応用Javaプログラミング

●授業内容と成績評価方法：シラバスを参照.

注：履修者の理解度で 内容を変更する可能性があり，
予定した内容を教えきれないかもしれない.

●難易度：超難しい．なぜなら，オブジェクト指向だけでさえ
難しいのに，それを土台としたデザインパターンは当然
もっと難しいから.
その分，評価も甘くする予定.

●重要度：極めて重要（実務レベルのプログラミングを目指す）.

参考書：
増補改訂版Java言語で学ぶデザインパターン入門
増補改訂版Java言語で学ぶデザインパターン入門「マルチスレッド編」
両方とも 結城浩著

応用Javaプログラミング
第1回

-- 抽象化 --
-- 多様性 --

継承と合わせて，オブジェクト指向の3大要素と呼ばれる

-- Generic Sorting --
-- Strategyデザインパターン --
-- 無名クラス --

抽象化はオブジェクト指向において最も難しいところ．

オブジェクト指向によるアプリケーション開発とは

変更されない箇所を軸に、頻繁に変更されるであろう箇所を
クラスやインターフェースに抽出するプログラミングスタイルである.

結果として，たくさんのクラスやインターフェースが作成される.
それらの管理が面倒なので，**統合開発環境**（Eclipse, IntlliJなど）
を利用するのは必須.

Integrated Development Environment
(IDE と略す）

https://www.kkaneko.jp/tools/win/eclipse.html

例：整列問題.

問：変更されないであろう箇所は何か？
答：処理順序：データの準備，データの整列，結果の表示.

オブジェクト指向によるアプリケーション開発とは

変更されない箇所を軸に、頻繁に変更されるであろう箇所を
クラスやインターフェースに抽出するプログラミングスタイルである.

例: 整列問題.

問: 頻繁に変更されるであろう箇所は何か？

答: ① 整列の対象データ → Integer, Double, String など

② データの記憶方法（i.e., データ構造） → 配列, LinkedList, など

③ データを並べ替えるための基準 → Comparator, など

（たとえば，学生データを成績の良い順に整列したり，学籍番号の小さい順に整列したりする）

Generic Sorting in Java

Javaに，汎用整列アルゴリズムが既に用意されている．

具体的には，JavaにArraysというクラスがあり，Arraysのクラスメソッドとして sort(T[ ], Comparator<T>) というものが用意されていて，それがT型要素の配列を指定コンパレータで整列．

```
//データの大小の比較を抽象化するインターフェース
public  interface Comparator<T> { //Tは型パラメータ.
    //戻り値：dataがanotherDataより小さければ  負の整数;
    //          dataがanotherDataと同じならば  ゼロ;
    //          dataがanotherDataより大きければ  正の整数.
    int compare(T data, T anotherData) ;
}
```

インタフェース：C言語の関数のヘッダーファイルみたいなもの．
メソッドの使い方（および 機能）を定めるだけ，その中身がなし．

クラス vs. インターフェース

●インターフェースを*抽象メソッドと定数しか持たない抽象クラス*として見なせるので，変数の型として使える．その変数に「そのインターフェースを実装した非抽象クラスのインスタンス」を代入できる．

●インターフェースの利点：

① Javaのどのクラスも1つの親クラスしか継承できないが，複数のインターフェースを実装できるので，（抽象）クラスよりも柔軟性がある．

② インターフェースが（メソッドの）機能しか定めなく，その機能の実現を（そのインターフェースを実装する）クラスに任せる．
実装側のクラスを自由に変えられるので，インターフェースが抽象化に適している．

③ 抽象クラスは（*子クラスによる*）*継承*を念頭に作られるものなので，“is-a”関係を表すのに対して，インターフェースが機能を定めるので，”can-do”関係を表す．

Comparator<T> インターフェースの説明

```
//データの大小の比較を抽象化するインターフェース
public  interface Comparator<T> { //Tは型パラメータ.
    //戻り値:dataがanotherDataより小さければ  負の整数;
    //            dataがanotherDataと同じならば  ゼロ;
    //            dataがanotherDataより大きければ  正の整数.
    int compare(T data, T anotherData) ;
}
```

Tは型パラメータであり，整列したいデータの型が決まった時に，
　Tをその型で置き換える.
たとえば，整列したいデータの型がStringであれば，
　　Comparator<String> を使う.

負（正）の整数のとき，Arraysのクラスメソッドsortで整列した結果の配列において，dataがanotherDataより配列の前（後）にある.

