されました。6.0 からは Java 2 ではなくなり、Java SE 6 と表記され、マイナーアップデートの表記も外されました。マイナーバージョンは update で表され、現在では、Java SE 8 update 25 などと表記されています。

J2SE 1.2 から、Java には SE(Standard Edition)、EE(Enterprise Edition)、ME(Micro Edition) という 3 つのエディションができました。このうち、Java SE が基本となるエディションでデスクトップなども対象にしています。Java EE は、Web や業務アプリケーション用のライブラリ、サーバの仕様をまとめたエディションで、Java 環境としては Java SE を使います。Java ME は組込み用のエディションで、メモリなどのリソースが少ない環境向けに API が簡素化されています。

JDK(Java Development Kit) は Java の開発環境であり、JRE(Java Runtime Environment) は Java の実行環境です。Java SE とは、Java の言語やライブラリ、実行環境がどのようなものであるかを規定した仕様です。JDK、JRE は Oracle 社によって開発されたこのような Java SE の実装の一つです。JRE には、Java SE の仕様に従ったライブラリと実行環境、そしてブラウザ内で Java アプレットを動かすためのプラグイン、加えて GUI ツールキットである JavaFX が含まれます。ここで JavaFX は JRE や JDK に含まれるものの、Java SE には含まれないことには注意が必要です (JavaFX は Java SE 9 から正式に標準として含まれる予定です)。JDK には JRE に加えて、javac コンパイラなどの開発ツールを含みます。

Java SE の仕様は前のバージョンとの差分として記述され、Java SE が全体としてどうなっているかを記した仕様はありません。JDK 1.1 までは仕様と実装が別れていなかったため、厳密にどこまでが Java SE の仕様で、どこからが JDK 独自の実装であるかはあいまいです。

付録 2: Eclipse で作成される class ファイルについて

Eclipse で作成された実行ファイル (.class ファイル) を以下の手順で直接実行して、Eclipse の画面で編集しているソースファイルと生成される実行ファイルの実体との関係は以下のようになっています。初期設定でワークスペースのディレクトリを変更した場合は適宜「`~/workspace`」をそのディレクトリに置き換えること。

```
$ cd ~/workspace/Kadai1/
$ ls
bin   src
$ cd bin
$ java HelloWorld
Hello, World
```

付録 3: 型, ラッパークラス, オートボクシング

ArrayList を定義するときに、どのようなクラスを含むのか<>で囲んで指定しています。これをジェネリクスと言います。どのような「クラス」も指定することができますが、int とか double などの**「型」を指定することはできない**ので注意が必要です。代わりに Integer や Double などの各々の型に相当するクラスを指定します。

Java のややこしい点の一つはこのように int 型、double 型などと別に Integer クラス、Double クラスというクラスがあることです。Java プログラムの基本構成単位はクラスですが、型は採用されています。例えば、byte(8bit 整数), short(16bit 整数), int(32bit 整数), long(64bit 整数), float(32bit 単精度浮

動小数点数), double(64bit 単精度浮動小数点数), char(16bit の Unicode 文字) の型は C 言語とほぼ同じです。「true(真)」と「false(偽)」の 2 値をとる boolean 型もあります[9]。

Java では基本構成単位がクラスなので、型のままだと扱えず、どうしてもクラスに変換する必要がある場合があるため、各々の型に対応するクラスが存在します。各々、java.lang.Byte, java.lang.Short, java.lang.Integer, java.lang.Long, java.lang.Float, java.lang.Double, java.lang.Character, java.lang.Boolean です。

ただし、面倒だということは Java 設計者も気づいており、オートボクシングという各々の型とクラスは自動変換される機能がついています。

```
1  public class TestTypes {
2      public static void main(String[] args) {
3      // 明示的な型-クラス変換
4      int int1 = 15;
5      Integer int2 = Integer.valueOf(int1);
6      int int3 = int2.intValue();
7      // オートボクシング
8      int int4 = 16;
9      Integer int5 = int4;
10     int int6 = int5;
11     }
12 }
```

