Functional Programming in Java (Functional Interface)

람다 식 (Lambda Expressions) 사용 동기

- 람다 식
  - 함수 이름이 없는 수식 (expression)
  - 인수의 순서 (입력), 함수의 body, 결과 값 (출력) 의 형태는 그대로 유지함

- 예

```
static boolean lessThan3 (int x) { return (x < 3) }
[1,2,3,4].map(lessThen3) = [True, True, False, False]

=>
[1,2,3,4].map(x -> x<3)  = [True, True, False, False]
```

고차 함수의 개념 (함수가 First-class Citizen)

- 람다 계산법(Lambda Calculus) 의 이론
  - Curried Expression, β-reduction, Pure Functions, Type Theory 등

- 함수를 수식으로서 표현 (즉, 함수를 마치 산술식처럼 사용)

- 함수가 할당문의 오른쪽에 나타날 수 있음

```
Function<Integer,Boolean> x = x ->  x < 3;
```

- 함수가 파라메터에 나타날 수 있음

```
static … f (Function<T,R> k) { … }
```

- 함수를 저장할 수 있음

```
List<Function<Integer,Boolean>> functions = [x -> x>2, x -> x==3, x -> even(x) ]
```

- 결과 값이 함수가 될 수 있음

```
static Function<Integer,Boolean> fun( ) { return x -> x <2; }        // fun().apply(3)
```

Static과 Instance 메소드에서의 파라메터 전송

```
class M {
  public int s, a, d;
  public M(int s, int a, int d) {
     this.s = s; this.a = a; this.d = d;
  }

  public int rating()  {
    return this.s + this.a + this.d;
  }
}

public class Test {
  public static void main (String argv[]) {
     M m = new M(8, 9, 6);
     System.out.print(“Result :"+ m.rating());
  }
}
```

```
class M {
  public int s, a, d;
  public M(int s, int a, int d) {
     this.s = s; this.a = a; this.d = d;
  }

  public static int rating(M m)  {
    return m.s + m.a + m.d;
  }
}

public class Test {
  public static void main (String argv[]) {
    M m = new M(8, 9, 6);
    System.out.print(“Result :" + M.rating(m));
  }
}
```

Functional Interface : 함수를 위한 타입의 표현

- 람다 식 (함수)를 표현할 수 있는 타입은?

  T x = a -> …

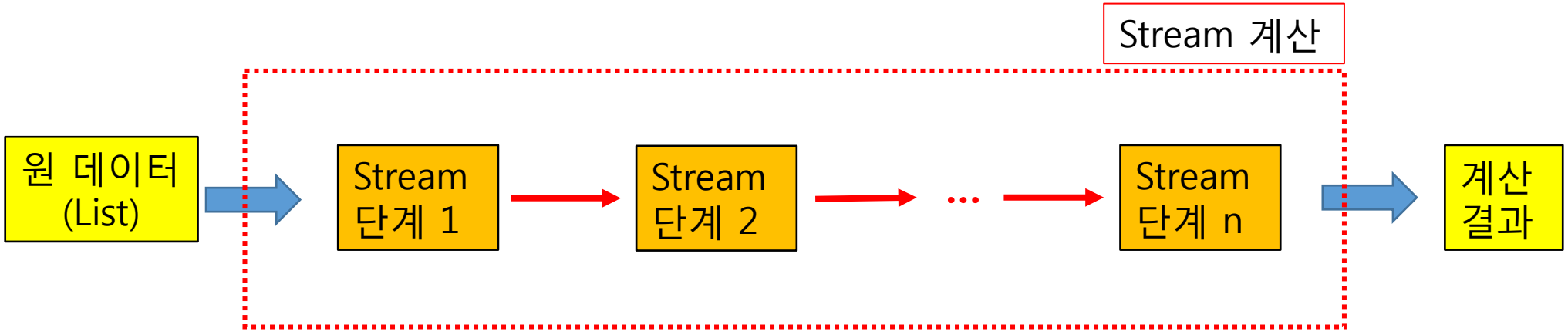
  - T는 Functional Interface (Annotation : @FunctionalInterface)

- Functional Interface
  - 단 하나의 abstract method로만 구성됨
  - (추가적으로 default 및 static 메소드를 포함할 수도 있음)
  - Functional Interface는 람다 식과 method reference의 타입 으로 이용될 수 있음