型パラメータTとインタフェース Comparator<T>を使う利点は何か？

答：T型の配列を整列するメソッドsort(T[ ], Comparator<T>)が書ける．
そのメソッドは汎用的である（すなわち，大小比較が可能なデータの配列が何であれ，整列できる）．

さて，実際にsort(T[ ], Comparator<T>)の自作バージョンを書いてみよう！

汎用(挿入)ソート用の自作クラス・メソッドmySort

T がこのクラスメソッドで使われる型パラメータである ことを示す.

```
public static <T> void mySort(T[ ] array, Comparator<T> cmp) {
    int size = array.length ; //配列のサイズを求めておく
    for (int i = 1 ; i < size ; i++) {
        T temp = array[i] ; //未整列部分の先頭要素を覚えておく
        int j = i - 1 ; //整列済み部分の末尾要素の居場所
        for (; j >= 0 ; j--) { //未整列部分を末尾から先頭まで見ていく
            if ( cmp.compare(array[j], temp) > 0 ) //tempより後ろにあるべきなら
                array[j + 1] = array[j] ; //後ろに1コマずらす
            else //さもなければ，tempのいるべき場所が見つかったので
                break; //内側のfor文から出る
        }
        array[j + 1] = temp ; //tempをそのいるべき場所に置く
    }
}
```

文字列を整列する問題

整数nおよびn個の文字列をキーボードから入力して，
アルファベット順とは逆の順に並べ替えてから画面に
表示するプログラムを書いてみよう．

まず，コンパレータを用意する．

```
package sorting ;
import  java.util.Comparator ;
public class ReverseStringComparator
          implements Comparator<String> {
    public int compare(String data, String anotherData) {
        return anotherData.compareTo(data) ;
    }// compareメソッドの終わり
}// ReverseStringComparatorクラスの終わり
```

文字列を整列する問題（続）

```
package  sorting ;

import  java.util.Scanner ;
import  java.util.Arrays ;

public class StringSorting {
     public static void main(String arg[ ]){
          Scanner sc = new  Scanner(System.in);

          int n =  sc.nextInt( ) ; //nを入力する
          String array[ ] =  new String[n] ; //配列のメモリを確保
          for (int i = 0 ; i < n ; i++)  // n個の文字列を入力する
               array[i] = sc.next( ) ;
         mySort( array, new ReverseStringComparator( ) ) ;
          for (int i = 0 ; i < n ; i++) // 整列結果の配列を表示する
               System.out.println( array[i] ) ;
     } // mainメソッドの終わり
     //mySortメソッドをここに置く
} // StringSortingクラスの終わり
```

汎用整列メソッドで整列

学生データの整列問題

整数nおよびn人の学生のデータをキーボードから入力して，
成績のよい順に並べ替えてから画面に表示して，さらにIDの昇順に
並べ替えてから画面に表示するプログラムを書いてみよう．

学生データを表すクラス

```
package  sorting ;

public class StudentData {
    private int id, score ;
    private String name, address ;

    // 引数なしコンストラクタ
    public StudentData( ) {
    } //引数なしコンストラクタの終わり

    // 引数ありコンストラクタ
    public StudentData(int id, String name, String address, int score) {
        this.id = id;
        this.name = name;
        this.address = address;
        this.score = score;
    } //引数ありコンストラクタの終わり
```

各学生の持つデータ

次頁に続く

学生データを表すクラス（続）

```
public int getID( ) { // ゲッターメソッド
    return id ;
}
public int getScore( ) { // ゲッターメソッド
    return score ;
}
public String getName( ) { // ゲッターメソッド
    return name ;
}
public String getAddress( ) { // ゲッターメソッド
    return address ;
}
public void setID(int id) { // セッターメソッド
    //本当はidの正当性を調べて，正当のときのみ下記の代入を行う
    this.id = id ;
}
```

この注意はすべてのsetterメソッドに共通.