ソースコード 25: TestTypes.java

付録 4: ファイル入出力と try-catch 構文による例外処理

次の例はファイルの入出力を扱うクラスです。file.txt というテキストファイルを 1 行づつ読み込んで表示する処理を二通りで書いたものです。try-catch 構文のところをのぞき、やはり標準クラスを使った処理なので、もう API リファレンスを見ながら動作を解析することができますね。java.util パッケージの Scanner クラスの説明と java.io の BufferedReader クラスの説明を読み比べてみてください。

```
1  import java.io.*;
2  import java.util.Scanner;
3
4  public class TestFileIO1 {
5      public static void main(String[] args) {
6          try{
7              Scanner scanner = new Scanner(new File("file.txt"));
8              while (scanner.hasNextLine()) {
9                  String line = scanner.nextLine();
10                 System.out.println("READ: "+line);
11             }
12             scanner.close();
13         }catch(FileNotFoundException e){
14             System.err.println("ERROR");
15         }
16     }
17 }
```

ソースコード 26: TestFileIO1.java

```
1  import java.io.*;
2  import java.nio.file.*;
3
4  public class TestFileIO2 {
```

[9] Java では boolean なので気をつけてください。C++には bool 型がありますし、C 言語でも C99 では stdbool.h を include することで bool 型が使えるようになりました。

```
5     public static void main(String[] args) {
6         try{
7             Path src = Paths.get("file.txt");
8             BufferedReader br = Files.newBufferedReader(src);
9             String line;
10            while((line = br.readLine()) != null){
11                System.out.println("READ: "+line);
12            }
13        }catch(IOException e){
14            System.out.println("ERROR");
15        }
16    }
17 }
```

ソースコード 27: TestFileIO2.java

try-catch 構文は C 言語では出会わない構文で、「実行時エラー/例外」の処理を行っています。C 言語でファイルを開く際、もしファイルが存在しなかったらエラーを表示する、などのエラー処理が必要であったと思います。また、ファイルを書き込む場合に書き込み権限がない、メモリを確保しにいったが空きメモリがない、通信を試みたがネット接続されていない、など、実行してみて初めてエラーになる場合、そのような状況をあらかじめ想定して、逐一エラー処理を記述する必要がありました。しかし、そのような従来型の例外処理の場合、

- 実行時エラーを起こす可能性があるか逐一考えてエラーチェックしなければならず、そもそもエラー処理を万全にするのが非常に大変な作業である。
- 「エラー処理」の部分の処理が「本来の処理」の流れをわかりづらくしてしまい、「本来の処理」がどの行なのかわかりづらくなる。
- あるいは、大変面倒なので、エラー処理の記述をサボって省略する恐れがある。

等の問題があります。趣味で作ったプログラムならばいざ知らず、現在では様々なソフトウェアが社会の至る所で動作しており、エラーの影響が大きく、エラーの対処や同定は非常に大事な問題です。

Java をはじめとする新しいプログラミング言語では try-catch 構文のような例外処理を記述する仕組みがそなわっています。通常実行時には try でかこまれたブロックのみが実行され、catch のブロックは実行されません。しかし、try ブロックの処理を実行中に例外的状況が生じたことがわかると、処理は直ちに catch ブロックに移行します。従って、catch ブロックには例外的な状況が起きたときの対処を記述しておきます。try-catch 構文を用いることで、本来の処理 (try ブロック) と例外処理 (catch ブロック) は明確に分離することができますし、例外的な状況が発生しているか?という面倒で煩雑なチェックをプログラマ側で行う必要がありません。

あるメソッドを用いたとき、どのような例外が発生しうるかは API リファレンスにきちんと記載されています。たとえば、例で用いている java.util の Scanner の場合、

```
public Scanner(File source)
        throws FileNotFoundException
```

とあることがわかります。この throws 以下の FileNotFoundException クラスが実際に catch すべき例外を表すクラスになります。また、API でこの FileNotFoundException をクリックしてみると

```
java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.io.IOException
        java.io.FileNotFoundException
```

