情報工学実験 I
「オブジェクト指向プログラミング I」

担当：表現系工学研究室

実験の概要

この実験では、オブジェクト指向プログラミングの基本概念を理解し、Java 言語による基本的なプログラミングができるようになることを目標とします。目標の詳細、および実験にあたって必要な準備は以下の通りです。

到達目標

1. 統合開発環境（Linux 上の Eclipse） を使って簡単な Java プログラムを開発できる。
2. オブジェクト指向プログラミングの 3 大特徴（カプセル化、継承、ポリモーフィズム）を理解し、そのプログラミングができる。
3. オブジェクトの連携による複合的なデータ表現（リストや木）を利用できる。

仮定する事前知識と準備

- C 言語の基礎を習得している。
- オブジェクト指向プログラミングの 3 大特徴の概念を理解している。
- 必要に応じて、Java の入門書の基本部分を読んでおく。

オブジェクト指向プログラミングの 3 大特徴については、「認知システム論」（3 年 1 学期）の第 2 回目「AI プログラミング」で解説します。また、3 日目の実験は、その第 3 回目および第 4 回目で扱う「探索」を題材としていますので、必要に応じて、それらの講義資料を参考にしてください。

Java の基本部分は C 言語と共通しているため、この指導書ではごく簡単な説明に留めています。C 言語との共通部分あるいはわずかな相違については「情報工学演習 I」（3 年 1 学期）の「認知システム論」の第 1 回目で解説します。指導書の中では、主にオブジェクト指向特有の言語機能とツールについて説明しますが、Java の全体像を理解するために、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 の言語仕様はここ 10 年間でかなり進歩しています。本実験で扱うプログラムは基本的には JDK 1.2 と呼ばれるバージョンで記述可能なものになっていますが、一部に最新の機能（JDK 5.0 以降のもの）を使用していますので、Java の入門書を入手するときには、なるべく JDK 5.0 に対応したもの、少なくとも JDK 1.2 に対応したものを選ぶようにしてください。

1　1 日目：Java 入門

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、簡単なプログラムを作成できることを目標とします。

1.1　文字列を出力するプログラムの解説

まず、Java プログラムの基本的な構造を理解するため、以下の文字列を出力する簡単なプログラムについて、その意味を各行毎に解説します。

ソースコード 1　HelloWorld.java

```
1  /*
2   HelloWorld.java
3   文字列を出力する
4   */
5  public class HelloWorld{
6          // プログラム起動時に呼ばれるメソッド
7          public static void main(String[] args){
8                  System.out.println("Hello,␣world!");
9          }
10 }
```

1 行目〜4 行目はコメントです。C 言語と同様、Java 言語においても /* がコメントの開始、 */ がコメントの終了を意味し、これらを用いて複数行に渡るコメントを記述することができます。

5 行目は、これから HelloWorld という名前のクラスを定義するという意味です。行頭の public は、このクラスが他のソースファイルから自由にアクセスできることを意味しています。

Java プログラムを実行するためには、一番始めに実行される main という名前の特別なメソッドを定義する必要がありますが、このメソッドを定義するクラスは、このように public なクラスとする必要があります。クラスの具体的な定義内容は中括弧（{}）の中に書きます。

6 行目はコメントです。// と書くと、そこから行末までがコメントとなります。コメントを 1 行だけ書く場合は、こちらの書き方が便利です。

7 行目は、クラス HelloWorld の内容として、C 言語の main 関数に相当する main メソッドを定義する見出しの部分です。初心者の Java プログラマはこの行の意味を深く知る必要はなく、おまじないのように基本的にこの通りに書くものだと思ってください。やや詳しく理解したい人のための説明を脚注[*1]に書いておきます。

8 行目は、メソッド main の動作として、標準出力に文字列を出力する println メソッドを呼び出すという意味です[*2]。println メソッドは、引数として与えられた文字列を出力し、その後さらに改行文字を出力するメソッドです。

9 行目、10 行目はそれぞれ main メソッド、HelloWorld クラスの定義がここで終わりであることを意味する中括弧です。

1.2　Eclipse の基本的な使い方

次に、Eclipse を使用して、前節で解説したプログラムを作成してみます。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。

1. プロジェクトの作成
2. ソースコードの作成
3. プログラムの実行設定と実行

まず、eclipse の起動方法と最低限必要な設定について述べ、次に、プログラムの作成の流れを順に説明します。

1.2.1　Eclipse の起動と設定

Eclipse の起動方法は以下の通りです。

1. 左上の「アプリケーション」から、「アクセサリ」→「GNOME 端末」を選択する。
2. 起動したターミナルで、eclipse を起動するコマンド「eclipse」を実行する。
3. 「ワークスペースの選択」ダイアログが表示されるので、eclipse 上で作成されるプログラムの保存場所を適当

[*1] 行頭の public は、このメソッドが他のクラスから自由にアクセスできることを意味しています。また、次の static は、このメソッドがインスタンスを作成しなくても実行できることを意味しています。次の void 以降では、メソッドの戻り値の型、名前、引数の型を定義しており、この定義部分は C 言語とほぼ同様の文法となっています。すなわち、メソッドは値を返さず (void)、名前は main であり、引数として String 型の配列 (String[]) を受け取るということを意味しています。

[*2] Java 言語では OS の標準出力を表すオブジェクトとして、System クラスのフィールド out が定義されています。out は PrintStream クラスのインスタンスです。

に設定し（基本的にはデフォルトで構わない)、「この選択をデフォルトとして使用し、今後この質問を表示しない」にチェックを入れ、「OK」を選択する。プログラムの保存場所は、デフォルトではホームディレクトリ直下の「workspace」ディレクトリとなっている。

デスクトップに起動用ショートカットを作成する場合は、以下のようにします。

1. デスクトップを右クリックし、「ランチャの生成」を選択する。
2. 「タイプ」の項目は「アプリケーション」を選択し、「名前」の欄に「eclipse」、「コマンド」の欄に「eclipse」を入力する。
3. アイコンを設定する場合は、左側の「なし」と書かれているボタンをクリックすると「アイコンを参照」というウィンドウが出現するので、一番上の欄に「/usr/local/eclipse」と入力し、下に現れる「icon.xpm」を選択し「OK」を押す。

eclipse が起動したら、以下の手順に従って設定を行ってください。この設定が正しく行われないと、プログラムのコンパイルや実行ができませんので、注意して下さい。

1. メニューから「ウインドウ」→「設定」を選択する。
2. 左側のメニューの「Java」をダブルクリックする。
3. 「Java」のサブメニューの「コンパイラー」を選択する。
4. 一番上の「コンパイラー準拠レベル」を「6.0」に設定する。

1.2.2 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、基本的に作成するプログラムの内容ごとにプロジェクトを作成することにします。

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（ここでは PrintString とする)、「次へ」を選択する。
4. ビルド設定は特に変更せず、「終了」を選択する。

以上でプロジェクトの作成は完了です。「ようこそ」ウィンドウが表示されている場合にはこれを閉じ、パッケージ・エクスプローラー（Eclipse ウインドウ左側の欄）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。

1.2.3 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。
2. 親フォルダとして作成したプロジェクト・フォルダ名を選択し、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。すなわち、今回は”HelloWorld.java”を入力する。最後に「終了」を選択する。
3. 作成したファイルの内容を編集するための画面が表示されるので、そこにソースコード 1 の内容を入力する。
4. メニューから「ファイル」→「保管」を選択し、ソースファイルの内容を保存する。

Eclipse は、入力された左丸括弧'(' に対応する右丸括弧')' を自動的に挿入します。括弧の中の内容を入力し終え、挿入された右丸括弧の次の位置にカーソルを移動したい場合は、TAB キーを押します。

ソースコードを入力中、Eclipse は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。Eclipse は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。

さらに Eclipse では、ソースコードを入力中に Ctrl+Space キーを押すと、その文脈で利用できるコードを補完してくれたり、Ctrl+Shift+Space キーを押すと、メソッドのパラメータヒントをポップアップで表示してくれるなど、この他にも多くの便利な機能が備わっているので、興味のある人は調べてみてください。

1.2.4 プログラムの実行設定と実行

最後に、作成したソースプログラムを実行するための設定を行います。Eclipse における、プログラムを実行するための設定は**構成**と呼ばれ、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」→「構成および実行」を選択する。
2. 左にある「Java アプリケーション」をダブルクリックする。
3. メイン・クラスの入力欄の右の「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「コンソール」欄に表示されます。

1.3 オブジェクト指向プログラムの例

次に、Java 言語を用いたオブジェクト指向的プログラミングの第一歩として、二次元空間上の座標をクラスとして表現するプログラムを作成します。

オブジェクト指向プログラミングの基礎は講義で一通り説明を受けていると思いますが、この実習書においても、必要に応じて補足的な説明を行います。クラスとは、データとそれを操作する関数（メソッド）をひとまとめにしたもので、C 言語の構造体を拡張したものと考えることができます。クラスはプログラムにおける何らかのオブジェクト、いわゆる「もの」を表現するために定義されます。ここで、表現するオブジェクトは、プログラムの実行時に具体的に目に見えるものである必要はありません。オブジェクト指向プログラミングでは、プログラムを構成する様々な概念をオブジェクトとして取り扱います。

1.3.1 クラスの定義

まず、座標を表現するクラス Point を定義します。ここでは新しいプログラムを作成するので、前節で作成したプロジェクトは一旦閉じ、新たにプロジェクトを作成すると良いでしょう。プロジェクトを閉じるには、画面左のパッケージ・エクスプローラーで閉じたいプロジェクト名を右クリックし、「プロジェクトを閉じる」を選択します。

クラスを定義するには、前節と同様にまずソースファイルをプロジェクトに追加し、そのファイルにクラスを定義するソースコードを書いていくという手順になりますが、Eclipse にはこれらの作業を簡易化するための機能が用意されていますので、今回は以下の手順に従い、その機能を利用することにします。

1. メニューから「ファイル」→「新規」→「クラス」を選択する。
2. 「ソース・フォルダー」が現在のプロジェクト名になっていることを確認する。

型名	値の範囲
boolean	true と false （真理値）
byte	-128 以上 127 以下の整数
short	-32768 以上 32767 以下の整数
int	-2147483648 以上 2147483647 以下の整数
long	-9223372036854775808 以上 9223372036854775807 以下の整数
char	0 以上 65535 以下の整数（文字コード）
float	IEEE754 規格の 32 ビット浮動小数点数
double	IEEE754 規格の 64 ビット浮動小数点数

表 1　Java の基本データ型

3. 「名前」の欄に作成するクラス名を入力する。ここでは Point と入力する（先頭の P は大文字です）。
4. 「修飾子」「スーパークラス」「インターフェース」の欄でこのクラスのアクセス権限や継承関係に関する設定が行える。ここでは設定を変更しない。
5. 「作成するメソッド・スタブの選択」では、よく実装されるメソッドのうち、定義の一部を自動生成したいものを選択する。ここでも設定を変更しない。
6. 「終了」を選択する。

以上で、現在のプロジェクトに Point.java という名前のファイルが追加され、ファイルの内容として Point クラスの定義に必要な最低限のソースコードが入力された状態になります。

1.3.2　フィールドの定義

次に、具体的な Point クラスのメンバー定義、すなわちフィールド（変数）やメソッド（関数）の定義をソースコードに追加します。ここでは、表現する二次元座標の各成分は整数とし、フィールドとして整数型の変数 x, y を Point クラスに追加します。Java 言語では、C 言語と同様、整数を表す変数の型として int 型が定義されています。表 1 に示した、int を含む 8 種類の型を Java の**基本データ型**といいます。boolean 型は C 言語には存在しない型で、真か偽かのどちらかを表す真理値を表現するための型です。boolean 以外の型は C 言語にも存在し、役割もほぼ同じです。Point クラスのメンバとして int 型のフィールド x,y を定義するには、以下のように書きます。

ソースコード 2　Point クラスのフィールドの定義

```
public class Point{
        public int x,y;
}
```

