

# Streams – pokročilé příklady

# Úplně blbý příklad I.

- Máme pole stringů:

```
List<String> strings = new ArrayList<String>();  
  
for(int i = 0; i < 100_000; i++) {  
    strings.add("test");  
}
```

Poznámka: šlo by to přepsat takto:

```
IntStream.range(0, 100_000)  
    .forEach(e -> strings.add("test"));
```

- A chceme spočítat jejich počet:

```
long[] count = new long[1];  
  
strings.forEach(e -> count[0]++);  
  
System.out.println(count[0]);
```

- Poznámka: Toto není blbý příklad v jedné drobnosti ... bude to fungovat správně ...

# Úplně blbý příklad II.

- ... jenom v jedné drobné situaci to fungovat nebude:

```
long[] count = new long[1];  
  
strings.parallelStream().forEach(e -> count[0]++);  
  
System.out.println(count[0]);
```

- ... bylo by možné to vyřešit synchronizací, ale to bychom popírali význam parallelStreamu.

# Úplně blbý příklad III.

- Lepší řešení:

```
long count = strings.stream()  
    .mapToLong(e -> 1)  
    .reduce(0, (a, b) -> a + b);
```

- Nebo:

```
long count = strings.stream()  
    .mapToLong(e -> 1)  
    .reduce(0, Long::sum);
```

# Úplně blbý příklad IV.

- Nebo:

```
long count = strings.stream()  
    .mapToLong(e -> 1)  
    .sum();
```

- Nebo:

```
long count = strings.stream()  
    .count();
```

# Úplně blbý příklad V.

- Pro jednoduché otestování (vhodné pro ad-hoc debugování) můžeme zavolat metodu peek:

```
long count = strings.stream()
    .peek(System.out::println)
    .mapToLong(e -> 1)
    .sum();
```

- Poznámka: Tato metoda se zavolá na každém záznamu aktuálního streamu

- Příklad: tady se peek nezavolá:

```
long count = strings.stream()
    .filter(e -> false)
    .peek(System.out::println)
    .mapToLong(e -> 1)
    .sum();
```

← Díky tomuto filtru se nepokračuje ve vyhodnocování operací streamu.

# Úplně blbý příklad VI.

- V tomto případě záleží na implementaci JRE jestli peek něco vypíše nebo ne (v aktuální verzi Oracle JRE se peek zavolá, v Java 9 to bude naopak ... kvůli optimalizaci):

```
long count = strings.stream()  
    .peek(System.out::println)  
    .count();
```

- Co je důležité? Pozor na operace streamu, měly by měnit pouze stav daného streamu a nic jiného.
- Navíc by se měly používat pouze statické metody. Proč?
  - <http://blog.jooq.org/2015/11/10/beware-of-functional-programming-in-java/>

# Stream + Stream (+ Stream) I.

- Streamy také můžeme spojovat dohromady pomocí metody `concat`:

```
List<String> strings1 = new ArrayList<String>();  
IntStream.range(0, 10).forEach(e -> strings1.add("test 1"));  
  
List<String> strings2 = new ArrayList<String>();  
IntStream.range(0, 10).forEach(e -> strings2.add("test 2"));  
  
Stream<String> stream3 = Stream.concat(strings1.stream(), strings2.stream());  
  
System.out.println(stream3.count());
```

- Spojení 3 streamů:

```
Stream<String> stream3 = Stream.concat(strings1.stream(),  
                                     Stream.concat(strings2.stream(), strings3.stream()));
```



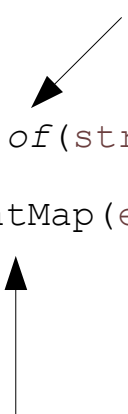
# Stream + Stream (+ Stream) II.

- Nebo:

Toto vrátí:

`Stream<Stream<String>>`

```
Stream<String> stream3 = Stream.of(strings1.stream(), strings2.stream())  
                                .flatMap(e -> e);
```



`flatMap()` slouží pro transformaci objektu streamu na jiný objekt.

Oproti funkci `map()` provede:

1) Transformaci struktury jako

`[ [ 1,2,3 ], [ 4,5,6 ] ]` na strukturu: `[ 1,2,3,4,5,6 ]`

2) Může provést transformaci hodnot streamu:

```
Stream<String> stream3 = Stream.of(strings1.stream(), strings2.stream())  
                                .flatMap(eStream -> eStream.filter(eString -> eString.equals("test 1")));
```

# Optionals Stream I.

- Máme stream, ve kterém jsou Optional prvky:

```
Stream<Optional<String>> optionalsStream  
    = Stream.of(Optional.of("A"), Optional.empty(), Optional.of("B"));  
optionalsStream  
    .forEach(System.out::println);
```

- Co toto vypíše?

