# Laboratorium 11

**CEL:** Programowanie funkcyjne.

**Zadanie#1.** Przeanalizuj poniższy fragment kodu.

```
Stream<String> stream = Stream.iterate("", (s) -> s + "1");
System.out.println(stream.limit(2).map(x -> x + "2"));
```

Jaki będzie wynik wykonania (proszę zaznaczyć wszystkie poprawne odpowiedzi)?

| | |
|---|---|
| a) | `12112` |
| b) | `212` |
| c) | `212112` |
| d) | `java.util.stream.ReferencePipeline$3@4517d9a3` |
| e) | Kod się nie kompiluje. |
| f) | Wyrzucany jest wyjątek. |
| g) | Kod zawiesza się. |

■

**Zadanie#2.** Przeanalizuj poniższy fragment kodu.

```
Predicate<? super String> predicate = s -> s.startsWith("g");
Stream<String> stream1 = Stream.generate(() -> "growl! ");
Stream<String> stream2 = Stream.generate(() -> "growl! ");
boolean b1 = stream1.anyMatch(predicate);
boolean b2 = stream2.allMatch(predicate);
System.out.println(b1 + " " + b2);
```

Jaki będzie wynik wykonania (proszę zaznaczyć wszystkie poprawne odpowiedzi)?

| | |
|---|---|
| a) | `true false` |
| b) | `true true` |
| c) | `java.util.stream.ReferencePipeline$3@4517d9a3` |
| d) | Kod się nie kompiluje. |
| e) | Wyrzucany jest wyjątek. |
| f) | Kod zawiesza się. |

■

**Zadanie#3.** Przeanalizuj poniższy fragment kodu.

```
Predicate<? super String> predicate = s -> s.length() > 3;
Stream<String> stream = Stream.iterate("-", (s) -> s + s);
boolean b1 = stream.noneMatch(predicate);
boolean b2 = stream.anyMatch(predicate);
System.out.println(b1 + " " + b2);
```

Jaki będzie wynik wykonania (proszę zaznaczyć wszystkie poprawne odpowiedzi)?

| | |
|---|---|
| a) | `false true` |
| b) | `false false` |
| c) | `java.util.stream.ReferencePipeline$3@4517d9a3` |
| d) | Kod się nie kompiluje. |
| e) | Wyrzucany jest wyjątek. |
| f) | Kod zawiesza się. |

■

**Zadanie#4.** Które z poniższych stwierdzeń są prawdziwe dla operacji końcowych (terminal operations) w strumieniach (proszę zaznaczyć wszystkie poprawne odpowiedzi)?

| | |
|---|---|
| a) | Co najwyżej jedna operacja końcowa może istnieć w potoku strumienia |
| b) | Operacje końcowe są wymaganą częścią potoku strumienia umożliwiającą uzyskanie wyniku |
| c) | Zwracanym typem operacji końcowych jest `Stream` |
| d) | Dana referencja `Stream` może być nadal użyta po wywołaniu operacji kończącej |
| e) | Metoda `peek()` jest przykładem operacji kończącej |

■

**Zadanie#5.** Które z poniższych operacji końcowych (terminal operations) w strumieniach są redukcjami (reductions) (proszę zaznaczyć wszystkie poprawne odpowiedzi)?

| | |
|---|---|
| a) | `collect()` |
| b) | `count()` |
| c) | `findFirst()` |
| d) | `map()` |
| e) | `peek()` |
| f) | `sum()` |

■

**Zadanie#6.** Przeanalizuj poniższy fragment kodu.

```
Stream<String> s = Stream.generate(() -> "meow");
boolean match = s._________________(String::isEmpty);
System.out.println(match);
```

Które z poniższych instrukcji wstawione w brakujący fragment kodu sprawi, że zostanie wyświetlone `false` (proszę zaznaczyć wszystkie poprawne odpowiedzi)?

| | |
|---|---|
| a) | `allMatch` |
| b) | `anyMatch` |
| c) | `findAny` |
| d) | `findFirst` |
| e) | `noneMatch` |
| f) | Żadne z powyższych |

■

**Zadanie#7.** Przeanalizuj poniższy fragment kodu.

```
private static List<String> sort(List<String> list) {
     List<String> copy = new ArrayList<>(list);
     Collections.sort(copy, (a, b) -> b.compareTo(a));
     return copy;
}
```

Powyższa metoda zwraca posortowaną listę bez zmiany oryginalnej listy. Które z poniższych fragmentów kodu mogą zastąpić powyższą metodę w celu osiągnięcia takiego samego wyniku za pomocą strumieni?

| | |
|---|---|
| a) | `return list.stream()`<br>`.compare((a, b) -> b.compareTo(a))`<br>`.collect(Collectors.toList());` |
| b) | `return list.stream()`<br>`.compare((a, b) -> b.compareTo(a))`<br>`.sort();` |
| c) | `return list.stream()`<br>`.compareTo((a, b) -> b.compareTo(a))`<br>`.collect(Collectors.toList());` |
| d) | `return list.stream()`<br>`.compareTo((a, b) -> b.compareTo(a))`<br>`.sort();` |
| e) | `return list.stream()`<br>`.sorted((a, b) -> b.compareTo(a))`<br>`.collect();` |
| f) | `return list.stream()`<br>`.sorted((a, b) -> b.compareTo(a))`<br>`.collect(Collectors.toList());` |