変数の定義の構文は C 言語とほぼ同じですが、先頭にアクセス制御のための修飾子（この場合は public）を記述できる点が異なります。修飾子の意味は後ほど説明します。

1.3.3　クラスの使用

最後に、main メソッドを定義し、main メソッドにこのクラスを使用するコードを書きます。まず、Point クラスを定義した時と同様の手順でクラス Main を作成し、そのクラスに main メソッドを定義します。

Point クラスを利用するプログラムの例として、main メソッドの内容として Point クラスのインスタンスの各フィールドに値を設定し、その値を出力するという処理を追加したものを以下に示します。なお、以降に示されるソースコードでは、main メソッドなどの自動生成時に Eclipse が生成したコメントは削除してあります。

ソースコード 3　Point クラスの利用例

```
1 public class Main {
2         public static void main(String[] args) {
```

```
3                 Point p = new Point();
4                 p.x = 10;
5                 p.y = 20;
6                 System.out.println("(x,y) = (" + p.x + "," + p.y + ")");
7         }
8 }
```

このプログラムを簡単に説明します。3 行目は Point クラスのインスタンスを参照するための変数 p を宣言し、同時に Point クラスのインスタンスを new キーワードにより生成し、そのインスタンスへの参照を p に代入しています。ここで宣言した p はクラス Point のインスタンスを参照するための変数です。このようなあるクラスのインスタンスを参照するための変数の型を**参照型**と呼び、int や double といった基本データ型と区別します。（図 1）参照型はインスタンスへの参照を保存する型であるため、C 言語におけるポインタに相当します。4,5 行目はインスタンス p のフィールド x,y に値を設定しています。フィールドへアクセスするには C 言語の構造体のメンバへのアクセスと同様、p.x のようにインスタンス名とフィールド名をドットで結びます。6 行目は p の内容を読みやすい形で出力しています。C 言語と異なり、Java 言語では文字列同士、もしくは文字列と他の基本データ型の値を + 演算子で結合し、新たな文字列を生成することができます。

このプログラムの実行結果は以下のようになります。

```
(x,y) = (10,20)
```

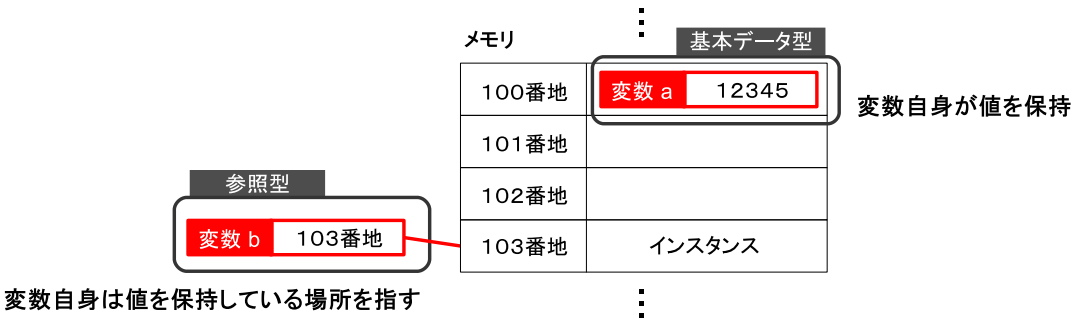

図 1　基本データ型変数と参照型変数

1.3.4　メソッドの定義

次に、Point クラスにメソッドを定義します。まず、コンストラクタと呼ばれる特別なメソッドを定義します。コンストラクタとはインスタンスの生成時に呼ばれるメソッドで、主にフィールドの初期化処理を行うために用いられます。コンストラクタを定義するには、戻り値の型を省略し、クラス名と同じ名前のメソッドを定義します。ここでは、引数が何も与えられなかった場合に 0 でフィールドを初期化するコンストラクタと、x,y に対応する 2 つの int 型の引数が与えられた場合に、それらの値でフィールドを初期化するコンストラクタの 2 種類を以下のように定義します。

ソースコード 4　コンストラクタの定義例

```
public class Point {
        public int x,y;

        public Point(){
                x = 0;
                y = 0;
        }

        public Point(int x, int y){
                this.x = x;
                this.y = y;
```

```
        }
}
```

一つ目のコンストラクタでは、フィールド x と y に 0 を代入しています。このように、メソッドの中ではインスタンス名とドットで結ぶことなしにフィールドにアクセスすることができます。このときにアクセスされるフィールドは、このメソッドが呼び出されたインスタンス自身のフィールドです。

二つ目のコンストラクタでは、引数の名前 x,y がフィールド名と重なっています。このような場合は引数の名前が優先されるため、インスタンスのフィールドにアクセスすることを明示するために this キーワードを用いています。this キーワードは、「メソッドが呼び出されたインスタンス自身」を意味します。

ここでは、同じ名前で複数のメソッドを定義しています。これをメソッドの**オーバーロード**と呼びます。オーバーロードされた複数のメソッドのうち実際にどれが実行されるかは、引数の数や型で決定されます。オーバーロードはコンストラクタに限らず、どのようなメソッドについても行うことができます。

次に、原点からのユークリッド距離を求める distance メソッドを定義します。距離は実数となるため、ここでは double 型を返すことにします。定義例は以下のようになります。

ソースコード 5　メソッド distance の定義例

```
public class Point {
        public int x,y;

        ...

        public double distance(){
                return Math.sqrt(x*x + y*y);
        }
}
```

ここで、Math.sqrt(...) は Math クラスで定義されている平方根を求めるための sqrt メソッドを呼び出す構文です。このような、「クラス名. メソッド名」の形で呼び出されるメソッドを**クラスメソッド**と呼びます。クラスメソッドは「インスタンス名. メソッド名」の呼び出し方と異なり、インスタンスを必要としません。その意味で、C 言語の「関数」とほとんど同じものです。メソッドの詳細な使い方を調べるには、メソッド名の上に入力カーソルを置き、F2 キーを押します。

最後に、インスタンスの内容を分かりやすい文字列に変換する、toString メソッドを定義します。toString メソッドは特別なメソッドの一つで、インスタンスの内容を文字列として画面に表示したり、インスタンスと文字列を + 演算子で連結するときなどに、自動的に呼び出されます。Point クラスのインスタンスは二次元座標なので、前節のテストプログラムの出力のように”(x,y)”という形式の文字列が得られるよう、toString メソッドを定義します。以下に定義例を示します。

ソースコード 6　メソッド toString の定義例

```
public class Point {
        public int x,y;

        ...

        public String toString(){
                return "(" + x + "," + y + ")";
        }
}
```

以上のメソッドを利用して、前節の Main クラスの main メソッドを書き直したものを以下に示します。

ソースコード 7　Point クラスのメソッドの利用例

```
1 public class Main {
2         public static void main(String[] args) {
3                 Point p = new Point(10,20);
4                 System.out.println("(x,y)␣=␣" + p);
```

```
5                 System.out.println("原点からの距離␣=␣" + p.distance());
6         }
7 }
```

Point クラスのインスタンスのフィールドの値は、コンストラクタの引数 (10,20) によって簡潔に設定されています。また、Point クラスで toString メソッドが実装されており、出力に先立って文字列を作成する必要がないため、println メソッドが何を出力するのかが分かりやすくなっています。実行結果は以下のようになります。

```
(x,y) = (10,20)
原点からの距離 = 22.360679774997898
```

1.4 API の利用

C 言語では、stdio.h や stdlib.h といったヘッダーファイルをインクルードし、コンパイル・リンクを行うことで、既に作成済みのプログラムの部品（ライブラリ）を利用する事ができます。Java 言語でも同様に、膨大な量のプログラム部品が Java 言語の標準ライブラリとして用意されています。ライブラリはクラスの集合として提供されており、それらは大まかな機能ごとに**パッケージ**としてグループ分けされています。このようなライブラリはアプリケーションを作成するためにプログラマに提供された、コンピュータを操作するためのインタフェースと考えられるため、*API(Application Programming Interface)* と呼ばれます。複雑なプログラムを誤りなく、かつ効率良く作成するためには、提供されている API をできるだけ活用し、自分でプログラムする範囲をできるだけ抑えることが重要となります。Java 言語の API のリファレンスは http://java.sun.com/javase/ja/6/docs/ja/api/index.html にあります。また、エディタ上でクラスやメソッドが選択されている状態で Shift+F2 キーを押すと、ブラウザが起動し該当のクラスやメソッドのリファレンスが表示されます。この機能を有効にするためには、各ライブラリごとにリファレンスへのパスを設定する必要があります。例えば、Java のランタイムライブラリについて設定を行う場合には、以下のように行います。

1. 画面左のパッケージ・エクスプローラーの中の、「JRE システム・ライブラリー [jdk1.6.0_05]」をダブルクリックする。
2. 「rt.jar」を右クリックし、プロパティーを選択する。
3. 出現するウィンドウ左の「Javadoc ロケーション」を選択し、「Javadoc ロケーション・パス」の欄に「http://java.sun.com/javase/ja/6/docs/ja/api」と入力し「OK」を選択する。

1.5 1 日目の課題

Point クラスに以下のメソッドを追加し、main メソッドに適切なコードを追加してテストしてください。

課題 1.1 int manhattan()

原点からのマンハッタン距離 $|x| + |y|$ を返す。

課題 1.2 void moveUp(), void moveDown(), void moveRight(), void moveLeft()

moveUp は y の値を 1 だけインクリメント (y++) する。同様に、moveDown は y−−、moveRight は x++、moveLeft は x−− する。

課題 1.3 boolean sameLocation(Point p)

点 p と同じ位置かどうかを判定する。点 q = (x,y)、点 p = (u,v) のとき、q.sameLocation(p) は、x=u かつ y=v のとき true、さもなくば false を返す。

課題 1.4 Point opposite()

原点に関して点対称の位置にある点を生成して返す。すなわち、点 q=(x,y) のとき、q.opposite() は点 (-x,-y) を生成して返す。

課題 1.5 Point relative(Point p)

点 p からの相対位置を表す点を生成して返す。点 q=(x,y)、点 p=(u,v) のとき、q.relative(p) は点 (x-u,y-v) を生成して返す。

課題 1.6 Point copy(Point p)

点 p のコピー（同じ位置をもつ点）を生成して返す。

※ヒント：単に「`return this;`」だけでは誤り。

2　2 日目：オブジェクト指向プログラミング

本節では、オブジェクト指向の 3 大特徴（カプセル化、継承、ポリモーフィズム）を理解し、それらを生かしたプログラミングができることを目標とします。

2.1　カプセル化

1 日目に作成した Point クラスでは、フィールドのアクセス制限の指定は、制限として最も緩い public を用いて public int x,y; としていました。こう指定すると、どのようなクラスからも、「オブジェクト.x」のような表記によって自由にこのフィールドの値を読み書きすることができます。しかし、オブジェクト指向プログラミングでは、クラスを利用する側が直接フィールドの値を操作するのではなく、メソッドを介して間接的に操作するのが望ましいとされています。クラスを構成するデータ（フィールド）を隠蔽し、クラスを利用するためのインタフェース（メソッド）のみを提供することを、クラスの**カプセル化**と呼びます。カプセル化により、フィールドの内容が更新されるときは必ずメソッドが実行されるため、メソッドに渡された引数のチェックを行う事で、フィールドに不適切な内容が代入されないことを保証できます。また、将来のバージョンアップの際に、高速化や高機能化のためにクラスのフィールドの構成を変更する必要が生じた場合にも、その変更はクラスの外部に対しては隠蔽されるため、外部に公開する機能（メソッド）の振る舞いを保ちながら、内部の実装の詳細を変更することができます。すなわち、クラスを利用する側のソースコードを修正することなく、クラスを改良することができます。このように、カプセル化はクラス部品の信頼性や独立性を保つために非常に重要な役割を果たします。

まず、Point クラスをカプセル化するため、フィールドにより厳しいアクセス制限をかけ、フィールドの値を取得するメソッドであるゲッターと、フィールドの値を設定するメソッドであるセッターの定義を行います。

