Java의 Functional Interface 및 Method Reference 실습

1 Functional Interface 만들기

1.1 입력/출력 타입이 특정된 경우

```
@FunctionalInterface
interface OneInt2Int { int apply(int x); }          // int -> int
interface TwoInt2Int { int apply(int x, int y); } // (int, int) -> int
interface OneInt2Void { void consume(int x); }     // int -> void
interface Void2Int { int supply(); }               // () -> int
```

1.2 제너릭스(타입변수)로 정의되는 경우

```
interface One2One <T,R> { R apply(T x); }          // T -> R
interface Two2One<T,U,R> { R apply(T x, U y); }    // (T, U) -> R
interface One2Void<T> { void consume(T x); }       // int -> void
interface Void2One<R> { R supply(); }              // () -> R
```

1.3 import java.util.function.*; 의 Functional Interface

자체적으로 만들지 않고 시스템이 제공하는 Functional Interface 이용할 때

```
// Unary Function (1 arguments)
interface Function<T,R> { R apply(T t); }          // T -> R
interface Predicate<T>  { boolean test(T t); }     // T -> boolean
interface Consumer<T>   { void accept(T t); }      // T -> ()
interface Supplier<T>   { T get(); }               // () -> T

// Binary Function (2 arguments)
interface BiFunction<T,U,R> { R apply(T t, U u); }         // (T,U) ->  R
interface BiPredicate<T,U>  { boolean test(T t, U u); }    // (T,U) -> boolean
interface BiConsumer<T,U>   { void accept(T t, U u); }     // (T,U) -> ()

// Input and output type are the same (Single Type)
interface UnaryOperator<T>  { T apply(T t); }          // T -> T
interface BinaryOperator<T> { T apply(T t, T u); }     // (T, T) -> T
```

2 람다 계산법의 beta(β) Conversion

- $(\lambda x.M)a = M[a/x]$
  추상화 된 람다 식이 파라메터 a 에 apply 되었을 때, M 안의 모든 free 변수들을 a 로 치환(substitution) 한다.

```
jshell> Function<Integer,Integer> square = x -> x*x;
jshell> square.apply(3)
$ ==> 9
```

- 위의 상황은 람다 식으로 정의된 square 함수에 파라메터 3을 전달하는 **함수 호출 (function call)** 인 데, 함수형 프로그래밍에서는 함수 호출이라는 표현 대신, 람다 식 함수 `square` 가 인수 3에 **apply** 된다는 표현을 사용한다.
- Java의 Functional Interface의 추상 메소드 이름이 apply인 것은 이러한 연유에 기반한다.
- 람다 식의 apply 는 이론적으로 β conversion 에 해당된다. 위의 상황은 이론적으로 $(\lambda x.x \cdot x)3 = 3 \cdot 3 = 9$ 에 해당된다.
- β conversion의 실행은 Java에 내재된 β reducer에 의해서 이루진다.
- 람다 식을 사용하는 모든 프로그래밍 언어는 언어 자체에 β reducer를 구현하여 내장하고 있다.

3 문제

- `jshell -v` 를 실행하고
- 위의 Interface 중에서 패키지를 제외한 8개의 Interface를 jshell에 입력할 것 (copy-paste 등 이용 가능).
- jshell에서 `/list` 명령어를 실행하여 8개의 interface가 입력되었는지를 확인할 것.

```
jshell> /list
   1 :   interface OneInt2Int { int apply(int x); }
   2 :   interface TwoInt2Int { int apply(int x, int y); }
   3 :   interface OneInt2Void { void consume(int x); }
   4 :   interface Void2Int { int supply(); }
   5 :   interface One2One <T,R> { R apply(T x); }
   6 :   interface Two2One<T,U,R> { R apply(T x, U y); }
   7 :   interface One2Void<T> { void consume(T x); }
   8 :   interface Void2One<R> { R supply(); }
```

3.1 각 람다 식의 타입 표현하기

- 위에 있는 Functional Interface 중에서 다음 각 람다 식 앞의 밑줄 친 부분에 적용될 수 있는 것 들을 모두 써라. 이때 제너릭스인 경우, 해당 타입을 또한 함께 표현한다.
- 이 타입이 적합한 지를 jshell 에서 확인하라. 문제가 있으면 에러 메시지를 발생시킨다.

```
_________ f1 = x -> x + 1;                          // 4 types

_________ f2 = x -> x > 10;                         // 3 types

_________ f3 = (x,y) -> x + y;                      // 4 types

_________ f4 = (x,y) -> x > y;                      // 2 types

_________ f5 = () -> (int)(Math.random()*100)+1;    // 2 types

_________ f6 = x -> System.out.println(x);          // 3 types

_________ f7  = (x,y) -> System.out.println(x+y); // 1 type
```

3.2 람다 식에 인수 전달하기 (parameter passing)

- 위의 7개 람다 식에 대해서 다음과 같이 적절한 값을 전달하고 그 결과를 확인하다.

```
jshell> Function<Integer,Integer> f1 = x -> x + 1;
f1 ==> $Lambda$22/0x00000008000b6c40@6eebc39e
|  created variable f1 : Function<Integer, Integer>

jshell> f1.apply(10)
$16 ==> 11
```

