Streams & Lambdas

Peter Borovanský
KAI, I-18

borovan 'at' ii.fmph.uniba.sk
http://dai.fmph.uniba.sk/courses/JAVA/

Streams & λ's
API Java 8

Prednáška prevzatá z

1-AIN-430: Programovacie paradigmy
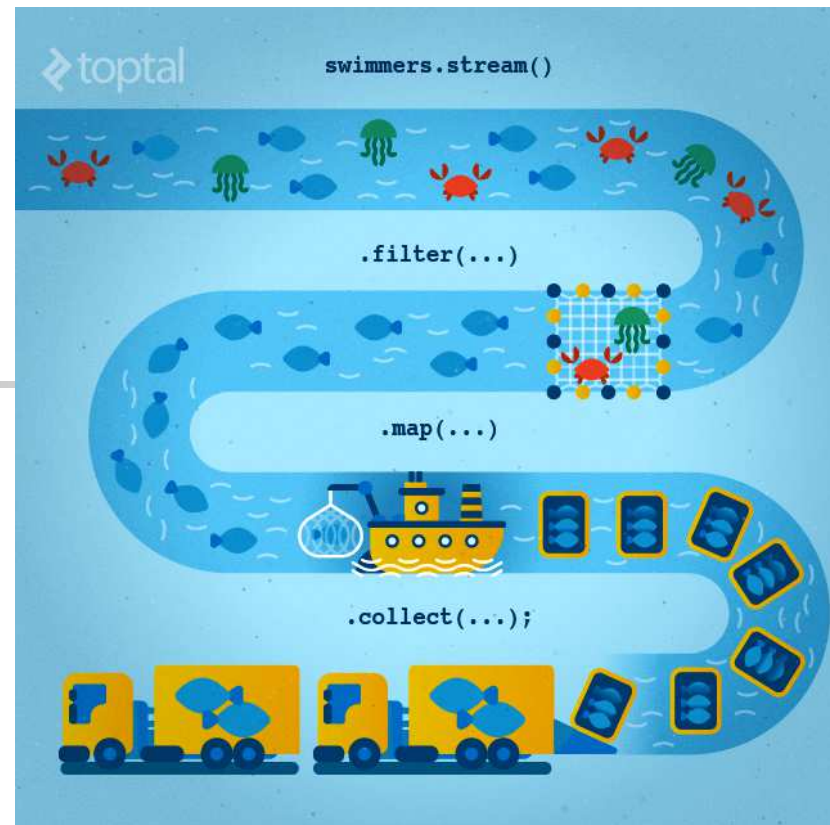
http://dai.fmph.uniba.sk/courses/PARA/

dnes bude:

- iný pohľad na prácu s kolekciami,
- ochutnávka funkcionálneho programovania,
- malý výlet do kombinatoriky, ako ju nepoznáte

Cvičenie:

- práca s kolekciami

< Java 8
Java 8
Java 9
Java 10

http://en.proft.me/2016/11/13/tutorial-lambda-expression-and-stream-api-java-8/

Kolekcie

(a práca s nimi – ako to poznáme)

```java
List<Integer> lst = new ArrayList<Integer>();
List<Integer> lst = new ArrayList<>();
ArrayList<Integer> lst = new ArrayList<>();
for (int i = 0; i < 100; i++)
    lst.add(i);
// explicitná inicializácia
List<Integer> lst1 = Arrays.asList(0,1,2,3,4,5,6,7,8,9);
// Nová syntax Java 9
List<Integer> list = List.of(0, 1, 2, 3, 4, 5, 6, 7, 8, 9);
Set<Integer> set = Set.of(0, 1, 2, 3, 4, 5, 6, 7, 8, 9);
Map<String,Integer> map = Map.of("Jano",1, "Palo",3, "Igor",0);

for (Integer value : lst)                  // foreach cyklus
    System.out.println(value);
lst.forEach(System.out::println);      // foreach metóda
lst.forEach(e -> System.out.println(e+e));
```

Anonymné funkcie v Jave

(lambdas)

Plná syntax pre zápis anonymnej funkcie:

```
(double a, double b) -> { return Math.sqrt(a*a+b*b); }
```