2.1.1　クラスメンバのアクセス制限

Java 言語では、アクセス制限のレベルを指定する**アクセス指定子**として、public、protected、default（指定なし）、private の 4 種類があります。表 2 にアクセス指定子と、その指定子が設定されたメンバにアクセス可能なクラスをまとめます。

アクセス指定子	アクセス可能なクラス
public	全てのクラス
protected	サブクラスおよび同じパッケージのクラス
default	同じパッケージのクラス
private	同じクラス

表 2　アクセス指定子

基本的に、フィールドは全て private に設定し、次に詳述するゲッターとセッターを定義するのが良いでしょう。また、メソッドについても、他のメソッドを定義するための補助的な役割のものは、private に設定することによって、外部に対して隠蔽するのが良いでしょう。

Point クラスのフィールド x,y の定義において、アクセス指定子を private とする場合のソースコードは、以下のようになります。

ソースコード 8　フィールドの private 宣言の例

```
public class Point {
        private int x, y;
        ...
}
```

2.1.2 ゲッターとセッターの定義

次に、private に設定したフィールドの値を取得するためのゲッターメソッドと、フィールドの値を更新するためのセッターメソッドを定義します。ゲッターおよびセッターの名前は、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。このように、ゲッターとセッター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 の機能を用いてゲッターとセッターを定義します。

1. 編集中のソースファイルを Point.java とし、入力カーソルを Point 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（x と y）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。

これにより、getX、setX、getY、setY の 4 つのメソッドが定義されます。

2.1.3 カプセル化の応用例：計算結果のキャッシュ

ここで、カプセル化が有効な例として、メソッドの振る舞いを変えずに、メソッド distance の計算結果をキャッシュする機能を加えることを考えます。メソッド distance は非常にシンプルなメソッドですが、乗算 2 回と加算 1 回に加え、平方根の計算が行われるため、フィールドの値が更新される頻度よりもこのメソッドが呼ばれる頻度の方が高い場合は、計算結果をキャッシュと呼ばれる内部変数に保存し、座標値が変更されない限りキャッシュの内容を返すことで計算時間が短縮されます。

まず、以下の 2 つのフィールドを private なメンバとして追加します。

- キャッシュ結果を保存する、double 型の変数 currentDistance
- 「キャッシュの値が現在の座標値を反映している」ことを意味する、boolean 型の変数 updated

次に、これらのフィールドが常に正しい値に保たれるよう、以下のようにコンストラクタ、distance、セッターの内容を変更します。

ソースコード 9　distance メソッドの計算結果のキャッシュ

```
public class Point {
	private int x, y;
	private double currentDistance;
	private boolean updated;

	public Point(){
		setX(0);
		setY(0);
	}

	public Point(int x, int y){
		setX(x);
		setY(y);
	}

	public double distance(){
		if( !updated ){
			currentDistance = Math.sqrt(x*x + y*y);
			updated = true;
		}
		return currentDistance;
	}
```

```
        ...

        public void setX(int x) {
                this.x = x;
                updated = false;
        }

        ...

        public void setY(int y) {
                this.y = y;
                updated = false;
        }
}
```

クラスが適切にカプセル化されているので、外部に提供しているメソッドの振る舞いを変えないようにクラス内部の修正を行うことで、クラス利用者のソースコードを修正することなく、クラスの改良を行うことができます。また、このキャッシュの仕組みにおいて重要な点は、アクセス制限とセッターメソッドによって、利用者にメソッドによるフィールドの更新を強制することにより、フィールドの値の更新と同時に必ず特定の処理（キャッシュ値の再計算）が行われることが保証されている点です。カプセル化は、このようにクラス内部のデータの整合性を保証するためにも重要です。

2.2 クラスの継承

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**があります。継承の特徴は元々のクラス（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。

2.2.1 簡単なクラスの継承例

まず、継承の簡単な例として、既に作成した Point クラスを拡張し、座標に加えてその座標の色を保持する事ができる ColorPoint クラスを定義します。ColorPoint クラスは、新しいメンバとして色名を保持する String 型のフィールド color を持つものとします。

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

```
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 Point クラスを継承した ColorPoint 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「ColorPoint」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「Point」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「Point」を選択して「OK」を選択する。
5. 「終了」を選択する。

これにより、ColorPoint 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。ColorPoint クラスの内容はまだ何も書かれていないため、この時点では、ColorPoint クラスは Point クラスと全く同じものと

なっています。まず、以下のように新しいフィールド color を追加します。

ソースコード 10　ColorPoint クラスの定義

```
1 public class ColorPoint extends Point{
2         private String color;
3 }
```

次に、Point 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[*3]。ColorPoint クラスでは、Point クラスと同様の 2 種類のコンストラクタに加え、3 つ目のコンストラクタとして、座標と色名の両方を設定できるコンストラクタを定義します。

ソースコード 11　ColorPoint クラスのコンストラクタの定義

```
1 public class ColorPoint extends Point{
2         private String color;
3
4         public ColorPoint(int x, int y, String color){
5                 super(x,y);
6                 this.color = color;
7         }
8 }
```

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、引数 x, y を用いて Point クラスのコンストラクタを呼び出し、座標の初期化をしています。それに加え、新たに追加したフィールド color の初期化を行っています。

次に、新しいフィールド color の内容を toString メソッドによって得られる文字列に反映させます。ColorPoint クラスのインスタンスを表す文字列は、”色名 (x,y)”のように、座標の前に色名が付加されたものとします。

ソースコード 12　ColorPoint クラスの toString メソッドの定義

```
public class ColorPoint extends Point{
        ...
        public String toString(){
                return color + "␣" + super.toString();
        }
}
```

これで、Point クラスから継承された toString メソッドが、上記の新しい定義によって上書きされたことになります。ここで、super.toString() はスーパークラスの toString メソッドを意味しており、これを用いることで、作成する文字列のうち、座標の文字列表現については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。

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**と呼びます。Point クラスを継承しているため、distance や moveUp といった Point 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自由に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。

Point クラスと ColorPoint クラスの継承関係は、図 2 のようなクラス図で表されます。図において、メンバの左側の-と + はアクセス指定子を表しており、それぞれ private、public に対応します。クラス図では継承を表現するため、サブクラスからスーパークラスの方向に、実線で矢印を引くことになっています。

[*3] 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。

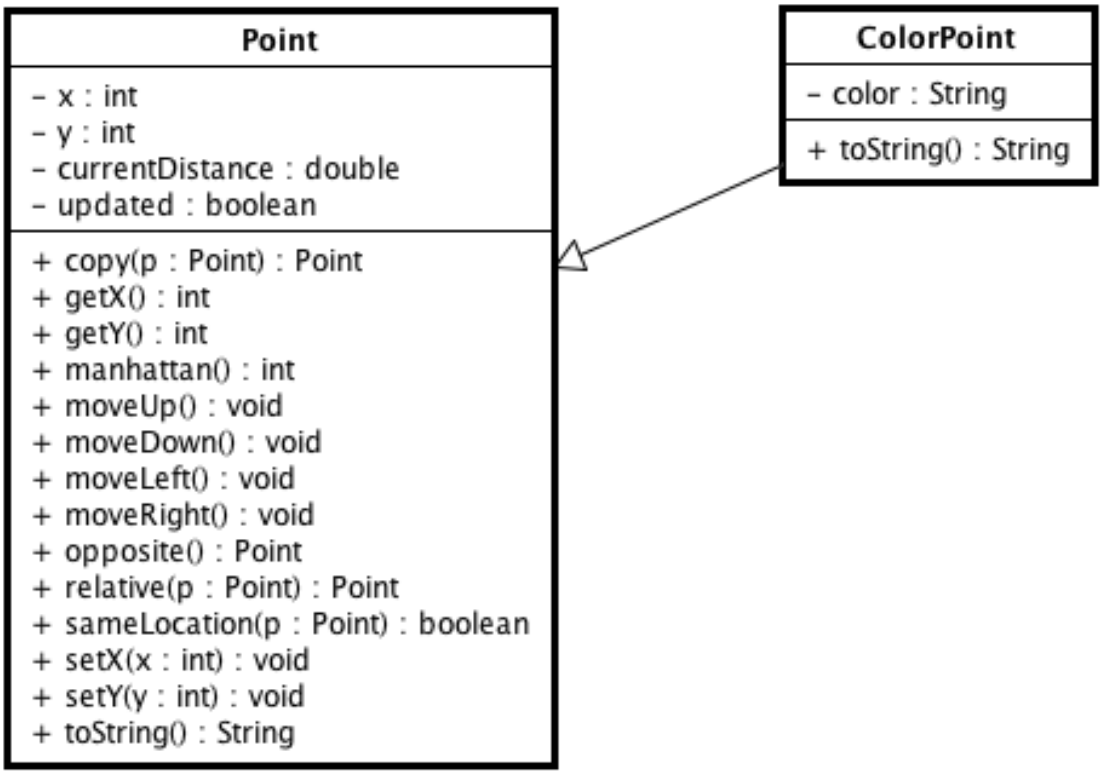

図 2　Point クラスと ColorPoint クラスのクラス図

2.2.2　Object クラスのオーバーライド

Java では、すべてのクラスは直接的あるいは間接的に Object というクラスのサブクラスになることを補足説明しておきます。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 Object がスーパークラスとなります。したがって、

```
public class Point { ... }
```

のように記述された Point クラスは、実際には

```
public class Point extends Object { ...}
```

と記述したのと同様に、Object クラスのサブクラスとして定義されます。また、

```
public class ColorPoint extends Point { ... }
```

で定義された ColorPoint クラスは、直接的には Object のサブクラスではありませんが、スーパークラスの Point が Object のサブクラスなので、間接的に Object のサブクラスとなります。（つまり、サブクラスのサブクラスもサブクラスです。）Object クラスでは、すべてのオブジェクトが備えるべきメソッドが定義されています。すべてのクラスは Object のサブクラスなので、それらのメソッドはすべてのクラスに継承されますが、サブクラスの定義の中で必要に応じてオーバーライドして、メソッドの具体的な内容を変更することが期待されています。そのうち、つぎの２つは良く用いられるものです。

- public String toString( )
- public boolean equals(Object obj)

1 つめの toString メソッドは、すでに 1.3.4 節で簡単に紹介したものです。このメソッドは、オブジェクトの情報を適切に表現する文字列を返すようにオーバーライドできます。これにより、文字列連結演算子（＋）や System.out.println メソッド等の中で、必要に応じて toString メソッドが呼び出され、適切な文字列が生成されま

す。2 つめの equals メソッドは、2 つのオブジェクトの値が同じかどうかを判定します。Object クラス内で事前に定義されているこのメソッドの内容は、2 つのオブジェクトが「同一」かどうかを判定します。つまり、2 つのオブジェクトが、メモリー内の同じ場所に存在するかどうかを「参照」（ポインタ）の同一性によって判定します。しかし、サブクラスでは、通常、オブジェクトの値（内容）に踏み込んでチェックし、応用目的にかなうような定義によってオーバーライドします。たとえば、Point クラス用の定義は、課題で作成した sameLocation メソッドを利用して、つぎのようにすればよいでしょう。

```
public boolean equals(Object obj) {
        return sameLocation( (Point)obj );
}
```

引数の obj は Object 型と宣言されていますが、応用上は、実際には Point 型のオブジェクトだけが渡されてくると仮定します。その情報をメソッド内で (Point)obj のように型変換（キャスト）によって明示します。sameLocation の引数は Point 型として設計されていたので、これで引数の型が一致します。ここで述べた equals メソッドのオーバーライドは、3 日目の実験でも活用されます。

2.2.3 継承したクラスの利用例

次に、継承したクラスの利用例として、サブクラス ColorPoint と、スーパークラス Point を一次元配列に混在させて使用してみます。

Java 言語では、以下のような構文で一次元配列が利用できます。

ソースコード 13　一次元配列の例

