ラムダ式と関連技術

オブジェクト指向プログラミング特論

2016年度

只木進一：工学系研究科

関数ポインタが無い

- C/C++には関数ポインタがある
  - 関数の引数に関数を渡すことができる
  - 例：Runge-Kutta法
    - 微分方程式の具体に関係ない、一般的方法
  - 例：台形公式を使った数値積分
- しかし、Java(<8)には、関数ポインタが無い！

一つしかメソッドの無いinterface

- 関数ポインタが無いことへの対応として、interfaceのインスタンスを渡す
- 一つしかメソッドが無いのだから、いちいちメソッド名を書かなくても良いでしょ
- 引数の型も分かっているのだから、いちいち型を指定しなくても良いでしょ
- 関数をデータとして扱える

型テンプレートとの齟齬

- Containerに型テンプレートが使える
- Containerに対する拡張for
  - 内容の型が分かっているのに、いちいち型を指定しなくて良いでしょ。

もっと、サボれるはず

関数の定義

- 関数テンプレートをinterfaceとして定義する

```
@FunctionalInterface
public interface インターフェイス名 {

    public 戻り値 関数名 (引数並び);

}
```

関数に渡す

- 定義済みの関数の利用
  - BinaryOperator<T>
  - クラスTのインスタンスを二つ引数に持ち、戻り値もTである関数applyのテンプレートが定義されている。
- https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html

関数に渡す

- 例：LambdaSample
  - 引数：(x,y)->{x+y*y;}
  - 受け取り側

```
new BinaryOperator<Double>(){
    Double apply(Double x, Double y){return x+y*y;}
}
```

```
static public double eval(
        List<Double> list, BinaryOperator<Double> op) {
    double result = 0;
    for (Double d : list) {//計算内容を直接記述
        result = op.apply(result, d);
    }
    return result;
}

/**
 * @param args the command line arguments
 */
public static void main(String[] args) {
    List<Double> list = Arrays.asList(new Double[]{0.1, 0.2, 0.8, 0.9});
    double result = LambdaSample.eval(list,
            (x, y) -> {
                return x + y * y;
            });
    System.out.println(result);
}
```

Collection.stream()

- Listなどの各要素に何かする

```
List<T> list;
for (T t:list){処理}
```

- 内容の型は分かってる

```
List<T> list;
list.stream(). forEachOrdered(
    (p) -> {処理;}
);
```

例：Runge-Kutta法

- 独立変数xに対して従属変数$\vec{y}$を記述する連立微分方程式

$$\frac{\mathrm{d}}{\mathrm{d}x}\vec{y} = \vec{f}\left(x, \vec{y}\right)$$

- 独立変数xをh刻みで増加させ、従属変数の列を得る

$$\left(x_n, \vec{y}_n\right) \to \left(x_{n+1} = x_n + h, \vec{y}_{n+1}\right)$$

$$\vec{k}_1 = h\vec{f}\left(x_n, \vec{y}_n\right)$$

$$\vec{k}_2 = h\vec{f}\left(x_n + \frac{h}{2}, \vec{y}_n + \frac{\vec{k}_1}{2}\right)$$

$$\vec{k}_3 = h\vec{f}\left(x_n + \frac{h}{2}, \vec{y}_n + \frac{\vec{k}_2}{2}\right)$$

$$\vec{k}_4 = h\vec{f}\left(x_n + h, \vec{y}_n + \vec{k}_3\right)$$

$$\vec{y}_{n+1} = \vec{y}_n + \frac{1}{6}\left(\vec{k}_1 + 2\vec{k}_2 + 2\vec{k}_3 + \vec{k}_4\right) + O\left(h^5\right)$$

連立方程式を表すインターフェース

```
@FunctionalInterface
public interface DifferentialEquation {
    public double[] rhs(double x, double y[]);
}
```

Runge-Kutta法の実行メソッド

```
/**
* One step from x to x + h
* @param x initial value of independent valiable
* @param y initial values of dependent valiables
* @param h step
* @param eq class contains differential equations
* @return next values of dependent valiables
*/
public static double[] rk4(double x, double y[],
double h, DifferentialEquation eq)
```

Runge-Kutta法の例 減衰振動

- 位置xと速度vの連立方程式

$$\frac{\mathrm{d}v}{\mathrm{d}t} = -\frac{k}{m}x - \frac{\gamma}{m}v$$

$$\frac{\mathrm{d}x}{\mathrm{d}t} = v$$

微分方程式の定義部分

```
equation = (double xx, double[] y) -> {
        double dy[] = new double[2];
        dy[0] = y[1];
        dy[1] = -k * y[0] - lambda * y[1];
        return dy;
    };
```