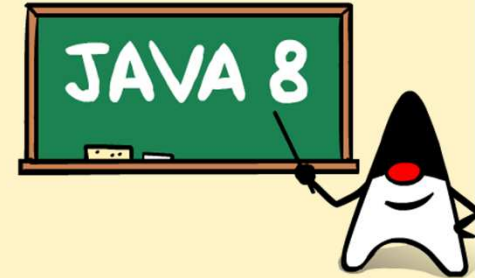Typová inferencia parametrov (odvodenie typu bez nutnosti typ explicitne uviesť)

```
(a, b) -> { return Math.sqrt(a*a+b*b); }
```

Syntax sugar, výsledkom je príjemná funkcionálna syntax:

```
(a, b) -> { Math.sqrt(a*a+b*b) }
(a, b) -> Math.sqrt(a*a+b*b)
n -> n*n
```

Funkcionálny interface

Funkcionálnym interface je interface, ktorý má jedinú metódu

Nepovinná anotácia pre FI je @FunctionalInterface

```java
@FunctionalInterface
interface BinOp { double operation(double a, double b);  }


BinOp plus = (a, b) -> a + b;
System.out.println("3 + 4 = " + plus.operation(3, 4));
```

```
3 + 4 = 7.0
```

```java
BinOp vector = (double a, double b) -> {return Math.sqrt(a*a + b*b); };
BinOp vector = (a,b) -> Math.sqrt(a*a + b*b);
System.out.println("vector(3,4) = " + vector.operation(3,4));


System.out.println("vector(3,4) = "+
  ((BinOp)(a, b) -> Math.sqrt(a*a + b*b)).operation(3,4));
```

```
vector(3,4) = 5.0
```

Example.java

It's cool, just use it !

Java(TM) Platform SE binary

```
jshell>

jshell>

jshell> public class Example {
   ...>     interface BinOp { double operation(double a, double b); }
   ...>     public static void main(String args[]){
   ...>         BinOp plus = (a, b) -> a + b;
   ...>         BinOp vector =
   ...>             (double a, double b) -> {return Math.sqrt(a*a + b*b); };
   ...>         System.out.println("3 + 4 = " + plus.operation(3, 4));
   ...>         System.out.println("vector(3,4) = " +vector.operation(3,4));
   ...>     }
   ...> }
|    modified class Example

jshell> Example.main(null);
3 + 4 = 7.0
vector(3,4) = 5.0

jshell>
```

Funkcionálny interface

(interface a `void` metóda = procedúra)

```java
@FunctionalInterface
interface FunkcionalnyInterface { // koncept funkcie v JDK8
    public void doit(String s);    // jediná "procedúra"
}

                        // metóda foo má procedúru ako argument
public static void foo(FunkcionalnyInterface fi) {
    fi.doit("hello");
}

                        // metóda goo vráti procedúru ako výsledok
public static FunkcionalnyInterface goo() {
    return (String s) -> System.out.println(s + s);
    resp.
    return s -> System.out.println(s + s);
}
```

foo(goo())
"hellohello"

Funkcie.java

Funkcionálny interface
(interface a NEvoid metóda = funkcia)

```java
@FunctionalInterface
interface FunkcionalnyInterface { //String->String
    public String doit(String s);   // jediná "funkcia"
}

                        // metóda foo má funkciu ako argument
public static String foo(FunkcionalnyInterface fi) {
    return fi.doit("hello");
}

                        // metóda goo vráti funkciu ako výsledok
public static FunkcionalnyInterface goo() {
    return (String s)->(s+s);
    resp.
    return s->s+s;
}
```

foo(goo())
"hellohello"

Funkcie.java

Funkcionálny interface
(interface a reálna funkcia)

```java
@FunctionalInterface
interface RealnaFunkcia {
    public double doit(double s);          // funkcia R->R
}
public static RealnaFunkcia iterate(int n, RealnaFunkcia f){
    if (n == 0)
        return d->d;                       // identita
    else {
        RealnaFunkcia rf = iterate(n-1, f); // f^(n-1)
        return d->f.doit(rf.doit(d));
        resp.
        return d->f.doit(iterate(n-1, f).doit(d));
    }
}
```

f^n

```java
RealnaFunkcia rf = iterate(5, (double d)->d*2);
System.out.println(rf.doit(1));
```