```
1 int[] nums = new int[5];
2 nums[0] = 1;
3
4 Point[] points = new Point[3];
5 points[0] = new Point(3,4);
6 points[0].moveUp();
```

1〜2 行目では、長さ 5 の int 型配列 nums を作成し、その 0 番目の要素に 1 を代入しています。このように、要素の型が基本データ型の場合は、C 言語の配列とほぼ同じ感覚で配列を使用できます。これに対し、4〜6 行目は、参照型の配列を利用する例です。4 行目のように、参照型の配列も構文的には基本データ型の場合と同様に宣言します。このように宣言された配列の要素は、「Point p」とだけ定義された変数 p のように、まだ特定のインスタンスを指し示していない状態になっています。つまり、配列の全ての要素には null が代入されています。このため、参照型の配列を宣言した後に、5 行目で行っているように各要素に具体的なインスタンスへの参照を代入する必要があります。この例では、0 番目の要素のみに新しいインスタンスへの参照を代入しています。全ての要素に新しいインスタンスを割り当てるには、以下のように配列の要素数を表す length フィールドを利用し、for ループを用いて全ての要素を初期化します。

ソースコード 14　一次元配列の初期化例

```
1 Point[] points = new Point[3];
2 for(int i=0;i<points.length;i++){
3         points[i] = new Point();
4 }
```

以上を踏まえて、以下のような main メソッドを作成します。

ソースコード 15　一次元配列に異なるクラスのインスタンスを混在させる例

```
1 public static void main(String[] args){
2         Point[] points = new Point[4];
3         points[0] = new Point(1,2);
4         points[1] = new ColorPoint(0,0,"red");
5         points[2] = new Point();
6         points[3] = new ColorPoint(2,5,"blue");
7
```

```
8         for(int i=0;i<points.length;i++){
9                 System.out.println(points[i]);
10        }
11  }
```

配列の要素の型は Point ですが、いくつかの要素にはサブクラスである ColorPoint クラスのインスタンスを代入しています。このように、クラスのインスタンスは、スーパークラスの型の変数に代入することができます。これにより、配列に異なるクラスのインスタンスを混在させることができるため、配列の全ての要素に対し同じメソッドを呼び出すという簡潔な形を取りながら、実際には要素ごとに異なる動作をさせることができます。このメソッドの実行結果は、以下のようになります。

```
(1,2)
red (0,0)
(0,0)
blue (2,5)
```

Java 言語では、スーパークラスとして複数のクラスを指定する**多重継承**は禁止されています。その代わりとして、次節で取り上げる**インタフェース**と呼ばれる仕組みが存在します。

2.3 インタフェースとポリモーフィズム

インタフェースとは、クラスと同様、新しい型を定義するための仕組みです。インタフェースは、どのようなメソッドが存在するかだけが定義され、具体的な処理内容は定義されていない、フィールドを持たない型のことをいいます。具体的な内容が定義されていないメソッドを**抽象メソッド**といいます。抽象メソッドが存在するため、インタフェースはクラスのように直接インスタンスを生成することはできず、何らかのクラスがインタフェースのメソッドの具体的な内容を実装する必要があります。あるクラス C が、あるインタフェース I で定義されている全てのメソッドを実装するとき、クラス C はインタフェース I を**実装する**といいます。インタフェースは、「〜というメソッドが実装されている」という形でクラスの性質を定義するもので、クラスはインタフェースを実装することで、そのインタフェースが定義する性質を備えていることを表現します。Java 言語では、多重継承（複数のクラスの継承）は許されていませんが、あるクラスが複数のインタフェースを実装することは許されています。

サブクラスのインスタンスをスーパークラスの変数に代入できるのと同様に、あるインタフェースが実装されている任意のクラスのインスタンスは、そのインタフェース型の変数に代入する事ができます。これは、異なるクラスのインスタンスも、「特定のメソッドが実装されている」という意味で同一視するということです。このように同一視された全てのインスタンスに対して同じメソッドを呼びだすことで、インスタンスごとに異なる動作をさせることができます。このような、特定のメソッドの呼び出しによる具体的な動作がインスタンスごとに異なることを**ポリモーフィズム（多相性）**と呼びます。

2.3.1 インタフェースの定義

まず、簡単なインタフェースの例として、Point クラスで定義した移動に関する 4 つのメソッド

- void moveUp()
- void moveDown()
- void moveLeft()
- void moveRight()

を、Movable という名前のインタフェースとして定義することにします。このインタフェースは、名前通り「動くことができる」という性質を表現しています。

クラスと同様、Eclipse にはインタフェースを作成するための機能があります。それを利用し、以下の手順で Movable インタフェースを定義します。

1. メニューから「ファイル」→「新規」→「インターフェース」を選択する。

2. 「ソース・フォルダー」が現在のプロジェクト名になっていることを確認する。
3. 「名前」の欄に作成するインタフェース名を入力する。ここでは Movable と入力する。
4. 「終了」を選択する。

クラスと同様、インタフェースも一つのインタフェースごとにソースファイルが作成されます。このソースファイルに、4 つのメソッドの定義を加えます。

ソースコード 16　Movable インタフェース

```
1 public interface Movable {
2         void moveUp();
3         void moveDown();
4         void moveRight();
5         void moveLeft();
6 }
```

インタフェースはクラスと同様の構文で定義しますが、class キーワードの代わりに interface キーワードを用います。メソッドは具体的な実装を持たない抽象メソッドであるため、フィールドの定義と同様に、セミコロンでメソッドの定義を終えます。インタフェースのメソッドのアクセス制限レベルは public でなければならないため、修飾子を何も書かない場合は、自動的に public となります。

インタフェースと、それを実装するクラスの関係は、図 3 のようなクラス図で表されます。インタフェースは、クラスと同様の形を取りますが、名前の部分に「<<interface>>」をつけます。クラス図ではインタフェースの実装を表現するため、実装するクラスからインタフェースの方向に、波線で矢印を引くことになっています。また、インタフェースのメソッドは抽象メソッドであるため、メソッド名が斜体で書かれています。

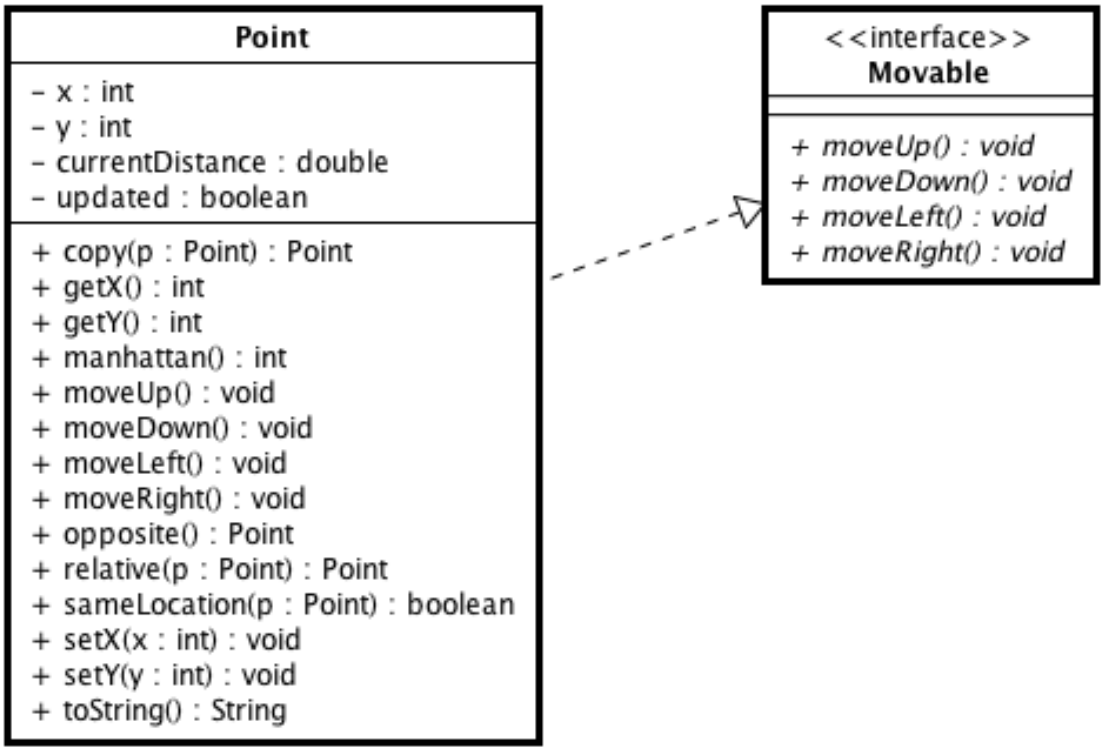

図 3　Movable インタフェースの実装のクラス図

2.3.2　インタフェースの実装

次に、作成した Movable インタフェースを、2 つのクラスに実装してみます。

まず、Point クラスに Movable インタフェースを実装します。Movable インタフェースに含まれる 4 つのメソッ

ドの具体的な実装は課題 1.2 の内容となっていますので、ここではそのような具体的な実装が済んでいると仮定し、Point クラスが Movable インタフェースを実装することを示す宣言の追加のみを行います。クラスの継承には extends キーワードを用いましたが、インタフェースの実装では、implements キーワードを用います。

```
class Point implements Movable{
        ...
}
```

インタフェースの具体的なメソッドが定義されていても、このような宣言がないとインタフェースを実装していることになりません。また、インタフェースのメソッドは public でなければならないため、Point クラスのこれらのメソッドが public として定義されていなかった場合、コンパイルエラーとなりますので注意してください。

次に、動けるものの 2 つ目の例として、カニを表す Crab クラスを定義し、同様に Movable インタフェースを実装します。Crab クラスは、現在位置を保持する double 型のフィールド x と、歩幅を表す double 型のフィールド step を持ちます。カニは moveLeft、moveRight メソッドによって、step の値だけ左、右に動くものとします。moveUp、moveDown では状態は変化しないものとします。

クラスの継承と同様に、Eclipse の支援機能を用いてインタフェースを実装する手順を示します。Crab クラスの空の定義が作成された状態で、Movable インタフェースを実装する手順は以下の通りです。

1. Crab クラスの定義の一番最初の部分「public class Crab{」を「public class Crab implements Movable{」に変更する。
2. Crab.java が表示されている状態で、メニューから「ソース」→「メソッドをオーバーライド/実装」を選択する。
3. Movable インタフェースの 4 つのメソッド全てにチェックが入っていることを確認し、「OK」を選択する。

これにより、Movable インタフェースの 4 つのメソッドの定義が、内容が空の状態で追加されます。これらのメソッドの具体的な内容、コンストラクタおよび toString メソッドを追加した Crab クラスの定義を以下に示します。

ソースコード 17　Crab クラスの定義

```
1  public class Crab implements Movable {
2          private double x;
3          private double step;
4
5          public Crab(int x, double step){
6                  this.x = x;
7                  this.step = step;
8          }
9
10         public void moveUp() {}
11
12         public void moveDown() {}
13
14         public void moveLeft() {
15                 x -= step;
16         }
17
18         public void moveRight() {
19                 x += step;
20         }
21
22         public String toString(){
23                 return "Crab(" + x + ":step=" + step + ")";
24         }
25 }
```

2.3.3 ポリモーフィズムの実行例

それでは、ポリモーフィズムの実行例として、これまで定義した Movable インタフェースを実装する 2 種類のクラスを一つの配列に混在させ、全ての要素に同じ名前のメソッドを呼び出すプログラムを作成します。これにより、ポリモーフィズムによって、異なるクラスのインスタンスは異なる振る舞いをすることを確認します。

一次元配列に 2 種類のクラスのインスタンスを混在させ、それらに moveUp、moveRight メソッドを呼び出し、状態の変化を出力する main メソッドは、以下のようになります。

ソースコード 18　ポリモーフィズムの実行例