次頁に続く

学生データを表すクラス（続）

```
public void setScore(int score) {// セッターメソッド
    if ( score < 0 || score > 100)
        throw new IllegalArgumentException( ) ;
    this.score = score ;
}
public void setName(String name) {// セッターメソッド
    this.name = name ;
}
public void setAddress(String address) {// セッターメソッド
    this.address = address ;
}
```

次頁に続く

学生データを表すクラス（続）

```
public void inputStudentData( ) { // 学生データを入力するメソッド
    Scanner  sc = new Scanner(System.in);

    System.out.printf("id: ") ;
    id = Integer.parseInt( sc.nextLine( ) ) ;
    System.out.printf("name: ") ;
    name = sc.nextLine( ) ;
    System.out.printf("address: ") ;
    address = sc.nextLine( ) ;
    System.out.printf("score: ") ;
    score = Integer.parseInt( sc.nextLine( ) ) ;
} // 学生データを入力するメソッドの終わり
```

次頁に続く

学生データを表すクラス（続）

```
    public void printStudentData( ) { // 学生データを表示するメソッド
        System.out.println("id: " + id) ;
        System.out.println("name: " + name) ;
        System.out.println("address: " + address) ;
        System.out.println("score: " + score) ;
    } // 学生データを入力するメソッドの終わり
} // StudentDataクラスの終わり
```

学生データを整列するときに使う大小比較クラス

成績のよい順に整列したいとき：

```
package sorting ;
import  java.util.Comparator ;
public class StudentScoreComparator
          implements Comparator<StudentData> {
    public int compare(StudentData data, StudentData anotherData) {
        return anotherData.getScore( ) – data.getScore( ) ;
    } // compareメソッドの終わり
} // StudentScoreComparatorクラスの終わり
```

ID（整数）の昇順に整列したいとき：

```
package sorting ;
public class StudentIDComparator
          implements Comparator<StudentData> {
    public int compare(StudentData data, StudentData anotherData) {
        return data.getID( ) – anotherData.getID( ) ;
    } // compareメソッドの終わり
} // StudentIDComparatorクラスの終わり
```

学生データを整列する問題

```
package  sorting ;

import  java.util.Scanner ;

public class StudentSorting2a {
    public static void main(String arg[ ]){
        Scanner  sc = new  Scanner(System.in);

        int n =  sc.nextInt( ) ;//nを入力する
        StudentData array[ ]=new StudentData[n] ;//配列のメモリを確保
        for (int i = 0 ; i < n ; i++)  // n個の文字列データを格納するメモリを確保
            array[i] = new StudentData( ) ;
        for (int i = 0 ; i < n ; i++)  // n個の文字列データを入力する
            array[i].inputStudentData( ) ;
```

学生データを整列する問題（続）

```
        //成績のよい順に整列してから表示
        StudentScoreComparator scoreComparator =
            new StudentScoreComparator( ) ;
        Arrays.sort(array, scoreComparator) ;
        for (int i = 0 ; i < n ; i++) // 整列結果の配列を表示する
            array[i].printStudentData( ) ;

        //IDの昇順に整列してから表示
        StudentIDComparator idComparator =
            new StudentIDComparator( ) ;
        Arrays.sort(array, idComparator) ;
        for (int i = 0 ; i < n ; i++) // 整列結果の配列を表示する
            array[i].printStudentData( ) ;
    } // mainメソッドの終わり
} //StudentSorting2aクラスの終わり
```

StringSorting.mySort に変更してもよい.

mySortは文字列の整列と関係のない汎用ソートメソッドなので，本当はStringSortingクラスに置くべきではない.

前頁の見直し

```
        //成績のよい順に整列してから表示
        StudentScoreComparator scoreComparator =
            new StudentScoreComparator( ) ;
        Arrays.sort(array, scoreComparator) ;
        for (int i = 0 ; i < n ; i++) // 整列結果の配列を表示する
            array[i].printStudentData( ) ;

        //IDの昇順に整列してから表示
        StudentIDComparator idComparator =
            new StudentIDComparator( ) ;
        Arrays.sort(array, idComparator) ;
        for (int i = 0 ; i < n ; i++) // 整列結果の配列を表示する
            array[i].printStudentData( ) ;
    } // mainメソッドの終わり
} //StudentSorting2aクラスの終わり
```

結果は StudentSorting2b クラスにある

StudentScoreComparatorクラスのインスタンスとしてscoreComparatorの1つしか使われていないのでStudentScoreComparatorクラスの代わりに無名クラスを使ったほうが良い．StudentIDComparatorクラスもそうだ．

Strategyデザインパターン

考え方: アルゴリズムの部分を，その操作対象と意識的に分離して，アルゴリズムとのインターフェース（API）の部分だけを規定する．その結果，**アルゴリズムを容易に切り替えることができる**．