と階層的になっていることが分かります。java.io.IOException や java.lang.Exception など上位のクラスを catch しておけば、下位クラスである java.io.FileNotFoundException も catch できることを示しています。この仕組みはオブジェクト指向の花形機能である「継承」という仕組みを用いています。try ブロックで様々な種類の例外が生じるような一般の場合、このような階層構造によって catch 処理を分けたりまとめたりできるのは「継承」の便利な点ですよね。ただし、上位クラスで catch すると java.io.FileNotFoundException 以外の例外処理も含むので「ファイルが見つからない時」だけの処理を記述したければそれに応じた下位クラスを catch するブロックを用意する必要があります。どのレベルで例外を catch するかは必要な処理にも依存します。

try-catch 文では、catch するエラーごとに catch ブロックを記述することができ、例外がおきてもおきなくても実行する finally ブロックをつけることもできます。この演習では上記以上の例外処理は必要ありませんが、実際に Java プログラム開発をする際にはぜひ気をつけて習得してください。

付録 5: 非標準ライブラリの利用

ライブラリのダウンロードと jar ファイル

まず、本文中の Guava の Web サイトから、トップページに見えている文書 (README.md) の「Latest release」の最新版のダウンロードリンク「Guava 19.0」をクリックしてみましょう。その先に最新版の情報があるので「guava-19.0.jar」をダウンロードして保存しましょう。

Commons Lang については、Web サイト左メニューの「Download」をクリックして、Binaries から commons-lang3-3.4-bin.tar.gz をダウンロードして保存しましょう。

charts4j については、Web サイトの「DOWNLOAD LATEST STABLE VERSION」か左メニューの「Downloads」をクリックし、最新版の「charts4j-1.3.zip」をダウンロードして保存しましょう。

「デスクトップ」等、自分がわかりやすい場所に保存すれば良いですが、日本語のディレクトリ名が入った場所に保存すると以下の端末操作の際、日本語入力が必要となり多少面倒なので、「`cd ~; mkdir download_files`」などとホームディレクトリ直下に英数字のファイル名の専用ディレクトリを作成し、そこへ保存すると便利が良いでしょう。何も指定しなければホームディレクトリにある「ダウンロード」という名前のディレクトリに保存されます。

これらのファイルを置いたディレクトリで、以下のコマンドを用いてこれらのファイルを調べてみます。charts4j については後で課題で扱うので、ここでは前者二つについて調べています。以下では tar.gz 形式を解凍し、できたフォルダからライブラリ本体をカレントディレクトリにコピーしています。

```
$ ls
commons-lang3-3.4-bin.tar.gz guava-19.0.jar
$ tar xzf commons-lang3-3.4-bin.tar.gz
$ ls
commons-lang3-3.4/ guava-19.0.jar
commons-lang3-3.4-bin.tar.gz
$ cp commons-lang3-3.4/commons-lang3-3.4.jar .
$ ls *.jar
commons-lang3-3.4.jar guava-19.0.jar
```

これらの JAR(Java Archive) ファイルにクラスライブラリの実体が収められています。また、中に収められているクラスとそのメソッドなどの API リファレンスはひとまず Web で見ることにしましょう。(UNIX コマンド tar については「man tar」などで各自調べること)

Guava の場合、上記リンクのトップページの文書 (README.md[10]) の「Snapshots」のセクションの「Snapshot API Docs:」の横の「guava」のリンクから標準 API のときに見た画面が見られます[11]。Commons Lang の場合も、トップページの「Documentation」のセクションにある「The current stable release 3.4 」のリンクで見られます。

これらの API はライブラリを活用する上で基本となる説明書なので、非標準ライブラリを使うときはまず API リファレンスを参照してください。オブジェクト指向は「部品化」であり、「部品」は中身の原理 (ソースコードの中身) を知らなくても「使い方」さえ分かれば便利であることを見てきました。そのためにはこれらの API リファレンスは非常に大切です。Java プログラミングでは javadoc というプログラムでソースコード中のコメントから自動生成する仕組みが用意されています。自分以外の人が利活用する「部品」を作る際には API リファレンスも用意することが必要なので覚えておきましょう。