```
1  public static void main(String[] args){
2          Movable objs[] = new Movable[3];
3          objs[0] = new Point(3,4);
4          objs[1] = new Crab(5.0, 0.5);
5          objs[2] = new Crab(0.0, 1.2);
6
7          System.out.println("moveUp()");
8          for(int i=0;i<objs.length;i++){
9                  System.out.print(objs[i] + "␣-->␣");
10                 objs[i].moveUp();
11                 System.out.println(objs[i]);
12         }
13
14         System.out.println("moveRight()");
15         for(int i=0;i<objs.length;i++){
16                 System.out.print(objs[i] + "␣-->␣");
17                 objs[i].moveRight();
18                 System.out.println(objs[i]);
19         }
20 }
```

このように、スーパークラス型の変数にサブクラスのインスタンスを代入できるのと同様、インタフェース型の変数には、そのインタフェースを実装したクラスのインスタンスを代入することができます。これにより、インタフェース型を要素型とする配列に異なるクラスのインスタンスを混在させることで、それら全てに同じメソッドを呼び出すという処理を簡潔に記述することができます。このメソッドの実行結果は、以下のようになります。

```
moveUp()
(3,4) --> (3,5)
Crab(5.0:step=0.5) --> Crab(5.0:step=0.5)
Crab(0.0:step=1.2) --> Crab(0.0:step=1.2)
moveRight()
(3,5) --> (4,5)
Crab(5.0:step=0.5) --> Crab(5.5:step=0.5)
Crab(0.0:step=1.2) --> Crab(1.2:step=1.2)
```

2.4 2 日目の課題

課題 2.1 Point クラスに整数型の 2 つのフィールド distanceCalled と sqrtComputed を追加し、distanceCalled は distance メソッドの呼ばれた回数、sqrtComputed は distance メソッドの内部で sqrt メソッドが呼ばれた回数を保持するように、distance メソッドを変更してください。これによって、Point クラスを利用しているソースコードに影響を与えることなく、これらの統計データを取ることができます。

課題 2.2 Point クラスのサブクラスとして、グレイスケール（白黒の色の濃さ）をもつ点を表すクラス GrayPoint を定義してください。このクラスは 0〜255 のグレイスケール値を保持するためのフィールド grayScale を持つものとします。つぎに、Point 型の一次元配列に、Point、ColorPoint、GrayPoint のインスタンスを混在させて格納し、for ループでその内容を表示してください。

課題 2.3 Movable インタフェースを実装した、以下の仕様を満たす Elevator クラスを作成してください。

- このエレベータは地下 3 階から地上 11 階の間を動きます。
- moveUp および moveDown のメソッドによって、それぞれ階を 1 つ上昇または下降します。ただし、11 階で moveUp、あるいは地下 3 階で moveDown を実行しても現在位置は変化しません。
- moveRight と moveLeft のメソッドは、いつ実行しても現在位置は変化しません。
- 現在位置は floor という整数型のフィールドに保持することとします。地下 n 階ならその値は −n、地上 n 階ならその値は n です。地下 1 階（floor=−1）にいるときに moveUp を実行しても、floor++ によって、floor が 0 になるわけではなく、floor = 1 となるべきであることに注意してください。

つぎに、Movable 型の一次元配列に Crab と Elevator のインスタンスをそれぞれ 2 つずつ格納し、各インスタンスについて、Movable インタフェースで定義された 4 つのメソッドを適当に実行し、その内容を表示してください。ただし、2 匹のカニは初期位置 x と歩幅 step が異なるものとします。また、2 台のエレベータの初期位置 floor も異なるものとします。

課題 2.4 「オーバーロード」、「オーバーライド」、「ポリモーフィズム」は、いずれも複数の手続きに単一のメソッド名を与えている点で共通していますが、これらの相違点をまとめてください。

課題 2.5 ソースコード 18 と同じ結果を出力するプログラムを、ポリモーフィズムを用いずに（つまり、Movable インタフェースを用いずに）実装し、ポリモーフィズムを用いた実装と比較して、どのような点が困難かを考察してください。

2.5 継承とインタフェースの応用

前節までは、継承とインタフェースについて、ごく簡単な例を通して学びました。ここでは、継承やインタフェースのより複雑な応用例として、二次元平面上を移動しゴミを拾い集める単純なロボットをクラスとして定義し、このロボットを継承してより複雑な動作を行うロボットクラスを定義します。この節の内容は比較的分量が多いため、余裕がある人向けの内容とします。（この内容を消化していることを前提とした課題はありません）

2.5.1 Environment クラスと Garbage クラスの定義

まず、ロボットを定義する前に、ロボットが動き回る環境を定義することにします。環境は二次元平面とし、平面上の位置を表す x 座標、y 座標の値は整数とします。平面上にはいくつかのゴミがランダムな場所に配置されており、「ある位置にゴミがあるかどうかを問い合わせる」「ある位置のゴミを拾う」という 2 種類の動作を、環境に対して行うことができるものとします。ここでは、座標を表現するクラスとして既に作成した Point クラスを利用し、ゴミを表現するクラスとして新たに Garbage クラスを定義することにします。

環境を表す Environment クラスとして、以下のフィールド・メンバを定義します。

- 環境中に存在するゴミ全体を表す配列（Garbage[] garbage）
- ある位置に存在するゴミが配列の何番目に存在するかを返す（ boolean findGarbage(Point p) ）
- 配列の i 番目のゴミを取り除く（ void removeGarbage(int i) ）

Environment クラスの実装は以下のようになります。

ソースコード 19　Environment クラスの定義

```
1  public class Environment {
2          private Garbage[] garbage;
3
4          public Environment(int n){
5                  garbage = new Garbage[n];
6                  for(int i=0;i<n;i++){
7                          garbage[i] = new Garbage(
8                                  (int)(Math.random()*20) - 10,
9                                  (int)(Math.random()*20) - 10
10                                 );
11                 }
12         }
13
14         public Garbage[] getGarbage() {
15                 return garbage;
16         }
17
18         public int findGarbage(Point pt){
19                 for(int i=0;i<garbage.length;i++){
20                         if( garbage[i] != null ){
21                                 Point p = garbage[i].getPosition();
22                                 if( p.sameLocation(pt) )
23                                         return i;
24                         }
25                 }
26                 return -1;
27         }
28
29         public void removeGarbage(int i){
30                 garbage[i] = null;
31         }
32 }
```

ここで、コンストラクタは環境に配置するゴミの数 n を受け取り、(-10,-10)〜(9,9) の範囲に n 個のゴミをランダムに配置しています。double Math.random() は $0 \leq u \leq 1$ の範囲の一様分布に従う乱数を生成して返します。

また、Garbage クラスはそのゴミの位置を保持する Point 型のフィールドを持つ、以下のようなクラスとします。

ソースコード 20　Garbage クラスの定義

```
1  public class Garbage {
2          private Point position;
3
4          public Garbage(int x, int y){
5                  position = new Point(x,y);
6          }
7
8          public Point getPosition() {
9                  return position;
10         }
11 }
```

2.5.2 Robot クラスの定義

まず、ロボット内部の状態として、以下のフィールドを持つものと定義します。

- 現在位置（Point position）
- ロボットの向き（int direction）
- 拾ったゴミの数（int garbageNum）
- ロボットが存在する環境（Environment env）

また、ロボットが実行可能な動作として、以下のメソッドを定義します。

- 一歩進む（ void goForward() ）
- 左を向く（ void turnLeft() ）
- 右を向く（ void turnRight() ）
- 行動する（ void action() ）

ロボットは一歩進むと同時に、新しい位置にゴミが存在するかを調べ、存在する場合は拾うものとします。また、「行動する」という動作は、ロボットができるだけ効率良くゴミを拾うことを念頭に置き、環境や自身の状態を考慮して方向転換や前進などの動作を決定・実行する動作と定義します。これらのメンバに加え、フィールドのゲッターを実装した例は、以下のようになります。

ソースコード 21　Robot クラスの定義

```
1  public class Robot {
2          public static final int UP = 0;
3          public static final int RIGHT = 1;
4          public static final int DOWN = 2;
5          public static final int LEFT = 3;
6          private Point position;
7          private int direction;
8          private int garbageNum;
9          private Environment env;
10
11         public Robot(Environment env){
12                 direction = (int)(Math.random()*4);
13                 position = new Point(0,0);
14                 garbageNum = 0;
15                 this.env = env;
16         }
17
18         public Point nextPosition(){
19                 Point p = position.copy();
20                 switch(direction){
21                 case UP:
22                         p.moveUp();
23                         break;
24                 case DOWN:
25                         p.moveDown();
26                         break;
27                 case LEFT:
28                         p.moveLeft();
29                         break;
30                 case RIGHT:
31                         p.moveRight();
32                         break;
33                 }
34                 return p;
35         }
36
37         public void goForward(){
```

```
38                  position = nextPosition();
39                  int i = env.findGarbage(position);
40                  if( i >= 0 ){
41                          env.removeGarbage(i);
42                          garbageNum++;
43                  }
44
45          }
46
47          public void turnLeft(){
48                  direction--;
49                  if( direction < 0 )
50                          direction = 3;
51          }
52
53          public void turnRight(){
54                  direction++;
55                  if( direction > 3 )
56                          direction = 0;
57          }
58
59          public void action(){
60                  int action = (int)(Math.random()*3);
61                  if( action == 0 ){
62                          turnLeft();
63                  }else if( action == 1){
64                          turnRight();
65                  }else{
66                          // 向きを変えない
67                  }
68                  goForward();
69          }
70
71          public Point getPosition() {
72                  return position;
73          }
74
75          public int getDirection() {
76                  return direction;
77          }
78
79          public void setPosition(Point position) {
80                  this.position = position;
81          }
82
83          public int getGarbageNum() {
84                  return garbageNum;
85          }
86  }
```

2〜5 行目では、4 通りのロボットの向きを表現するため 4 つの定数を宣言し、それぞれに 0 から 3 の整数値を割り当てています。このような定数の宣言は、C 言語では define キーワードを用いて行っていましたが、Java 言語では、フィールドの修飾子に final と static を指定することで、そのフィールドが定数であることを表現します。これらの定数 UP、 DOWN、LEFT、RIGHT はそれぞれ、y 軸の正の方向、y 軸の負の方向、x 軸の負の方向、x 軸の正の方向を意味します。

メソッド getNextPosition は、ロボットが現在の向きに一歩進んだ場合に到着する位置を返します。このメソッドは、ロボットを実際に一歩進める goForward メソッドで利用されます。

メソッド action は、現在向いている方向に対して左、前、右のいずれかの方向を等確率で選択し、前進するという非常に単純な動作となっています。

このように定義したロボット 20 体を 50 個のゴミが散乱した環境の中で動き回らせ、最終的に得られたゴミの数の合計を出力する main メソッドを以下のように定義します。

ソースコード 22　ロボットを動かす main メソッド

```
1  public class Main{
2          public static void main(String[] args){
3                  Environment env = new Environment(50);
4
5                  Robot[] robots = new Robot[20];
6                  for(int i=0;i<robots.length;i++){
7                          robots[i] = new Robot(env);
8                  }
9
10                 for(int time=0;time<20;time++){
11                         for(int i=0;i<robots.length;i++){
12                                 robots[i].action();
13                         }
14                 }
15
16                 int score = 0;
17                 for(int i=0;i<robots.length;i++){
18                         score += robots[i].getGarbageNum();
19                 }
20                 System.out.println("score:" + score);
21         }
22 }
```

5〜8 行目では Robot クラスのインスタンスを 20 個保存するための配列を宣言し、配列の各要素に新たに作成されたインスタンスへの参照を代入しています。10〜14 行目では、全てのロボットを 1 ステップ動作させる処理を 20 回繰り返しています。最後に、16〜20 行目でロボットが拾ったゴミの総数を計算し、ロボット全体の動作の評価値として出力しています。

2.5.3　Robot クラスの継承

次に、Robot クラスを継承し、より効率良くゴミを拾い集めるロボットを定義します。このロボットは、一度訪問したことのある位置を記憶し、同じ位置に二度移動することをなるべく避けることで、より効率良くゴミを探索します。

ここでは、Robot クラスを継承した新しいクラスを MemorableRobot クラスとします。新しいロボットには、一度行ったことのある位置を記録する、Point 型の配列を保持するフィールド visited を追加し、コンストラクタで初期化を行うものとします。

ソースコード 23　MemorableRobot クラス

