Javaプログラミング（11）
ラムダ式と関数型

成蹊大学理工学部
情報科学科

関数

- 関数 $f(x) = x^2$
- これを以下のように記載しよう
- $x \longmapsto x^2$
- これを一般化すると

  (引数1の型 引数1, … 引数nの型 引数n) -> 処理ブロック
- プログラムでは以下のように記述する

  (double x) -> {return x * x;}
- このような書き方をラムダ式という．つまり，ラムダ式は名前のないメソッドである．
- ラムダ式は，関数をデータとして定義するともいえる．

Javaにおけるラムダ式の実装

- 関数型インタフェース
  - 関数の型として用いられるインタフェース
  - 抽象メソッドを1つだけ持つ

```
public interface UnitFun {
    double apply(double x);
}
```

UnitFun型の変数squareにdoubleからdoubleへの$x \longmapsto x^2$という関数を代入．それを入力値0.5に適用．

```
public class SquareFun{
    public static void main(String[] args) {
        UnitFun square = (double x) -> {return x*x;};
        System.out.println(square.apply(0.5));
    }
}
```

様々なラムダ式

- 記述の簡略化
  (double x) -> {return x*x;};
- インタフェースの定義より，解釈できる部分は省略可能
  x -> x*x;
- 抽象メソッドが1つのインタフェースであれば，関数型インタフェースとなる．

```
public interface PrintFun {
    void bi_print (double x,  double y);
}

public class RectangleFun {
    public static void main(String[] args) {
        PrintFun pf = (x, y)->
                System.out.println(x+","+y);
        pf.bi_print(0.3, 0.7);
    }
}
```

メソッドとラムダ式

- ラムダ式をメソッドの引数や戻り値にすることができる.

```
public class RectangleCal2 {
    public static void main(String[] args) {
        RectangleCal2 rc2 = new RectangleCal2();
        rc2.printResult(0.5, process(2));
    }

    static UnitFun process(int a) {
        System.out.println("係数の値:"+a);
        return x -> a * (x-1);
    }

    void printResult (double d, UnitFun uf) {
        System.out.println(uf.apply(d));
    }
}
```

メソッド参照

- jdkであらかじめ定義されているメソッドを用いて関数型インタフェースのインスタンスを作成できる.
- これをメソッド参照という.
- メソッド参照の種類
  - クラス名::staticメソッド名
  - オブジェクト名::インスタンスメソッド名
  - クラス名::インスタンスメソッド名

```
static UnitFun process(int a) {
    return Math::sqrt;
}
```

前頁のプログラムで利用すると，ルートの値が出力される.

java.util.function

- java.util.functionパッケージにある関数型インタフェース
- よく使う関数型インタフェースをまとめたパッケージが提供されている

	インタフェース	メソッド
	Consumer <T>	void accept (T t)
	BiConsumer <T, U>	void accept (T t, U u)
値を返す	Supplier<T>	T get()
	Function<T,R>	R apply(T t)
	BiFunction<T,U,R>	R apply(T t, U u)
真偽値を返す	Predicate<T>	boolean test(T t)
	BiPredicate<T,U>	boolean test(T t, U u)
演算結果を返す	UnaryOperator<T>	T apply(T t)
	BinaryOperator<T>	T apply(T t, T u)

java.util.functionの利用例

ラムダ式を使わない(昔の)方法

```
import java.util.ArrayList;
import java.util.function.Consumer;

public class ConsumerSample implements Consumer <String>{
    static ArrayList <String> alist = new ArrayList <String> ()

    public static void main(String[] args) {
        alist.add("apple"); alist.add("orange"); alist.add("banana");
        method1();
    }

    public void accept(String t) { //抽象メソッドacceptをオーバーライド
        System.out.println(t);
    }

    static void method1 (){
        ConsumerSample cs = new ConsumerSample();
        for(String str : alist) {
            cs.accept(str);
        }
    }
}
```

java.util.functionの利用例

ラムダ式を使う方法

```
public class ConsumerSample {
    static ArrayList <String> alist = new ArrayList <String> ();

    public static void main(String[] args) {
        alist.add("apple"); alist.add("orange"); alist.add("banana");

        // (2) ラムダ式
        alist.forEach(s -> System.out.println(s));

        // (3) メソッド参照
        alist.forEach(System.out::println);
    }
}
```

forEach()メソッドは，Iterableインタフェースのメソッド
default void　forEach(Consumer<? super T> action)　を実装したクラスで使用できる
forEach()メソッドの引数に，Consumerの実装をラムダ式で渡している

無名のインナークラス

- インスタンスを作成する時(newする時)にクラス定義を一緒に記述する.
- そのクラスが一度しか参照されない場合などでは, クラスをわざわざ定義するのも面倒なので, このような簡易的な内部クラスの記述をすることがある.

無名のインナークラスの定義方法

```
new クラス or インターフェース() {
        メソッドの定義
    }
```

無名のインナークラス実装例

```
interface MyInterface {
    String getString();
}
public class NonameInnerSample{
    public static void main(String[] args) {
        // インタフェースを実装し，作ったオブジェクトを渡す
        System.out.println(new MyInterface () {
            public String getString() {
                return "無名内部クラス";
            }
        });

        // メソッドを作り、そのメソッドを呼ぶ
        new Object() {
            void test(){
                System.out.println(this.getClass()); } }.test();
    }
}
```