さて、JAR(Java Archive) 形式は Java の複数のファイル (クラスファイル) を一つのファイルにまとめるアーカイブ形式の一つです。UNIX で用いられる tar と似ており、作成や解凍も類似のオプションで可能です。ここでは以下のように、各々の jar ファイルの中身を less コマンドで覗いてみましょう。less コマンドではスペースキーで次のページ、q で抜けてもとのプロンプトに戻ります (「man less」でチェック)。縦棒「—」は左のコマンドの出力を右のコマンドの入力に渡す「パイプ」というもので、UNIX 操作を忘れてしまった人はシェルの操作を復習しておきましょう。

```
$ jar tvf commons-lang3-3.4.jar  | less
  <Space> or 'q'
$ jar tvf guava-19.0.jar | less
  <Space> or 'q'
```

[10] これは Markdown という軽量マークアップ言語で書かれたただのテキストファイルですが HTML などと同じくビューアを通して見栄えがフォーマットされます。Markdown はカンタンなので興味があれば、ページ上のファイルリスト画面から「README.md」をクリックし、上部の「RAW」を押し見てみてください。

[11] Guava の 19.0 のリリース日が December 9, 2015 となっていますが、トップページが普通のページだったのにいきなり github にうつっていて、作っておいた演習資料を直前で直すことになりびびった...。github は現在最も人気の高いバージョン管理システム git のホスティングサービスの一つですがここではスペースがないし多くは語るまい。最近はオープンソース・ソフトウェアは github に置かれることも多くなりました。

パッケージと package 文

jar ファイルの中身を見ると、それぞれ関連する機能単位ごとに階層的なディレクトリにまとめられており、それぞれのフォルダに様々な「.class」ファイルが収められていることが分かると思います。またこの階層が API リファレンスの各々の「パッケージ」に対応することも用意に推察できると思います。実際に、Java の「パッケージ」とは機能ごとにこのように階層に分けておいて、他の人が作る「部品」の名前と衝突しないようにするための整理機構です。例えば、ある開発者が配列を拡張した「ArrayList」を定義してしまうと、名前が重複してしまうため、他の開発者はもうこの名前のクラスを作ることができません。しかし、いま存在するクラス名をいちいち全てチェックして、自分の設計するクラスの名前を考えるのはあまりに非効率なので、関連する機能ごとに「パッケージ」にまとめているわけです。ArrayList は java.util.ArrayList には定義されているため、java.util パッケージではこれ以上この名前が使えませんが、「package org.mypackage;」などと package 文で異なるパッケージ名を宣言してしまえば ArrayList という名前も使えます。

これを試すために次のコードを試してみましょう。なお、パッケージ名はなんでも良いのですが、他の開発者のパッケージ名との衝突を確実にさけるため、自分の所属組織の URL を逆にしたものを使う慣例があります (たとえば hoge.ist.hokudai.ac.jp なら jp.ac.hokudai.ist.hoge)。

```
1  package org.mypackage.test;
2  public class TestPackage {
3      public static void main(String[] args) {
4          String str = "AA,BBB,C,DD,EEE,F,GGGGG";
5          String[] strArray = str.split(",");
6          for(String s : strArray){
7              System.out.println(s);
8          }
9      }
10 }
```

ソースコード 28: TestPackage.java

ところが、これをそのままコンパイルして実行しようとしても、TestPackage.class は生成されているようなのにうまくいきません。

```
$ javac TestPackage.java
$ ls
TestPackage.class TestPackage.java
4 java TestPackage
Error: Could not find or load main class TestPackage
```

この TestPackage クラスは package 文「package org.mypackage.test;」で org.mypackage.test というパッケージに属することになっているので、物理的にもこのような階層のディレクトリに納めておく必要があります。Java ではクラス名がそのままファイル名になるので、パッケージを用いて名前の衝突がおこらないようにするために、物理的なディレクトリを対応づけているのです。

```
$ mkdir -p org/mypackage/test
$ mv TestPackage.class org/mypackage/test
$ ls
TestPackage.java org/
$ java TestPackage
Error: Could not find or load main class TestPackage
$ java org.mypackage.test.TestPackage
```

このようにパッケージ名を頭につけたものが正式なクラス名となります。これはこのクラスを他のプログラムで用いる場合でも同じです。ある package に属するクラスを他のクラスで使うには完全名で呼び出すか import 文が必要であったことをここで再度思い出しておくと良いでしょう。

さて、ここで、一番初めに HelloWorld を実行したときのことを思い出しましょう。

```
$ java -verbose:class HelloWorld
[Opened /<省略>/rt.jar]
:
```