4 Stream 연산 (ListDemo.java 파일 참조)

- List는 mutable 하므로 함수형 프로그래밍을 적용할 수 없으며, 이를 Stream 으로 전환해야 한다.
- Stream은 immuable 하지만, 한 번 만들어진 Stream은 오직 한 번만 사용할 수 있다.
- 한 List에 대해서 여러 번의 Stream을 적용하기 위해서는 List를 Stream으로 만드는 일을 반복해야 한다.
- Stream에 적용할 함수형 연산자들

```
stream, forEach, map, filter, sorted, findFirst, peek, limit
```

- 이 함수들에 대한 설명은 ch14_Lambda_Stream_1_20160404.pdf 파일의 2.1~2.4 까지, 인터넷, 매뉴얼 참조.

```
https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html
```

4.1 문제

- 위에 명시한 함수들을 이용하여 다음을 jshell -v 를 적용하여 실습해 볼 것.

```
jshell> List<Integer> list = List.of(3,1,6,2)
list ==> [3, 1, 6, 2]
|  created variable list : List<Integer>

// convert list to stream
jshell> Stream<Integer> strm = list.____()

// multiply 10 to each element
jshell> strm.______.forEach(x -> System.out.println(x))
30
10
60
20

// sort the stream
jshell> list.stream()._______.forEach(System.out::println)
1
2
3
6

// select numbers with greater than 2
jshell> list.stream().________.forEach(System.out::println)
```

```
3
6

// find the first element from the stream, whose type is of Optional. apply get()
jshell> list.stream()._______.get()
$ ==> 3

// take 3 elements in the front
jshell> list.stream().peek(x -> System.out.println(x))._______.count()
3
1
6
$31 ==> 3
```

5 Method Reference 이해하기

5.1 람다 계산법의 eta (η) Conversion

- $\lambda x.fx \equiv f$
- 왼쪽: free 변수 x 가 없는 f 가 x 에 apply 된 후, x 에 의해서 람다 추상화 된 형태
- 오른쪽: 람다 추상화 및 함수 apply 를 생략하고, f 만으로 표현.
- 이 등식의 타당성은 임의의 식 a 를 양쪽에 apply 하였을 경우 동등한 결과를 얻는 것에 기반한다. $(\lambda x.fx)a = fa \iff (\lambda x.fx) \equiv f$
- 람다 계산법의 eta (η) 변환 규칙은 Java에서 method reference 라는 표현으로 불리고 있다. 이 규칙은 람다 식의 표현을 간결하게 하므로, 실용적으로 매우 유용하다.
- Java의 Method Reference 예

```
Function<Integer,Integer> square = x -> x*x;
Stream.of(2,3,4).map(a -> square.apply(a)).forEach(x -> System.out.println(x))
Stream.of(2,3,4).map(square).forEach(System.out::println)
```

- 위의 수행 결과 두 번째와 세 번째 모두 동일한 `4 9 16` 을 출력한다. 두 번째는 `map` 과 `forEach` 의 인수를 람다 식으로 표현한 것이고, 세 번째는 method reference 형태로 표현한 것이다.
- 람다 함수 `square` 가 free 변수 `a` 를 내포하지 않으므로, 이론적으로 $(\lambda a.square\ a) \equiv square$ 이 성립하며, `map(a -> square.apply(a))` 은 `map(square)` 으로 대신 표현될 수 있다.

5.2 `map, filter` 함수

- 이 두 함수는 Haskell의 list와 Java의 Stream에 사용되는 대표적인 함수로서, 이 둘 모두 iterator 기능을 내장하고 있음 (재귀적으로 정의됨).
- 이 함수들은 인수로서 스트림과 함수를 갖는데 (`strm.map(function)`), 이 `function` 은 람다 식이나 method reference 형태로 표현될 수 있다.
- 람다 식 표현 형태: `strm.map(x -> x.method())`. `strm` 의 각 객체 `x` 에 대해 `x.method()` 를 적용하여, 변화된 `strm` 의 얻는다.
- `map` 은 계산 과정에서 `strm` 원소들의 순서나 갯 수를 변화시키지 않는다 (structure-preserving).
- 람다 식 대신 method reference로서 표현할 수 있다. `method` 가 `A` 클래스에 소속된 경우 `map(A::method)` 형태로 표현된다.

5.3 Method Reference 코딩 예

- 프로그램 수행 결과 출력되는 내용

```
[A{name=John, age=10}, A{name=Ted, age=14}]
10
14
John
Ted
[John, Ted]
[John, Ted]
```

- 아래 코드를 실행시켜 위의 결과가 되도록 빈칸을 완성하세요.

```
import java.util.List;
import java.util.stream.*;

public class A {
  String name;   Integer age;
  A(String name, Integer age) { this.name = name; this.age = age; }
  String getName() { return this.name; }
  Integer getAge() { return this.age; }
  @Override
  public String toString() {
    return "A{" + "name=" + name + ", age=" + age + '}';
  }

  public static void main(String[] args) {


    List<A> list = List.of(new A("John",10), new A("Ted",14));
    System.out.println(list);

    Stream<A> strm = list.stream();              // convert to stream

    // apply getAge in A to each element (method reference)
    strm.map(________).forEach(System.out::println);

    Stream<A> strm2 = list.stream();             // conver to stream

    // apply getName in A to each element  (method reference)
    strm2.map(________).forEach(System.out::println);

    List<String> names
      = list.stream().map(A::getName).collect(Collectors.toList());
    System.out.println(names);

    List<String> names2
     // in lambda expression
      = list.stream().map(__________).collect(Collectors.toList());
    System.out.println(names2);
  }
}
```