Strategy役（戦略役）：戦略を利用するためのインターフェース（API）を定める役．学生データ整列の例では，Comparator<T>インターフェースがこの役をつとめている．

ConcreteStrategy役（具体的戦略役）：Strategy役のインターフェースを実際に実装する役．ここで具体的な戦略（作戦・方策・方法・アルゴリズム）を実際にプログラミングする．学生データ整列の例では，StudentScoreComparatorクラスやStudentIDComparatorクラスがこの役をつとめている．

Context役（文脈役）：Strategy役を利用する役．ConcreteStrategy役のインスタンスを持っていて，必要に応じてそれを利用する．学生データの例では，StudentSorting2aクラスがこの役を務めている．

抽象化

例：整列問題.

問：頻繁に変更されるであろう箇所は何か？

答：① 整列の対象データ

② データの記憶方法（i.e., データ構造）

③ データを並べ替えるための基準
（たとえば，学生データを成績の良い順に整列したり，学籍番号の小さい順に整列したりする）

Integer, Double, String など

配列, LinkedList, など

Comparator, など

この「頻繁に変更されるであろう箇所を抜き出す」という作業を**抽象化**という.

抽象化のポイント

① 外から見てそのものが複雑でない状態を作る.
② 無駄を省き，洗練させてわかりやすいものを作る.
結果として，汎用的なプログラムが書ける.
③ 抽象化を行うとき，「クラスの役割は一つにする」ということを
意識しておくべき.
これは「Single Responsibility Principle」という.
④ newによるインスタンスの生成は具象化で，抽象化の反対なので
できるだけ遅らせる.（1つの方法：factoryメソッドを利用）
⑤ 抽象化に強く関連する重要な仕上げとして，クラスや
インターフェースに正しい名前にを付けること.

抽象化はオブジェクト指向において最も難しいところ.

なるべく様々な使い方ができるように，本質的な，抽象的なコードを書くべき.
抽象に対して作用するプログラムが出来上がることで多様性が生まれる.

StudentSorting2a.javaの見直し

ポイント：多様性（ポリモーフィズム）.

```
package  sorting ;

import  java.util.Scanner ;

public class StudentSorting2aa {
    public static void main(String arg[ ]){
        Scanner  sc = new  Scanner(System.in);

        int n =  sc.nextInt( ) ;//nを入力する
        StudentData array[ ]=new StudentData[n] ;//配列のメモリを確保
        for (int i = 0 ; i < n ; i++)  // n個の文字列データを格納するメモリを確保
            array[i] = new StudentData( ) ;
        for (int i = 0 ; i < n ; i++)  // n個の文字列データを入力する
            array[i].inputStudentData( ) ;
```

学生データを整列する問題（続）

```
        //成績のよい順に整列してから表示
        Comparator<StudentData> comparator =
            new StudentScoreComparator( ) ;
        Arrays.sort(array, comparator) ;
        for (int i = 0 ; i < n ; i++) // 整列結果の配列を表示する
            array[i].printStudentData( ) ;

        //IDの昇順に整列してから表示
        comparator = new StudentIDComparator( ) ;
        Arrays.sort(array, idComparator) ;
        for (int i = 0 ; i < n ; i++) // 整列結果の配列を表示する
            array[i].printStudentData( ) ;
    } // mainメソッドの終わり
} //StudentSorting2aaクラスの終わり
```

多様性

Comparator<StudentData>型の変数comparatorに2つの異なる型のインスタンスを代入できている．これは，StudentScoreComparatorもStudentIDComparatorもComparator<StudentData>型を実装しているから．

演習課題（必須）

課題：自作**mySort**メソッドでは挿入ソートを実装したが，quickソートを実装した別のメソッド **myQuickSort** を書いて動作を確認せよ．

採点者：副手（Chen研M2の山田君）

演習課題(選択)

ヒープ構造: 大小の比較が可能な一組のデータを記憶するためのデータ構造で, 下記の操作を提供するもの.

① 優先順位が一番高い要素を定数時間で返す.

② 優先順位が一番高い要素を対数時間で削除する.

③ 要素を対数時間で挿入する.

同じデータでも応用によって優先順位の決め方が変わってしまう. そのため, ヒープ構造で優先順位の決め方を自由に変えられるようにしたい.

課題: 優先順位の決め方を自由に変えられるようなヒープのクラスを作成せよ. それを使ってヒープソートのプログラムを書け.