ここで呼ばれている rt.jar の中身を見てみることにしましょう。「省略」という部分はシステムにより異なるため、実際にターミナルで表示されるものをそのままコピーしてください。

```
$ jar tvf /<省略>/rt.jar | less
:
```

less の機能で「/java/lang/String.class」と入力し、エンターキーを押すことにより、「java/lang/String.class」を検索してみましょう。ちゃんと rt.jar の中に存在していますね。つまり、Java で HelloWorld が実行できた背景にはこのようにすでに誰かが作った「String.class」が存在し、かつ、適切に呼び出されているということが必要です。従って、Java をインストールする際にはコンパイラだけではなく、このように標準クラスライブラリのインストールが必要です[12]。

従って、非標準ライブラリを呼ぶ際にもっとも大切なこと、および、もっともつまずきやすいこと、はこのように対応するクラスファイルが見つかるような設定ができているか、ということになります。

クラスパスの設定

非標準ライブラリのクラスを自分の Java プログラムで活用するためには「java」コマンドの実行時に必要なクラスファイルがどこにあるのか「java」コマンドに通知する必要があります。java が実行時にクラスファイルをハードディスクから効率よく検索するために使う場所の情報を「クラスパス」と呼びます。java の実行時にはこのクラスパスの範囲で必要なクラスがあるかが調べられます。見つからない場合は ClassNotFound エラーになります。

クラスパスを指定する方法は主に 3 つあります。

- 環境変数 CLASSPATH に宣言する。
- javac や java コマンド実行時に-cp オプションで指定する。複数のディレクトリや jar ファイルを指定する場合は「:(コロン)」で繋げる。

[12]class ファイルがあれば実行ができますが、開発用には各々の java ファイルもあるほうが良いかもわかりません。

- 何も指定しない。現在のディレクトリ (.) を指定したのと同じなので、必要なクラスファイルを一つのディレクトリに置いてそこで実行。(今までのプログラムが動いてきた理由)

クラスパスには class ファイルが置いてあるディレクトリを指定することができますが、そのディレクトリが 1 つの JAR ファイルになっている場合は、その JAR ファイル名を指定します。JAR ファイルが置いてあるディレクトリを指定するのではない点に注意してください。例えば次節 (p.**??**) で用いる例で調べてみると (Controller クラスの実行には YesMan クラスと NoMan クラスが必要な状況)、以下のようになります。注意してください。

```
$ ls
Controller.class  NoMan.class  YesMan.class
$ java Controller
..うまくいく..
$ mkdir tmp
$ mv NoMan.class YesMan.class tmp
$ java Controller
..ClassNotFound エラー..
$ java -cp .:./tmp Controller
..うまくいく..
$ jar cvf test.jar tmp
$ java Controller
..ClassNotFound エラー..
$ cd tmp
$ ls
NoMan.class  YesMan.class
$ jar cvf test.jar *.class
$ cd ..
$ java -cp .:./tmp/test.jar Controller
..うまくいく..
```

Eclipse でのクラスパスの設定

Eclipse では、非標準の外部ライブラリの指定はより直感的になっています。まず、外部ライブラリは上記のように test.jar という JAR ファイルにアーカイブされているとします。まず、画面左のパッケージ・エクスプローラの開発中の自分のプロジェクト名を右クリックして出てくるメニューから、ビルド・パス → 外部アーカイブの追加を選びます。すると、「JAR の選択画面」というファイルを選ぶ画面が出るので、追加したい JAR ファイル (例えば test.jar) を選択する、という手順になります。

なお、外部ライブラリの JAR ファイルを検索できるように場所情報を入力しただけなので、標準ライブラリのクラスと異なり、外部ライブラリのクラスに関する情報はマウスオーバーでは出ません。これを指定するには再び、プロジェクト名の右クリックメニューから、ビルド・パス → ビルド・パスの構成を開き、ライブラリ名の ▽ を開くと「ソース添付」や「Javadoc ロケーション」で指定することができます。先ほどの外部 JAR ファイルもこの場面から指定できます。この実習では必要ありませんが実際の開発ではソースや API リファレンスを関連付けておくと便利です。