Stream 프로그래밍

1. List 등을 Stream으로 변환하여, 함수형 프로그래밍을 적용함

2. Stream 형태의 데이터 구조
   - Stream 형태로 전환
     - stream, of
   - Stream의 함수형 operators (pipeline 형태)
     - map, filter, sorted, 등의 함수 적용
   - Stream 종료 (다른 타입으로 전환)
     - reduce, collect, count, sum 등



람다 함수에 파라메터 전달

```
@FunctionalInterface
interface Square {
    int calculate(int x);
}

class FunInterface {
    public static void main(String args[]) {
        int a = 5;
        Square s = x -> x*x + a;
        int ans = s.calculate(a);
        System.out.println(ans);
        a = a + 1; // error
    }
}
```

1. Functional Interface 정의
   - 오직 하나의 추상 메소드
2. Interface 타입의 객체를 람다 식으로 정의 (함수)
   - 입출력 타입 일치
3. 함수 호출(파라메터 전송)
   - 인터페이스의 메소드 이용
4. 람다 식 내의 의 변수는 (effectively) final
   - Single assignment

Binary Functions

```
@FunctionalInterface
interface M { int max(int x, int y); }          // binary

class Max {
    public static void main(String args[]) {
        M m = (a, b) -> a > b ? a : b;          //binary
        int result = m.max(10, 20);
        System.out.println(result);
    }
}
```

No Input and No Return Value

```
@FunctionalInterface
interface N { void method1(); }

class Void {
    public static void main(String args[]) {
        N f = () -> System.out.println("No return value");
      f.method1();

      Runnable f2 = () -> System.out.println("No return value");
      f2.run();
    }
}
```

함수를 파라메터로 전송

- Collection (Iterable 인터페이스) 의 forEach

  void forEach (Consumer<? super T> action)

  - forEach의 인수는 FI (Functional Interface) 의 instance (즉, 함수)
  - forEach의 인수로 람다 식과 method reference 를 사용할 수 있음
  - Inline implementation of functional interface

- FI 패키지 java.util.function
  - Runnable 인터페이스 run 메소드
  - Supplier 인터페이스 get 메소드
  - Consumer인터페이스 accept 메소드
  - Function 인터페이스 apply 메소드
  - Predicate 인터페이스 test 메소드

```
List<String> items = new ArrayList<>();
items.add("A"); items.add("B"); items.add("C"); items.add("D"); items.add("E");

for(String item : items)
    System.out.print(item + " " );

items.forEach(item -> System.out.print(item + " ")); // lambda

items.forEach (item -> {
    if("C".equals(item))
        System.out.print(item + " ");                // Output : C
});

items.forEach(System.out::print);                    // method reference

items.stream()                                       // Stream and filter, Output: B
        .filter(s->s.contains("B"))
        .forEach(System.out::print);
```

람다를 사용하는 재귀함수

```
import java.util.function.*;

public class FacRec {
    static final Function<Integer,Integer> factorial =
            x -> x == 0 ? 1 : x * FacRec.factorial.apply(x-1);

    final UnaryOperator<Integer> factorial2 = x -> x == 0   // instance
            ? 1 :  x * this.factorial2.apply(x-1);

    public static void main(String[] args) {
            System.out.println("factorial.apply(4) : " + FacRec.factorial.apply(4));
            FacRec a = new FacRec();                a.f();
    }

    void f() {  // instance functions are called within instance functions
            System.out.println("factorial2.apply(4) : " + factorial2.apply(4));
    }
}
```

리스트 처리를 위한 재귀 함수

주어진 리스트의 맨 앞 원소를 빼내고, 나머지 원소들을 가지고 새로운 리스트를 만들어서 재귀적으로 처리함 (리스트의 원소 수가 하나 감소됨)

- Haskell 의 경우

```
sum xs = if x==[] then 0 else head(xs) + sum tail(xs)
```

- Java의 람다를 이용하는 재귀함수

```
static final Function<ArrayList<Integer>,Integer> sum = xs -> xs.isEmpty()
        ? 0
        : xs.get(0) + ListApply.sum.apply(new ArrayList<Integer>(xs.subList(1,xs.size())));

ArrayList<Integer> list1 = new ArrayList<Integer>() {{add(2);add(4);add(3);add(7);add(5);}};

Integer result = sum.apply(list2);
```