Java 8 - Lambda Expression Ⅱ

함수형 프로그래밍 (Functional Programming)?

우선 위키피디아에 설명하는 함수형 프로그래밍에 대한 내용이다

- 함수의 응용을 강조하는 프로그래밍
- 함수를 값(Value)으로 취급한다.

- 상태와 가변 데이터 (변수)를 멀리하는 프로그래밍 패러다임 - Stateless
- 함수 정의와 응용, 재귀를 연구하기 위한 람다 대수에 근간을 둔다.

Robert C. Martin이 설명하는 내용

Functional Programming is Programming without assignment statements
(함수형 프로그래밍은 대입 연산문이 없는 프로그래밍이다.)

Java에서 함수형 프로그래밍 생각해보기

```
f : N -> Z

// 메소드로 함수 표현하기
Z function(N x){
    return 4 - x;
}
```

// 객체로 함수 표현하기

```
Function<N, Z> function = new Function<N,Z>() {
    @Override
    Z call(N x) { return 4 - x; }
};
```

- 함수가 객체화 되면서 함수자체가 값(데이터)이 되었다.

- 함수형 프로그래밍에서는 함수를 1급 객체로 취급한다.
  - 변수나 데이터 구조안에 담을 수 있다.
  - 파라미터로 전달이 가능하다.
  - 반환값으로 사용할 수 있다.

- 함수형 프로그래밍은 함수를 값(데이터)으로 취급한다.

Java에서의 람다식

```
Function<N, Z> function = new Function<N, Z>() {
    @Override
    Z call(N x) { return 4 - x; }
};
```

↓

```
Function<N, Z> function = x -> 4 - x
```

- f : (정의역 원소 x) -> (치역 원소 y)
- 함수에 입력값이 들어오면 정의역 N의 원소x 가 결정되므로
  치역 Z의 원소 y가 결정되어 함수값이 출력값이 된다.

T -> R

정의역 T -> 치역 R

함수를 데이터 화 시키기 위한 함수형 인터페이스

```
interface Function<T, R> {
    R apply(T t);
}
```

(파라미터) -> {코드블럭}

- **함수형 인터페이스 : abstract 메소드가 1개인 인터페이스**

```
public interface Comparator<T> {
    public T apply(T t1, T t2);
}

public interface Adder<T> {
    public T apply(T t1, T t2);
}

public class FunctionalDemo {
    public static void main(String[] args) {
        Comparator<Integer> comp = (t1, t2) -> {
            if(t1 > t2)
                return t1;
            else
                return t2;
        };

        Adder<Integer> adder = (t1, t2) -> t1 + t2; // return t1 + t2

        System.out.println(String.valueOf(comp.apply(1, 2)));
        System.out.println(String.valueOf(adder.apply(1, 2)));
    }
}
```

- **람다식의 기본 문법**

- 매개변수가 있는 경우
  `( int a, int b ) -> { System.out.println(a+b); }`

- 매개변수 타입은 런타임 시 대입값에 따라 자동인식 되므로 일반적으로 언급하지 않는다.
  `( a, b ) -> { System.out.println(a+b); }`

- 하나의 매개변수만 있다면 괄호 생략가능
  `a -> { System.out.println(a); }`

- 하나의 실행문만 있다면 중괄호 생략가능
  `a -> System.out.println();`

- 실행문이 return 문 하나라면 함수에서 치역에 Mapping하는 것과 같은 것이므로 return 키워드도 생략가능
  `a -> a+1`

- 매개변수가 없다면 반드시 괄호를 사용해야 한다.
  `( ) -> { System.out.println("apply!"); }`

- **람다식의 프리변수 (클래스, 로컬 멤버)**
  - 람다식은 클래스의 멤버를 제약없이사용가능하다.
    하지만, this의 경우 익명객체 자신의 참조가 아니라, 람다식을 호출한 바깥 객체의 참조가 된다.

  - 메소드안의 로컬 변수나 매개변수를 사용할 경우 두 로컬변수는 모두 Final 특성을 가져야 한다.
    (함수형 프로그래밍은 변수의 사용을 멀리하고, 변수 때문에 출력값이 바뀔수 있기 때문)

- **@FunctionalInterface 어노테이션**
  함수형 인터페이스를 작성할 때, 추상 메소드는 1개가 되야 하므로, 2개 이상의 추상 메소드가 선언되지 않도록 체크해주는 어노테이션 (2개 이상 시 컴파일 오류 발생시킴)

```
@FunctionalInterface
public interface MyFunctionalInterface<T> {
     public void method(T t);
     public void otherMethod(T t); // 컴파일 오류
}
```

- **Method Reference**
  메소드를 참조하여 매개변수와 리턴값의 타입을 알아내는 것
  람다식의 표현에서 불필요한 매개변수를 제거하고 람다식이 메소드에 값만 전달하는 목적으로 사용될때 사용

  - 인스턴스 메소드
    참조변수 :: 메소드

```
public class FunctionalDemo {
     public static void main(String[] args) {
          FunctionalDemo funcDemo = new FunctionalDemo();

          // Comparator<Integer> comp = (i1, i2) -> funcDemo.comparate(i1, i2);
          Comparator<Integer> comp = funcDemo :: comparate;

          System.out.println(String.valueOf(comp.apply(1, 2)));
```

```
    }

    public int comparate(int n1, int n2){
        if(n1 > n2)
            return n1;
        else
            return n2;
    }
}
```

- **정적 메소드**

  클래스 :: 메소드

```
public class FunctionalDemo {
    public static void main(String[] args) {
        // Comparator<Integer> comp = (i1, i2) -> FunctionalDemo.comparate(i1, i2);
        Comparator<Integer> comp = FunctionalDemo :: comparate;

        System.out.println(String.valueOf(comp.apply(1, 2)));
    }

    public static int comparate(int n1, int n2){
        if(n1 > n2)
            return n1;
        else
            return n2;
    }
}
```