■

**Zadanie#8**. Które z poniższych stwierdzeń są prawdziwe dla deklaracji `IntStream is = IntStream.empty()` (proszę zaznaczyć wszystkie poprawne odpowiedzi)?

| | |
|---|---|
| a) | `is.average()` zwraca typ `int` |
| b) | `is.average()` zwraca typ `OptionalInt` |
| c) | `is.findAny()` zwraca typ `int` |
| d) | `is.findAny()` zwraca typ `OptionalInt` |
| e) | `is.sum()` zwraca typ `int` |
| f) | `is.sum()` zwraca typ `OptionalInt` |

■

**Zadanie#9**. Przeanalizuj poniższy fragment kodu.

```
4: LongStream ls = LongStream.of(1, 2, 3);
5: OptionalLong opt = ls.map(n -> n * 10).filter(n -> n < 5).findFirst();
```

Które z poniższych instrukcji można dodać po wierszu 5, aby kod działał bez błędów i nie generował żadnych wyników (proszę zaznaczyć wszystkie poprawne odpowiedzi)?

| | |
|---|---|
| a) | `if (opt.isPresent()) System.out.println(opt.get());` |
| b) | `if (opt.isPresent()) System.out.println(opt.getAsLong());` |
| c) | `opt.ifPresent(System.out.println)` |
| d) | `opt.ifPresent(System.out::println)` |
| e) | Żadne z powyższych; kod się nie kompiluje. |
| f) | Żadne z powyższych; linia 5 wyrzuca wyjątek podczas wykonywania. |

■

**Zadanie#10**. Wskaż kolejność w jakiej powinny występować poniższe instrukcje, aby na wyjściu zostało wypisane 10 wierszy.

```
Stream.generate(() -> "1")
L: .filter(x -> x.length() > 1)
M: .forEach(System.out::println)
N: .limit(10)
O: .peek(System.out::println)
;
```

| | |
|---|---|
| a) | `L, N` |
| b) | `L, N, O` |
| c) | `L, N, M` |
| d) | `L, N, M, O` |
| e) | `L, O, M` |
| f) | `N, M` |
| g) | `N, O` |

■

**Zadanie#11.** Jakie zmiany należy wprowadzić do kodu, aby wyświetlał łańcuch `12345` (proszę zaznaczyć wszystkie poprawne odpowiedzi)?

```
Stream.iterate(1, x -> x++).limit(5).map(x -> x)
      .collect(Collectors.joining());
```

| | |
|---|---|
| a) | Zmienić `Collectors.joining()` na `Collectors.joining("")` |
| b) | Zmienić `map(x -> x)` na `map(x -> "" + x)` |
| c) | Zmienić `x -> x++` na `x -> ++x` |
| d) | Dodać `forEach(System.out::print)` po wywołaniu `collect()` |
| e) | Opakować całą linię instrukcją `System.out.print` |
| f) | Żadne z powyższych. Kod już wyświetla `12345`. |

■

**Zadanie#12.** Przeanalizuj poniższy fragment kodu.

```
6: _____________ x = String::new;
7: _____________ y = (a, b) -> System.out.println();
8: _____________ z = a -> a + a;
```

Które z poniższych interfejsów funkcyjnych wstawione w brakujące fragmenty uzupełniają poprawnie kod (proszę zaznaczyć wszystkie poprawne odpowiedzi)?

| | |
|---|---|
| a) | `BiConsumer<String, String>` |
| b) | `BiFunction<String, String>` |
| c) | `BinaryConsumer<String, String>` |
| d) | `BinaryFunction<String, String>` |
| e) | `Consumer<String>` |
| f) | `Supplier<String>` |
| g) | `UnaryOperator<String>` |
| h) | `UnaryOperator<String, String>` |

■

**Zadanie#13.** Przeanalizuj poniższy fragment kodu.

```
List<Integer> l1 = Arrays.asList(1, 2, 3);
List<Integer> l2 = Arrays.asList(4, 5, 6);
List<Integer> l3 = Arrays.asList();
Stream.of(l1, l2, l3).map(x -> x + 1)
  .flatMap(x -> x.stream()).forEach(System.out::print);
```

Które z poniższych stwierdzeń są prawdziwe (proszę zaznaczyć wszystkie poprawne odpowiedzi)?

| | |
|---|---|
| a) | Kod kompiluje się i wyświetla `123456` |
| b) | Kod kompiluje się i wyświetla `234567` |
| c) | Kod kompiluje się lecz niczego nie wyświetla |
| d) | Kod kompiluje się lecz wyświetla referencje do strumieni |
| e) | Kod działa nieskończenie. |
| f) | Kod się nie kompiluje. |
| g) | Kod wyrzuca wyjątek. |

