Zadanie G - Rozwijanie funkcji

Wraz z ukazaniem się wersji 8, świat programistyczny Javy się zmienił. Implementacja wyrażeń lambda i interfejsu funkcyjnego otwartła nowe możliwości, z których programiści z wielką radością korzystają, ścigając się z innymi językami posiadającymi już elementy funkcyjności.

Jedną z cech programowanie funkcyjnego, jest możliwość **rozwijania funkcji** znana jako **currying**. Operacja ta polega na spojrzeniu na funkcje wieloargumentowe, jako na ciąg funkcji jednoargumentowych. W uogólnieniu, dla funkcji kilku argumentowych, polega to na częściowym zaaplikowaniu argumentów, w wyniku którego dostajemy funkcję o mniejszej liczbie argumentów.

W Javie też jest to możliwe!

```
BinaryOperator<Integer> sum = (x,y) -> x + y;
System.out.println(sum.apply(10,5)); // 15

Function<Integer, Function<Integer, Integer>> sum_curry = x -> y -> x + y;
Function<Integer, Integer> plus10 = sum_curry.apply(10);  // y -> 10 + y
System.out.println(plus10.apply(5)); // 15
```

W powyższym przykładzie `sum` jest funkcją dwuargumentową. Tymczasem `sum_curry` jest funkcją jednoargumentową, która zwraca funkcję jednoargumentową. Ta druga możliwość pozwala na więcej zastosowań.

Wyobraźmy sobie, że chcemy w ten sposób przedstawić funkcje trójargumentowe, czteroargumentowe, itd.:

- `Function<Integer, Function<Integer, Function<Integer, Integer>>>`
- `Function<Integer, Function<Integer, Function<Integer, ...>>>`
- ...

Widać, że nie jest to zbyt poręczne. Wobec tego, po udanej implementacji generycznych funkcji wieloargumentowych, specjaliści od Javy z TCSLabs postanowili rozszerzyć język implementując **wygodny interfejs, który pozwoli na obsługę funkcji wieloargumentowych w sposób funkcyjny**.

1. Interfejs ogranicza się do funkcji, których argumenty i wyniki są typu `int`. Interfejs ma umożliwiać zaaplikowanie argumentów (zwracające wynik), częściowe zaaplikowanie argumentów (zwracające funkcję z podstawionymi początkowymi argumentami) oraz konwersję do funkcji jednoargumentowej.

2. Dodatkowo postanowiono rozszerzyć rozwijanie funkcji o możliwość aplikacji argumentów w innej kolejności (zwracające funkcję aplikującą argumenty w podanej kolejności), dzięki czemu możliwe jest np. zwrócenie funkcji z podstawionym już drugim argumentem.

3. Ponieważ interfejs nie specyfikuje liczby argumentów jakie wymaga funkcja, przewidziano możliwość obłożenia funkcji sprawdzeniem poprawności (not null) i oczekiwanej liczby argumentów.

4. Aby zachować interfejs funkcyjny, metody te są zaimplementowane jako metody domyslne interfejsu.

```
@FunctionalInterface
interface VarIntFunction extends ToIntFunction<int[]> {
    default int apply(int... args){...}
    default VarIntFunction applyPartial(int... partialArgs){...}
    default IntUnaryOperator unary(){...}

    default VarIntFunction orderArgs(int... argsOrder){...}
    default VarIntFunction checkArgs(int expectedArity){...}
}
```

Uwagi:

- należy dostarczyć jeden plik `VarIntFunction.java`, zawierający podany powyżej interfejs
- nie wolno implementować nowych klas ani klas anonimowych; **należy korzystać z funkcyjnego tworzenia nowych obiektów**
- w `orderArgs` argumenty numerowane są od 1 i mają być niepustą permutacją kolejnych indeksów `(1,2,3,...)`
- dla niepoprawnych argumentów (null, zła permutacja, arity<0) należy rzucić wyjątek `IllegalArgumentException`
- w `applyPartial` w przypadku braku podania argumentów należy zwrócić tę samą funkcję
- funkcja zwrócona przez `checkArgs` rzuca wyjątek dopiero w momencie wywołania jej na złych argumentach
- podobnie funkcja z `orderArgs` rzuca wyjątek jeśli podana zostanie inna liczba argumentów niż w permutacji

Przykładowe testy

```
@Test
public void simpleTest(){
    VarIntFunction f_xyz = A -> A[0]*A[0] + 2*A[1] + A[2]; // x^2 + 2y + z
    assertEquals(10,f_xyz.apply(1,2,5));

    VarIntFunction f_xy5 = f_xyz.orderArgs(3,1,2).applyPartial(5);
    assertEquals(10,f_xy5.apply(1,2));

    VarIntFunction f_yz = f_xyz.applyPartial(5);
    assertEquals(29,f_yz.apply(1,2));

    try{f_xyz.apply(1,2);} catch(Exception e) {
        assertEquals(ArrayIndexOutOfBoundsException.class, e.getClass());
    }
    try{f_xyz.checkArgs(3).apply(1,2);} catch(Exception e) {
        assertEquals(IllegalArgumentException.class, e.getClass());
    }
}
```

```
@Test
public void testDivisorsNumber(){
    VarIntFunction divisors_num = x -> {
        int num = x[x.length-1];
        for(int i=0;i<x.length-1;++i){
            if(num%x[i] == 0)
                return 0;
        }
        return num;
    };
    IntUnaryOperator isOdd = divisors_num.applyPartial(2).unary();
    IntUnaryOperator smallSieve = divisors_num.applyPartial(2,3,5,7,11,13,17,19).unary();

    assertArrayEquals(
        new int[]{0, 1, 0, 3, 0, 5, 0, 7, 0, 9},
        IntStream.range(0,10).map(isOdd).toArray()
    );
    assertArrayEquals(
        new int[]{31, 0, 0, 37, 0, 41, 43, 0, 47, 0, 0, 53, 0, 0, 59, 61, 0, 0, 67, 0},
        IntStream.iterate(31,i -> i+2).limit(20).map(smallSieve).toArray()
    );
}
```