```
1  public class MemorableRobot extends Robot {
2          private Point[] visited;
3
4          public MemorableRobot(Environment env, int memoSize){
5                  super(env);
6                  visited = new Point[memoSize];
7          }
8  }
```

ここでは、ロボットが記憶しておける位置の数は限られているものとし、その数 memoSize をロボットの生成時に、環境 env と共にコンストラクタに与えます。また、記憶に関する処理を簡単にするため、ロボットは一歩動く度に、ランダムに選ばれた配列要素に現在位置を記録することにします。つまり、ロボットは一つ新しい位置を記憶するたびに、現在覚えている位置のうち一つを忘れます。このようにして、配列に一度行った位置を記憶し、次に移動する位置がこの配列に含まれないように行動の種類を何度か選び直すように改良した action メソッドは、以下のようになります。

ソースコード 24　改良された action メソッド

```
public class MemorableRobot extends Robot {
        private Point[] visited;
        ...

        public void action() {
                // 現在位置を登録
                int memoSize = visited.length;
                visited[(int)(Math.random()*memoSize)] = position;

                // 次の向きの選択を最大で 10回試す
                for(int i=0;i<10;i++){
                        int action = (int)(Math.random()*3);
                        if( action == 0 ){
                                turnLeft();
                        }else if( action == 1){
                                turnRight();
                        }else{
                                // 向きを変えない
                        }
                        Point pt = nextPosition();
                        boolean found = false;
                        for(int j=0;j<memoSize;j++){
                                if( visited[j] != null && visited[j].sameLocation(pt) ){
                                        found = true;
                                        break;
                                }
                        }
                        if( !found ){ // pt は訪れていない可能性が高い
                                break;
                        }
                }
                goForward();
        }
}
```

それでは、20 体のロボットのうち、10 体を memoSize=8 の新しいロボットとするように main メソッドを修正します。

ソースコード 25　新しいロボットを動かす main メソッド

```
1  public class Main{
2          public static void main(String[] args){
3                  ...
4                  Robot[] robots = new Robot[20];
5                  for(int i=0;i<robots.length;i++){
6                          if(i < 10){
7                                  robots[i] = new Robot(env);
8                          }else{
9                                  robots[i] = new MemorableRobot(env,8);
10                         }
11                 }
12                 ...
13         }
14 }
```

何回か実行すると、20 体全部が Robot クラスのインスタンスだった場合に比べ、評価値は平均的に多少大きくなっていることが確認できます。次節では、ロボットや環境中のゴミを統一的に表示するため、インタフェースを利用する例を示します。

2.5.4 マップの表示

ここでは、インタフェースの利用例として、ロボットやゴミが存在する環境のマップを、キャラクタベースで二次元的に画面に表示することを考えます。

キャラクタベースの出力は、基本的に一行ずつ、左から順に出力していく必要があるため、あらかじめどの座標にどのキャラクタを表示するかという「出力結果」を計算しておき、その情報を元に、左上から順にキャラクタの出力を行う必要があります。このような処理は、インタフェースを用いると次のようにして実現することができます。

1. 「1 文字で表示可能である」性質を表現する Symbolizable インタフェースを定義する。このインタフェースは、インスタンスを表現した char 型のデータを返す toSymbol メソッドをメンバとして持つ。
2. Robot クラスと Garbage クラスに、Symbolizable インタフェースを実装する。
3. 出力結果に対応する Symbolizable 型の二次元配列 table を用意する
4. table[x][y] に、y 行 x 列に表示すべき Robot や Garbage のインスタンスを代入しておく
5. table[0][0] から table[1][0], table[2][0],...,table[width-1][0],table[0][1],... と順に、各要素に toSymbol メソッドを呼び出し、得られるキャラクタを表示していく

Symbolizable インタフェースが実装された型は「画面に 1 文字で表示可能なもの」を表し、そのような型のインスタンスを、実際の出力結果に対応する二次元配列に代入することで座標と表示するインスタンスの対応関係を保持します。最後に、各座標の表示を行うためにその座標のインスタンスの toSymbol メソッドを呼び出し、各座標ごとに適切な文字を取得・表示します。

2.5.5 Symbolizable インタフェースの定義と実装

まず、toSymbol メソッドをメンバとする Symbolizable インタフェースを、以下のように定義します。

ソースコード 26　Symbolizable インタフェースの定義

```
1 public interface Symbolizable {
2         char toSymbol();
3 }
```

次に、Robot クラスと Garbage クラスに、Symbolizable インタフェースを実装します。Robot クラスのインスタンスを表す文字は'1'、Garbage クラスのインスタンスを表す文字は'*' とします。Robot クラス、Garbage クラスの toSymbol メソッドはそれぞれ以下のようになります。

ソースコード 27　Robot クラスの toSymbol メソッド

```
        public char toSymbol() {
                return '1';
        }
```

ソースコード 28　Garbage クラスの toSymbol メソッド

```
        public char toSymbol() {
                return '*';
        }
```

これに加え、ロボットやゴミが存在しない空の位置に配置するオブジェクトを表す MapCell クラスを、以下のように定義します。

ソースコード 29　MapCell クラス

```
public class MapCell implements Symbolizable{
        public char toSymbol(){
                return '.';
        }
}
```

2.5.6 二次元マップの表示

最後に、Symbolizable 型の二次元配列にデータを格納し、表示を行う処理を main メソッドに追加します。二次元空間のサイズは決まっていないため、ここではゴミを配置した $(-10, -10)$〜$(9, 9)$ の範囲を表示することにします。

ソースコード 30　二次元マップの表示

```
class Main{
        public static void main(String[] args){
                ...

                Symbolizable symTable[][] = new Symbolizable[20][20];
                for(int time=0;time<20;time++){
                        System.out.println("Step␣" + (time+1));
                        for(int i=0;i<robots.length;i++){
                                robots[i].action();
                        }


                        MapCell emptyCell = new MapCell();
                        for(int y=0;y<20;y++){
                                for(int x=0;x<20;x++){
                                        symTable[x][y] = emptyCell;
                                }
                        }

                        for(int i=0;i<robots.length;i++){
                                Point pt = robots[i].getPosition();
                                int x = pt.getX();
                                int y = pt.getY();
                                if( -10 <= x && x < 10 &&
                                    -10 <= y && y < 10 ){
                                        symTable[x+10][9-y] = robots[i];
                                }
                        }

                        Garbage[] garbage = env.getGarbage();
                        for(int j=0;j<garbage.length;j++){
                                if( garbage[j] != null ){
                                        Point pt = garbage[j].getPosition();
                                        int x = pt.getX();
                                        int y = pt.getY();
                                        symTable[x+10][9-y] = garbage[j];
                                }
                        }

                        for(int y=0;y<20;y++){
                                for(int x=0;x<20;x++){
                                        System.out.print(symTable[x][y].toSymbol());
                                        System.out.print('␣');
                                }
                                System.out.println();
                        }

                }

                ...
        }
}
```

実行結果の一部を以下に示します。

```
...
```

```
Step 20
. * * . * . . . * . . . . . . . . . 1 . .
. . . . . . . * . . . . 1 . . . . . . . .
* . . . . . . . . . . . . . . . . . 1 . .
. . . . * . . . . . . . . . . . . . . . .
* . . . . . . . . * . . . . . . . . . . .
. . . . . . . . . . . . . . . 1 . . . * .
* . . . . . . . . . . . . . . . . . . . .
. . . . . . . 1 . 1 . . . . . . . . 1 . . *
. . . . . . . . . . . . . . . . . . . . 1
1 . . . . . . . . . . . . . . . * * . . . .
. . . . . . . . . . . . . 1 . 1 . . . * * .
1 . . . . . . . . 1 . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . 1 . . .
. . . . . 1 . . . * . . . . . . . . . . .
* . . . 1 . . . . . 1 . . . . . . . . 1 .
. . . . . . . . 1 . . . . . . . . . . . *
. . . . . . . . . * . . . . . . . . . . .
. . . . . * . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . * *
Number of garbages: 24
```

3　3 日目：複合的なデータ構造の利用

3.1　探索木を用いた探索アルゴリズム

この節では、リスト構造のような複合的なデータ表現の利用と、オブジェクト指向に基づく問題の抽象化の題材として、探索問題を解くプログラムを取り上げます。

3.1.1　探索とは

探索問題は、次の 3 つの情報から定義されます。

- 初期状態 s
- 目標状態 g
- オペレータの集合

オペレータとは、状態を受けとりその次の状態を返す関数です。探索とは、初期状態を出発点として、オペレータによる「状態から状態への遷移」を目標状態に辿りつくまで次々と繰り返すことであり、探索問題の解とは、初期状態から目標状態に至るオペレータの系列です。一般的にオペレータは複数存在し、ある一つの状態から遷移可能な状態は複数存在します。ここで、状態をノードとし、遷移元のノード（親ノード）から遷移先のノード（子ノード）へ有向辺が存在すると考えると、探索の過程が木構造のグラフによって表現されることになります。この木構造を探索木といいます。親ノードから遷移先のすべての子ノードを生成することを、その親ノードを**展開する**といいます。

一般的に、探索アルゴリズムは、これから探索すべきノードのリスト（**Open リスト**）と、すでに探索したノードの集合（**Closed リスト**）を持ちます。Open リストからノードを一つ取り出し、そのノードを展開して得られる子ノードを Open リストに入れることを繰り返すことで、探索が行われます。このとき、Open リストから取り出すノードをどのように決定するかで、探索の戦略が決まります。基本的なものとして、ノードを常にリストの先頭から取り出し、展開して得られるノードはリストの末尾に追加することで実現される、**幅優先探索**と呼ばれるものがあります。

3.1.2　簡単な探索問題

本節では、簡単な探索問題を定義し、それを幅優先探索で解くプログラムを作成します。

探索問題は以下のようなものとします。本実験では、すべての状態は整数で表現されるものとします。

- 初期状態は 1
- 目標状態は、実行時に与えられる
- オペレータは、以下の 2 つとする
  - $A(x) = 2 * x$
  - $B(x) = 2 * x + 1$

まず、ノードを表現する Node クラスを定義します。Node クラスは、以下のフィールドを持ちます。

- int state
- Node parent
- String opname
- int depth

state は、そのノードの状態を表します。parent は、そのノードを生成した、遷移元のノードを表します。opname は、そのノードを生成したオペレータの種類を表します。depth は整数値で、そのノードが初期状態から何回の遷移によって得られたものかを表します。

次に、これらのフィールドとコンストラクタ、ゲッターおよびノードの等価性を判定する equals メソッドを実装します。実装例は以下のようになります。（ゲッターの実装は省略します）

ソースコード 31　Node クラスの実装

```
public class Node {
        private int state;
        private Node parent;
        private String opname;
        private int depth;

        public Node(int state, Node parent, String opname, int depth){
                this.state = state;
                this.parent = parent;
                this.opname = opname;
                this.depth = depth;
        }

        public boolean equals(Object obj) {
                Node other = (Node)obj; // obj は Node 型であることをコンパイラに知らせる
                return state == other.state;
        }
}
```

次にこのノードクラスを用いて幅優先探索を行う処理を、クラス NumSearch のメソッド search として定義します。実装例は以下のようになります。

ソースコード 32　NumSearch クラスの実装