■

**Zadanie#14.** Przeanalizuj poniższy fragment kodu.

```
4: Stream<Integer> s = Stream.of(1);
5: IntStream is = s.mapToInt(x -> x);
6: DoubleStream ds = s.mapToDouble(x -> x);
7: Stream<Integer> s2 = ds.mapToInt(x -> x);
8: s2.forEach(System.out::print);
```

Które z poniższych stwierdzeń są prawdziwe (proszę zaznaczyć wszystkie poprawne odpowiedzi)?

| | |
|---|---|
| a) | Linia 4 nie kompiluje się |
| b) | Linia 5 nie kompiluje się |
| c) | Linia 6 nie kompiluje się |
| d) | Linia 7 nie kompiluje się |
| e) | Linia 8 nie kompiluje się |
| f) | Kod wyrzuca wyjątek |
| g) | Kod kompiluje się i wyświetla 1 |

■

**Zadanie#15.** Kolektor `partitioningBy()` domyślnie tworzy `Map<Boolean, List<String>>`, gdy jest przekazywany do `collect()`. Które z poniższych zwracanych typów mogą być tworzone, gdy specyficzne parametry są przekazywane do `partitioningBy()` (proszę zaznaczyć wszystkie poprawne odpowiedzi)?

| | |
|---|---|
| a) | `Map<boolean, List<String>>` |
| b) | `Map<Boolean, Map<String>>` |
| c) | `Map<Long, TreeSet<String>>` |
| d) | `Map<Boolean, List<String>>` |
| e) | `Map<Boolean, Set<String>>` |
| f) | Żadne z powyższych |

■

**Zadanie#16.** Przeanalizuj poniższy fragment kodu.

```
Stream<String> s = Stream.empty();
Stream<String> s2 = Stream.empty();
Map<Boolean, List<String>> p = s.collect(
     Collectors.partitioningBy(b -> b.startsWith("c")));
Map<Boolean, List<String>> g = s2.collect(
     Collectors.groupingBy(b -> b.startsWith("c")));
System.out.println(p + " " + g);
```

Jaki będzie wynik wykonania?

| | |
|---|---|
| a) | `{} {}` |
| b) | `{} {false=[], true=[]}` |
| c) | `{false=[], true=[]} {}` |
| d) | `{false=[], true=[]} {false=[], true=[]}` |
| e) | Kod się nie kompiluje. |
| f) | Wyrzucany jest wyjątek. |

■

**Zadanie#17**. Przeanalizuj poniższy fragment kodu.

```
UnaryOperator<Integer> u = x -> x * x;
```

Które z poniższych jest równoważne z przedstawionym kodem?

| | |
|---|---|
| a) | `BiFunction<Integer> f = x -> x*x;` |
| b) | `BiFunction<Integer, Integer> f = x -> x*x;` |
| c) | `BinaryOperator<Integer, Integer> f = x -> x*x;` |
| d) | `Function<Integer> f = x -> x*x;` |
| e) | `Function<Integer, Integer> f = x -> x*x;` |
| f) | Żadne z powyższych |

■

**Zadanie#18**. Przeanalizuj poniższy fragment kodu.

```
DoubleStream s = DoubleStream.of(1.2, 2.4);
s.peek(System.out::println).filter(x -> x > 2).count();
```

Jaki będzie wynik wykonania?

| | |
|---|---|
| a) | `1` |
| b) | `2` |
| c) | `2.4` |
| d) | `1.2` i `2.4` |
| e) | Nic nie zostanie wyświetlone |
| f) | Kod się nie kompiluje |
| g) | Wyrzucany jest wyjątek |

■

**Zadanie#19**. Które z poniższych zwracają typy prymitywne (proszę zaznaczyć wszystkie poprawne odpowiedzi)?

| | |
|---|---|
| a) | `BooleanSupplier` |
| b) | `CharSupplier` |
| c) | `DoubleSupplier` |
| d) | `FloatSupplier` |
| e) | `IntSupplier` |
| f) | `StringSupplier` |

■

**Zadanie#20**. Przeanalizuj poniższy fragment kodu.

```
List<Integer> l = IntStream.range(1, 6)
     .mapToObj(i -> i).collect(Collectors.toList());
l.forEach(System.out::println);
```

W jaki najprostszy sposób poniżej można by było przepisać powyższy kod?

| | |
|---|---|
| a) | `IntStream.range(1, 6);` |
| b) | `IntStream.range(1, 6)`<br>`.forEach(System.out::println);` |
| c) | `IntStream.range(1, 6).mapToObj(1 -> i)`<br>`.forEach(System.out::println);` |
| d) | Żaden powyższy sposób nie jest równoważny |
| e) | Wyjściowy fragment kodu nie kompiluje się |

■

| *Poprawnych odpowiedzi* | *Ocena* |
|---|---|
| <0> | n/k |
| <1, 11> | 2.0 |
| <12, 13> | 3.0 |
| <14, 15> | 3.5 |
| <16, 17> | 4.0 |
| <18, 19> | 4.5 |
| <20> | 5.0 |