# Optionals Stream II.

```
optionalsStream
```

```
.foreach(System.out::println);
```

... vypíše objekty typu Optional. Co kdybychom chtěli jejich hodnoty?

# Optionals Stream III.

- Řešení:

```
optionalsStream
```

```
.filter(Optional::isPresent)
.map(Optional::get)
.forEach(System.out::println);
```

← Pozor! Nemůžeme zavolat optional.get(),  
když neobsahuje prvek.

- Nebo pomocí flatMap() 1.:

```
optionalsStream
```

```
.flatMap(o -> o.isPresent() ? Stream.of(o.get()) : null)
.forEach(System.out::println);
```

Tady jsme mohli  
použít Stream.empty()

- flatMap() 2.:

```
optionalsStream
```

```
.flatMap(o -> o.map(e -> Stream.of(e)).orElse(Stream.empty()))
.forEach(System.out::println);
```

Tady jsme mohli  
použít null

Poznámka: V Java 9 bude v Optional metoda stream(), která toto zjednoduší.

# Arrays

- Ve třídě Arrays je několik funkcí pro práci se streamy.

# Lambda + výjimky

- Lambda výraz (nebo Method reference) nesmí vyhazovat výjimku. Jak to vyřešit?
  - try – catch blok ... to ale po chvíli omrzí
  - @SneakyThrows z Lomboku, což prakticky udělá to samé
  - Když tvoříte vlastní metody do kterých vstupuje @FunctionalInterface, pak můžete vytvořit vlastní, chytřejší. Například:

@FunctionalInterface

```
public static interface Consumer_WithExceptions<T, E extends Exception> {  
    void accept(T t) throws E;  
}
```

# GoF patterns & Functional programming

- <https://www.voxxed.com/blog/2016/04/gang-four-patterns-functional-light-part-1/>
- <https://www.voxxed.com/blog/2016/05/gang-four-patterns-functional-light-part-2/>
- <https://www.voxxed.com/blog/2016/05/gang-four-patterns-functional-light-part-3/>
- <https://www.voxxed.com/blog/2016/05/gang-four-patterns-functional-light-part-4/>

# Functional Interfaces

- Java 8 má tyto základní functional interfaces:

Functional Interfaces	# Parameters	Return Type	Single Abstract Method	
Supplier<T>	0	T	get	
Consumer<T>	1 (T)	void	accept	+ Runnable s metodou: void run()
BiConsumer<T, U>	2 (T, U)	void	accept	
Predicate<T>	1 (T)	boolean	test	
BiPredicate<T, U>	2 (T, U)	boolean	test	<b>Poznámky:</b>  T = 1. vstupní parametr U = 2. vstupní parametr R = výstupní parametr  Bi = Dva (vstupní parametry)
Function<T, R>	1 (T)	R	apply	
BiFunction<T, U, R>	2 (T, U)	R	apply	
UnaryOperator<T>	1 (T)	T	apply	
BinaryOperator<T>	2 (T, T)	T	apply	



# ::new

- Vytvoření objektu:

NazevTridy::new

# peek & anyMatch

- Je možné použít nekonečný stream například pro čtení souboru. S tím, že bychom chtěli po nalezení hledaného textu prohledávání ukončit. V Java 9 bude nová metoda takeWhile(), v Java 8 to musíme řešit tímto způsobem:

```
try (BufferedReader reader
    = Files.newBufferedReader(Paths.get("SOUBOR"))) {
    Stream.generate(() -> readLine(reader))
        .peek(e -> {
            if(e != null && e.contains("HLEDANY_TEXT")) {
                System.out.printf("OBSAHUJE NALEZENY TEXT%n");
            }
        })
        .anyMatch(e -> e == null || e.contains("HLEDANY_TEXT"));
```

# Fun. programování ve vlastních třídách

- Příklad 1:
  - Mějme 3 operace a u každé z nich chceme na začátku vypsát že operace byla zahájena a na konci chceme vypsát že byla ukončena a jak dlouho trvala:
  - <https://gist.github.com/jirkapinkas/5c37be59e35f0ab6a9cf8fe7702160bb>
- Příklad 2:
  - Potřebujeme udělat X rozdílných operací paralelně a spojit výsledky z nich do jednoho výstupu:
  - <https://gist.github.com/jirkapinkas/f7bdaca557280af436d8f577eedd53fb>

# Další knihovny

- Další knihovny:
  - <https://github.com/amaembo/streamex>
  - <https://github.com/jOOQ/jOOL>