# 3. オブジェクト指向プログラミング

2018/10/22

## 1　はじめに

前回は、Student クラスの配列に対して、泡立ち法を用いて整列を行いました。今回は、Student クラスのインスタンスをリストに保存して、整列を行いましょう。さらに、Student クラスに Comparable インターフェースを追加することで、整列を行うアルゴリズムの汎用性を高めます。

## 2　リストの利用

### 2.1　リストへの変更

前回の StudentMain の main メソッドの中では、Student クラスのインスタンスを students という配列に保存していました。今回は List<Student> students に保存しなおします。リストに保存することで、大きさの変更、要素の挿入など、柔軟な操作が可能となります。

ソースコード 2.1　List<Student> students への保存

```
1        //クラスインスタンスのリスト
2        List<Student> students
3            =Collections.synchronizedList(new ArrayList<>());
4        //配列に登録、及び成績登録
5        for (int i = 0; i < names.length; i++) {
6            Student s =new Student(names[i], i);
7            s.setRecord(records[i]);
8            students.add(s);
9        }
```

ソースコード 2.1 にデータを保存する部分を示します。また、印刷部分をソースファイル 2.2 に示します。

ソースコード 2.2　`List<Student> students` の印刷

```
1        //一覧を印刷
2        for (int i = 0; i < students.size(); i++) {
3            Student s = students.get(i);
4            System.out.println(s.getName() +
5                    "(" + s.getStudentID() + "):"
6                     + s.getRecord());
7        }
```

ここで、リスト (`java.util.List`) の基本的な操作をまとめておきます。ここで `E` は、リストに保存されるクラスを表します。

- `boolean add(E e)`：要素 `e` をリスト末尾に追加する。
- `boolean add(int index, E e)`：：要素 `e` を `index` で指定した位置に挿入する。
- `E get(int index)`：`index` で指定した位置にある要素を返す。
- `int indexOf(E e)`：要素 `e` のインデクスを返す。リストに無い場合には`-1` が返る。
- `int size()`：リストの長さを返す。

`java.util.List` は、インターフェースと呼ばれる特殊な抽象クラスであり、インスタンスを生成することはできません。そこで、例では、そのインターフェースを実装している `ArrayList` を用いています。`ArrayList` は複数スレッドからの操作の場合、その整合性が保証されていません。そこで、スレッド間の同期が可能なように、`Collections.synchronizedList()` メソッドを用いて生成しています。

**課題 1**　前回の `StudentMain` を変更して、`List<Student> students` を利用する形に変更しなさい。`sort` メソッド及び `sort` メソッドを呼び出している部分をコメントアウトして、コンパイルし、正しくコードできたことを確かめなさい。なお、`List` などとタイプした行には、バルーンで `java.util.List` を `import` に追加するなどのヒントが表示されます。バルーンから解決策を選択することで `import` を追加することができます。

## 2.2 整列

前回は、配列を用いた整列プログラムを sort に記述しました。今回は、リストで実装しましょう。

**課題 2**　sort メソッドにリストを引数として渡し、整列させるように変更しなさい。実行し、正しく実行できることを確かめなさい。

# 3 Comparable インターフェースの利用

## 3.1 Student クラスに Comparable インターフェースを追加

泡立ち法は、要素の間に大小関係が定義されていれば利用できる一般的手法です。クラスに Comparable インターフェースを実装することで、他のクラスから見て、そのクラスが比較する方法 compareTo を持っていることを宣言することができます。

既存のクラスに Comparable インターフェースを実装する方法は、以下の二つの手順で行います。最初にクラスに Comparable インターフェースを追記します。

```
public class Data implements Comparable<Data>
```

ここで、Comparable<Data>は、クラス Data と比較できることを表しています。比較の対象となるクラスを指定することが重要です。

クラスの宣言に上のように追記すると、クラス宣言の行にバルーンの注意がでます。そこにマウスを合わせると、compareTo をオーバーライド、つまり上書きしていないというメッセージがでます。そこでバルーンをマウス右ボタンでクリックすると、「すべての抽象メソッドを実装」が現れますから、それを選択することで、compareTo のテンプレートを生成することができます。

compareTo は、そのメソッドがあるクラスインスタンス (this) とメソッドの引数であるインスタンス object を比較し、

- this が大きければ正の整数
- this が小さければ負の整数
- this と object の大きさが等しければゼロ

を返す必要があります。

**課題 3**　クラス Student に Comparable インターフェースを追加しなさい。record フィールドの値で比較するように、compareTo メソッドを実装しなさい。

## 3.2　整列メソッドの一般化

泡立ち法は、要素の大小を比較できるならば一般的に利用できるアルゴリズムです。整列する対象毎に泡立ち法のプログラムを作成するのは無駄です。比較可能なクラス、つまり Comparable インターフェースが実装された任意のクラスに対応できるようにしましょう。

今回の前半で sort メソッドの引数として、Student クラスのリストに変更しました。これを、一般的な比較できる要素を持つリストに変更しましょう。つまり、Comparable インターフェースが実装されたクラスのリストを引数にします。

メソッドの引数のクラスを一般化する、つまり型パラメタにするには、メソッドの引数の前に型パラメタを定義します。今回の場合は

```
public static <T extends Comparable<T>> void sort(List<T> list)
```

とします。ここで、<T extends Comparable<T>>は、型パラメタ T は、Comparable クラスの拡張であることを示しています。つまり、型パラメタ T のインスタンスには compareTo メソッドがあることを示しています。

**課題 4**　sort メソッドの引数を型パラメタに変更しなさい。型パラメタ T のインスタンスには compareTo メソッドしかないことに注意して、比較の部分を変更しなさい。また、正しく整列ができることを実行結果で示しなさい。