```
1  import java.util.LinkedList;
2
3  public class NumSearch {
4          private LinkedList<Node> open, closed;
5          private Node parent;
6
7          public void addNode(Node n){
8                  if(!open.contains(n) && !closed.contains(n))
9                          open.add(n);
10         }
11
12         public void search(int goal){
13                 open = new LinkedList<Node>();
14                 closed = new LinkedList<Node>();
15
16                 // 初期ノードの状態は 1、深さは 0、オペレータの名前は空文字列とする
17                 open.add(new Node( 1, null, "", 0 ));
18                 while( !open.isEmpty() ){
19                         parent = open.getFirst();
20                         open.removeFirst();
21                         closed.add( parent );
22                         if( parent.getState() == goal ){
23                                 // 初期状態から目標状態までのノードの系列を作成
24                                 LinkedList<Node> path = new LinkedList<Node>();
25                                 for(Node n=parent;n!=null;n=n.getParent()){
26                                         path.addFirst(n);
27                                 }
28                                 // 初期状態から順にノードのオペレータと状態を表示
29                                 for(Node n : path){
30                                         System.out.printf("--%s-->␣%s␣", n.getOpname(), n.
                                               getState());
31                                 }
32                                 System.out.println("=␣goal");
33                                 return;
34                         }
35                         int state = parent.getState();
36                         int depth = parent.getDepth();
37                         Node n1 = new Node(state*2, parent, "A", depth+1);
38                         Node n2 = new Node(state*2+1, parent, "B", depth+1);
```

```
39                          addNode(n1);
40                          addNode(n2);
41                  }
42          }
43  }
```

Open リストと Closed リストには、線形リストの実装である LinkedList クラス（java.util パッケージ）を使用しています。これにより、リストの先頭要素の削除および末尾への要素の追加は、常に定数時間で行うことができます。このような「データの集まりを保存するためのクラス」を**コレクションクラス**と呼びます。線形リストの要素のデータ型（クラス名）が一般に E で表されるとき、このリストは LinkedList<E> クラスです。LinkedList<E> クラスの主要なメソッドを表 3 に示します。

メソッドの形式	概要
boolean add(E o)	リストの末尾に、指定された要素 o を追加する
void addFirst(E o)	リストの先頭に、指定された要素を挿入する
void clear()	全ての要素を削除する
boolean contains(Object o)	指定された要素 o がリストに含まれている場合に true を返す
E getFirst()	リストの先頭要素を返す
E getLast()	リストの末尾要素を返す
boolean isEmpty()	リストが空である場合に true を返す
boolean remove(Object o)	リスト内で最初に指定要素 o と一致したものを削除する
E removeFirst()	リストの先頭要素を削除する
E removeLast()	リストの末尾要素を削除する

表 3　LinkedList<E> クラスの主要なメソッド

なお、今回は用いませんが、LinkedList に関連したクラスに ArrayList があります。このクラスは各要素がメモリ上で連続した位置に存在する、ランダムアクセスが定数時間で行えるクラスです。このクラスはこのような性質に関しては通常の配列と同じですが、配列のサイズが自動的に調整されるため、リストへの要素の追加が容易となっています。

search メソッドは、まず Open リストと Closed リストを初期化し、初期ノードを Open リストに追加します（13〜17 行）。次の while ループは、展開すべきノードが存在する（Open リストが空でない）間、ノードの展開を繰り返すためのものです。ループ内の処理は、

1. Open リストの先頭要素を取り出し、Closed リストに追加する（19〜21 行）
2. その要素が目標状態であれば、それに至ったオペレータの系列を出力してメソッドを終了（22〜34 行）
3. そうでない場合は、各オペレータを適用して新たなノードを作成し、Open リストに追加（35〜40 行）

という流れになっています。メソッド addNode は、新たなノードが Open リストと Closed リストのどちらにも含まれていないこと、つまり、既に発見されたノードでないことをチェックし、そのようなノードのみを Open リストに追加するメソッドです。addNode で利用されている、LinkedList クラスの contains メソッドは、引数で指定された要素がリストに含まれている場合に真を返すメソッドです。contains メソッドは、要素のクラスの equals メソッドの定義に基づき「等しい」要素が含まれているかを判定しています。このため、Node クラスの equals メソッドでは「state フィールドが等しい場合にのみ真を返す」ように定義しています。

目標状態が見つかった場合の処理として、最終状態のノードから次々と親を辿っていき、見つかったノードをリストの先頭に次々と追加しています。これにより、最終的に初期状態を先頭要素、最終状態を末尾要素とするノードのリストが作成されます。続いてこのリストを出力するループでは、「**拡張** *for* **文**」を用いています。この記法は一般的には「*for-each* **構文**」などと呼ばれ、配列やリストといったデータの集まりに対して、それらの要素一つ一つに対して何か処理を行う場合に用いられます。拡張 for 文は以下のような構文で利用します。

ソースコード 33　拡張 for 文の構文

```
for(型名 変数名 : データの集まり){
        ...
}
```

この構文の「データの集まり」の部分には、LinkedList などのコレクションクラスのオブジェクトのほか、通常の配列も用いることができます。また、この左に書かれる変数の型は、「データの集まり」の要素の型である必要があります。search メソッドの例では、データの集まりが LinkedList<Node> であるため、この集合に含まれる要素の型は Node となります。この例では、ノードのリスト path に含まれる全てのノード n に対して、その並びの順に処理を繰り返すことになります。このように拡張 for 文を用いることで、添字を用いることなく簡潔にループが記述でき、さらに、添字の値が有効な範囲を逸脱してしまうことによるバグも起こりにくくなります。

また、目標状態が見つかったときのオペレータの系列の出力には、書式付き出力を行う printf メソッドを用いています。printf メソッドは、C 言語における printf 関数とほぼ同じ使い方ができます。例えば、int や long などの整数型は %d によって、float や double などの浮動小数点型は %f によって、書式を指定することができます。

この NumSearch クラスを利用した main メソッドの例を以下に示します。

ソースコード 34　NumSearch クラスを用いた探索

```
1  import java.util.Scanner;
2
3  public class Main {
4          public static void main(String args[]) {
5                  Scanner sc = new Scanner(System.in);
6                  System.out.print("Goal=?␣");
7                  int goal = sc.nextInt();
8
9                  NumSearch ns = new NumSearch();
10                 ns.search(goal);
11         }
12
13 }
```

ここで使用している Scanner クラスは、文字列を空白等の区切り文字で区切られた固まり（トークン）に分割したり、基本データ型の値に変換する機能を持ちます。システムの標準入力を表す、System クラスの in フィールドと組み合わせることで、C 言語における scanf 関数と同様の機能を実現することができます。ソースコードの 5 及び 7 行目が、標準入力 System.in と結合した Scanner オブジェクト sc の nextInt メソッドを使って、int 型のデータを標準入力から読み取る処理を表しています。目標状態を 100 に設定し、実行した結果を以下に示します。

```
Goal=? 100
----> 1 --B--> 3 --A--> 6 --A--> 12 --B--> 25 --A--> 50 --A--> 100 = goal
```

3.2　問題の抽象化とフレームワークの作成

前節で作成した NumSearch クラスは、解こうとしている特定の問題に固有の記述と、広い範囲の探索問題を解く場合にも現れるプログラムの構造が混在しており、これを再利用して別の問題を解くプログラムを作成する場合に、使いやすいソースコードになっているとは言えません。本節では、探索問題に共通する本質的な処理を抽出し、クラスとして独立させる形で問題を抽象化します。抽象化されたクラスでは、具体的な問題固有の部分を暫定的なメソッドとして記述し、具体的な問題が与えられる前に、問題を解くアルゴリズムを記述しておきます。個々の具体的な問題を解くには、このクラスを継承したサブクラスを定義し、この暫定的なメソッドをオーバーライドして問題固有の処理を記述するだけでよいことになります。このように、抽象化されたクラスは、問題の見通しを良くするだけでなく、様々な探索問題を解くための基盤となる枠組としての役割も果たします。このようなクラスをオブジェクト指向プログラミングでは「フレームワーク」と呼んでいます。その例として、Java のプログラムをホームページに埋め込んで動作させる「アプレット」と呼ばれるものが良く知られています。個々のアプレットは Applet クラスとして提供されているフレームワークを継承したサブクラスとして定義されます。

探索問題は、初期状態、目標状態、オペレータの集合の 3 つによって定義されました。以下では、これらの要素を NumSearch クラスから取り出し、メソッドによる抽象化を行った新たな Search クラスを定義します。

3.2.1 初期状態と目標状態の抽象化

NumSearch クラスにおいては初期状態は 1 に固定され、目標状態はただ 1 つだけが許されていました。これに対し、これから定義する Search クラスではこれらの情報をオーバーライド可能なメソッドとして定義することにします。初期状態を返すメソッドを Node initialNode() とします。また、目標状態は問題によっては一つとは限らず、かつ明示的に列挙して与えることが困難な場合もあるため、状態が目標状態かどうかを判定するメソッド boolean isGoal(Node) によって、目標状態を定義するものとします。

まず、NumSearch クラスと全く同じ内容で Search クラスを作成し、search メソッドにおいて、初期状態の決定や目標状態の判定処理を initialNode、isGoal メソッドを用いるよう変更します。最後に、新しく導入したメソッドの暫定的な定義内容をメンバに追加すると、以下のようになります。

ソースコード 35　初期状態と目標状態の抽象化

```
import java.util.LinkedList;

public class Search {
        private LinkedList<Node> open, closed;
        private Node parent;

        public int initialState(){ // サブクラスでオーバーライド
                return 1;
        }

        public Node initialNode(){
                return new Node(initialState(), null, "", 0);
        }

        public boolean isGoal(int state){ // サブクラスでオーバーライド
                return state >= 10;
        }

        public boolean isGoal(Node n){
                return isGoal(n.getState());
        }

        ...

        public void search(){ // 目標状態を受け取らないよう変更
                ...
                open.add(initialNode());
                while( !open.isEmpty() ){
                        parent = open.getFirst();
                        ...
                        if( isGoal(parent) ){ // isGoal を用いるよう変更
                                ...
                        }
                        int state = parent.getState();
                        ...
                }
        }
}
```

ここで、isGoal メソッドは直接 int 型の状態を受け取る「状態レベル」のものをまず定義し、そのメソッドを用いて、ノードの状態を判定する「ノードレベル」のものを定義しています。これにより、このフレームワークをスーパークラスとして用いて具体的な判定処理を記述する際は、引数として int 型の状態を受け取るメソッドのみをオーバーライドすればよいことになります。

3.2.2 オペレータの集合の抽象化

次に、オペレータの集合によって、あるノードから遷移可能な別のノード集合を求める処理を抽象化します。ある状態から遷移可能な複数の状態を列挙する抽象メソッドとして、void expand(int s) を定義します。メソッド expand は、引数として与えられた状態 s から遷移可能な任意の状態 s' について、その状態へ遷移するために用いたオペレータ op とともに、新しい状態からなるノードを作成するメソッド addNode(s',op) を呼び出すものとします。addNode は、引数として与えられた状態 s' とオペレータ op に加え、適切な親ノードと深さが設定された新しいノードを Open リストに追加します。expand メソッドの暫定的な実装および addNode メソッドの実装例は以下のようになります。

ソースコード 36　オペレータの集合の抽象化

```
import java.util.LinkedList;

public class Search{
        private LinkedList<Node> open, closed;
        private Node parent;

        ...

        public void expand(int state){ // サブクラスでオーバーライド
                addNode(state+1, "addOne");
        }

        public void expand(Node n){
                expand(n.getState());
        }

        public void addNode(int state, String opname){
                addNode(new Node(state, parent, opname, parent.getDepth()+1));
        }

        public void addNode(Node n){
                ... // NumSearch で定義したものを利用
        }

        public void search(){
                ...
                while( !open.isEmpty() ){
                        ...
                        if( isGoal(parent) ){
                                ...
                        }
                        expand(parent);
                }
        }
}
```

ここで、search メソッドを簡潔にするため、expand メソッドも isGoal メソッドなどと同様に、int 型の状態を受けとる「状態レベル」のメソッドを用いて「ノードレベル」のメソッドを定義しています。

3.2.3 探索終了時の処理

現在の search メソッドには、解が見つかった時の処理が埋め込まれていますが、そのような処理も、探索の本体からは切り離されるべきです。このため、search メソッドは探索が成功した場合には単にゴールノードを返すのみとし、そうでない場合は null を返すメソッドとします。また、ゴールノードから解に至る系列を生成するメソッド makePath、その系列を出力する printPath メソッドを探索内部から外に出し、これらはユーザーが必要に応じて利用するものとします。以下に実装例を示します。

ソースコード 37　探索終了時の処理の外部化

```
import java.util.LinkedList;

public class Search{
	...

	public Node search(){ // Node を返すように変更
		...
		while( !open.isEmpty() ){
			...
			if( isGoal(parent) ){
				return parent;
			}
			...
		}
		return null;
	}

	public LinkedList<Node> makePath(Node goal){
		LinkedList<Node> path = new LinkedList<Node>();
		for(Node n=goal; n!=null; n=n.getParent()){
			path.addFirst(n);
		}
		return path;
	}

	public void printPath(LinkedList<Node> path){
		for(Node n : path){
			System.out.printf("--%s-->␣%d␣", n.getOpname(), n.getState());
		}
		System.out.println("=␣goal");
	}
}
```

これで抽象化が完了しました。Search クラスの全体のコードを以下に示します。

ソースコード 38　Search クラス

```
import java.util.LinkedList;

public class Search{
	private LinkedList<Node> open, closed;
	private Node parent;

	public int initialState(){
		return 1;
	}

	public Node initialNode(){
		return new Node(initialState(), null, "", 0);
	}

	public boolean isGoal(int state){
		return state >= 10;
	}

	public boolean isGoal(Node n){
		return isGoal(n.getState());
	}

	public void expand(int state){
		addNode(state+1, "addOne");
	}

	public void expand(Node n){
```

```
                expand(n.getState());
        }

        public void addNode(int state, String opname){
                addNode(new Node(state, parent, opname, parent.getDepth()+1));
        }

        public void addNode(Node n){
                if(!open.contains(n) && !closed.contains(n))
                        open.add(n);
        }

        public Node search(){
                open = new LinkedList<Node>();
                closed = new LinkedList<Node>();

                open.add(initialNode());
                while( !open.isEmpty() ){
                        parent = open.getFirst();
                        open.removeFirst();
                        closed.add(parent);
                        if( isGoal(parent) ){
                                return parent;
                        }
                        expand(parent);
                }
                return null;
        }

        public LinkedList<Node> makePath(Node goal){
                LinkedList<Node> path = new LinkedList<Node>();
                for(Node n=goal; n!=null; n=n.getParent()){
                        path.addFirst(n);
                }
                return path;
        }

        public void printPath(LinkedList<Node> path){
                for(Node n : path){
                        System.out.printf("--%s-->␣%d␣", n.getOpname(), n.getState());
                }
                System.out.println("=␣goal");
        }
}
```

Search クラスは探索のためのフレームワークを提供しています。このフレームワークを用いて具体的な問題を解くには、Search クラスを継承し、抽象化された 3 つの概念「初期状態」「目標状態」「オペレータの集合」に対応する 3 つのメソッド

- int initialState()
- boolean isGoal(Node n)
- void expand(int state)

の具体的な定義を与えて、暫定版の定義をオーバーライドするだけでよいことになります。このようにサブクラスで具体的に定義されるべきメソッドを、そのフレームワークの**ホットスポット**（hot spot）ということがあります。

3.3 作成したフレームワークの利用

最後に、作成した Search クラスをフレームワークとして利用し、具体的な問題を解くクラスを定義します。

3.3.1 NumSearch クラスの再定義

まず、最初に定義した、2 つのオペレータからなる単純な問題を解く NumSearch クラスを再定義します。前節の最後で述べた 3 つのメソッドに加え、isGoal メソッドで目標状態と判定する状態を保持するための int 型のフィールド goalState と、このフィールドを初期化するコンストラクタを定義します。

ソースコード 39　Search クラスを利用した NumSearch クラス

```
public class NumSearch extends Search {
        private int goalState;

        public int initialState(){ // Search をオーバーライド
                return 1;
        }

        public boolean isGoal(int state) { // Search をオーバーライド
                return state == goalState;
        }

        public void expand(int state) { // Search をオーバーライド
                addNode(state*2, "A");
                addNode(state*2+1, "B");

        }

        public NumSearch(int goalState) { // コンストラクタ
                this.goalState = goalState;
        }
}
```

このように、NumSearch クラスはフレームワークを用いることで簡潔に定義することができます。探索問題の詳細として、新たに NumSearch クラスで定義している点は

- ある状態に対して 2 種類のオペレータが適用できる
- 目標状態は特定の一つの状態である

という 2 点であることが、ソースコードから容易に理解できます。この新しい NumSearch クラスを利用して探索を行い、結果を表示する main メソッドは以下のようになります。

ソースコード 40　新しい NumSearch クラスの利用

```
import java.util.Scanner;

public class Main {
        public static void main(String args[]) {
                Scanner sc = new Scanner(System.in);
                System.out.println("Goal=?␣");
                int goalState = sc.nextInt();

                NumSearch ns = new NumSearch(goalState);
                Node goal = ns.search();
                if( goal != null ){
                        ns.printPath(ns.makePath(goal));
                }else{
                        System.out.println("fail");
                }
        }
}
```

3.3.2 より複雑な探索問題を解くクラスの作成

次に、フレームワークを利用してもう少し複雑な探索問題を解いてみます。ここで解く問題は、「宣教師と人食い人種」と呼ばれるパズルです。問題の定義を以下に示します。

- 3 人の宣教師 (missionary) と 3 人の人食い人種 (cannibal) が、2 人乗りの 1 そうの舟で南北に流れる川を西岸から東岸に渡ることを考えます。
- 初期状態は、西岸に 3 人の宣教師と 3 人の人食い人種と舟が存在する状態です。
- 宣教師も人食い人種も舟を漕ぐことができます。
- 舟を利用して、最大で 2 人同時に反対側の岸に移動して、岸に降ります。ただし、このとき川のそれぞれの岸で宣教師の数がそこにいる人食い人種の数より少なくなると、その宣教師は殺されてしまいます。
- 目標状態は、宣教師が一人も殺されることなく、東岸に 3 人の宣教師と 3 人の人食い人種が存在する状態です。

宣教師および人食い人種の人数は決まっているため、この問題における状態は、

- 西岸に存在する宣教師の人数 m
- 西岸に存在する人食い人種の人数 c
- 舟の位置 b（西なら 1、東なら 0）

の 3 つのパラメータで決定されます。この 3 つのパラメータをさらに一つの整数値 ”mcb”（m*100 + c*10 + b）に符号化したものを状態として取り扱います。

この問題を解くクラス AcrossRiverSearch の定義の一部を以下に示します。

ソースコード 41　AcrossRiverSearch クラスの一部

```
1  public class AcrossRiverSearch extends Search {
2          public static final int WEST = 1;
3          public static final int EAST = 0;
4
5          public int initialState() { // Search をオーバーライド
6                  return 331;
7          }
8
9          public boolean isGoal(int state) { // Search をオーバーライド
10                 return state == 0;
11         }
12
13         public int getMissionary(int state) {
14                 return state/100;
15         }
16
17         public int getCannibal(int state) {
18                 return (state/10)%10;
19         }
20
21         public int getBoatPosition(int state) {
22                 return state%10;
23         }
24
25         public int encodeState(int m, int c, int b) {
26                 return m*100 + c*10 + b;
27         }
28
29         public void expand(int state) { // Search をオーバーライド
30                 int m = getMissionary(state);
31                 int c = getCannibal(state);
32                 int b = getBoatPosition(state);
33                 int dir; // 西から東に移動するならば 1、逆向きならば -1
34
```

```
35                  if(b == WEST) {dir = 1;} else {dir = -1;}
36
37                  check_and_add(m-2*dir, c, 1-b, "MM"); // 宣教師 2人を移動させる
38                  check_and_add( ); // 宣教師 1人、人食い人種 1人を移動させる
39                  check_and_add( ); // 人食い人種 2人を移動させる
40                  check_and_add( ); // 宣教師 1人を移動させる
41                  check_and_add( ); // 人食い人種 1人を移動させる
42          }
43
44          public void check_and_add(int newM, int newC, int newB, String opname) {
45                  if(newM >= 0 && newM<=3 && newC >= 0 && newC<=3
46                     && (newM == 0 || newM >= newC) // 西岸のチェック
47                     && (3-newM == 0 || 3-newM >= 3-newC) ) { // 東岸のチェック
48                              addNode(encodeState(newM, newC, newB), opname);
49                  }
50          }
51
52          public static void main(String[] args) {
53                  AcrossRiverSearch ar = new AcrossRiverSearch();
54                  Node goal = ar.search();
55                  if(goal != null) {
56                          ar.printPath(ar.makePath(goal));
57                  } else {
58                          System.out.println("fail");
59                  }
60          }
61  }
```

メソッド initialState では初期状態として 331（西岸には宣教師と人食い人種が 3 人ずつ及び舟が存在する）が設定されており、同様にメソッド isGoal では目標状態として 0（西岸は宣教師と人食い人種が共に 0 人で、舟が東岸に存在する）というただ一つの状態が設定されています。

メソッド getMissionary, getCannibal, getBoatPosition は状態を表す整数を復号化し、各パラメータを得るためのものです。また、メソッド encodeState は逆に、状態を決定付ける 3 種類のパラメータを一つの整数値に符号化します。

メソッド expand は、未完成なものとなっています。expand は、補助的なメソッド check_and_add を用いて、考えられる 5 種類のオペレータを適用しますが、5 つのうち 4 つがまだ実装されていません。メソッド check_and_add は、オペレータ適用後の西岸の状態および適用したオペレータの名前を受け取り、その状態が制約を満たしているならば、新しいノードを作成し、Open リストに追加します。メソッド check_and_add の 4 つの引数の意味を、順に以下に示します。

- 次の状態における西岸の宣教師の人数
- 次の状態における西岸の人食い人種の人数
- 次の状態における舟の位置
- 適用したオペレータの名前

オペレータの名前は、宣教師と人食い人種をそれぞれ M,C で表し、それらを移動人数と同じ数だけ並べたものとします。例えば、”MM”は宣教師 2 人を、”MC”は宣教師 1 人と人食い人種 1 人を、”C”は人食い人種 1 人を移動させるオペレータを表します。

このクラスは、フレームワークを用いて具体的な問題を解くクラスであり、再利用を目的としたクラスではありません。このような場合は、このクラスに「Main クラス」の役割をさせてしまってよいと考えられるため、AcrossRiverSearch クラスは、パズルの解を求める main メソッドも含んでいます。expand メソッドが完成した場合の実行結果は以下のようになります。

```
----> 331 --MC--> 220 --M--> 321 --CC--> 300 --C--> 311 --MM--> 110 --MC--> 221 --MM-->
    20 --C--> 31 --CC--> 10 --M--> 111 --MC--> 0 = goal
```

3.4 3 日目の課題

課題 3.1 Search クラスを継承して、つぎの仕様を満たす NumSearch2 クラスを作成してください。

- 初期状態を保持する整数フィールド initialState を導入する。
- 目標状態を定義するために、2 つの整数フィールド min と max を導入し、目標状態は min 以上かつ max 以下の整数とする。
- initialState、min、max の値は、コンストラクタで任意の値に初期化できるものとする。
- オペレータとして次の 3 つがある。
  - M1(x) = x − 1
  - TR(x) = 3 * x
  - SQ(x) = x * x

main メソッドでは、キーボードから initialState、min、max の値を読み取り、NumSearch2 のインスタンスをコンストラクタで初期化し、search メソッドを呼び出してこの探索問題を解いてください。

課題 3.2 AcrossRiverSearch クラスの expand メソッドにおける、実装されていない 4 種類のオペレータを実装し、expand メソッドを完成させてください。

課題 3.3 本実験では、状態空間を整数の集合として実験を行ってきましたが、整数の代わりに、任意のデータ構造を扱えるようにするにはどうしたらよいか、考察してください。

課題の提出に関して

提出締切 4 月 21 日 月曜日 午後 1 時まで

提出先 情報科学研究科棟 8-02 室前 レポートボックス

連絡先 表現系工学研究室 山内（yamauchi@complex.eng.hokudai.ac.jp）