Funkcie.java

Existujúce
@FunctionalInterface

miesto doit() apply(), test()

java.util.function.Function<T,R>   <R> apply(<T>) // funkcia T -> R

java.util.function.Predicate<T>   boolean test(<T>) // T ->boolean

Príklady:

```
Function<Double,Double>
    celsius2Fahrenheit = x -> (x*9/5)+32,
    rad2Deg = r -> (r/Math.PI)*180;
Function<String, Integer>
    string2Int = x -> Integer.valueOf(x);
Function<Integer, String>
    int2String = x -> String.valueOf(x);
Predicate<Integer>
    odd = n -> n % 2 > 0;
Predicate<Integer>
    square = n ->
            Math.pow(Math.floor(Math.sqrt(n)),2)==n;
```

```
celsius2Fahrenheit.apply(30.0)  86.0
rad2Deg.apply(Math.PI)          180


string2Int.apply("4")             4


int2String.apply(123))          "123"

odd.test(5));                   true
odd.test(4));                   false

square.test(9));                true
square.test(8));                false
```

Funkcie.java

Existujúce @FunctionalInterface

```
java.lang.Runnable                          void run()
java.util.concurrent.Callable<E>            E call() throws Exception
java.util.Comparator<T>                     int compare(<T> o1, <T> o2)
```

Príklady:

OLD STYLE < Java 8

```java
Runnable r = new Runnable() {

    public void run() {

        // Run Forest, Run !

    };
};
ArrayList<String> l = new ArrayList<>(Arrays.asList(
    "Xenia", "Jan", "Peter", "Zora", "Pavel", "Jana"));

l.sort(new Comparator<String>() {
  @Override
  public int compare(String o1, String o2) {
    return Integer.compare(o1.length(), o2.length());
}});
```

NEW STYLE = Java 8

```java
Runnable r1 = ()->{
    Run Forest, Run !
};

l.sort((o1, o2) ->
    Integer.compare(
        o1.length(),
        o2.length()));
```

Funkcie.java

Comparator

```
String[] pole = { "GULA", "cerven", "zelen", "ZALUD" };
Comparator<String> comp =
    (fst, snd)->Integer.compare(fst.length(), snd.length());
Arrays.sort(pole, comp);
```

```
GULA
zelen
ZALUD
cerven
```

```
Arrays.sort(pole,
    (fst,snd)-> fst.toUpperCase().compareTo(snd.toUpperCase()));
```

```
cerven
GULA
ZALUD
zelen
```

```
class Karta {
    int hodnota;
    String farba;  // konštruktor, gettery, settery...
    ... }
List<Karta> karty = new ArrayList<>(Arrays.asList(
    new Karta(7,"Gula"), new Karta(8,"Zalud"),
    new Karta(9,"Cerven"), new Karta(10,"Zelen")));
```

```
[Gula/7, Zalud/8,
Cerven/9, Zelen/10]
```

MapFilter.java

forEach, map, filter

```java
karty.forEach(k -> k.setFarba("Cerven"));
```

[Cerven/7, Cerven/8, Cerven/9, Cerven/10]

```java
Stream<Karta> vacsieKartyStream =
    karty.stream().filter(k -> k.getHodnota() > 8);
List<Karta> vacsieKarty =
    vacsieKartyStream.collect(Collectors.toList());
```
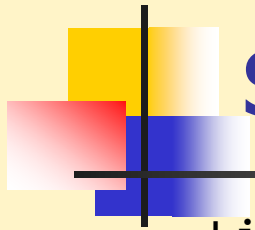
[Cerven/9, Cerven/10]

```java
List<Karta> vacsieKarty2 = karty
    .stream()
    .filter(k -> k.getHodnota() > 8)
    .collect(Collectors.toList());
```

[Cerven/9, Cerven/10]

MapFilter.java

stream()-collect()

```
List<Karta> vacsieKarty3 = karty
    .stream()
    .map(k->new Karta(k.getHodnota()+1,k.getFarba()))
    .filter(k -> k.getHodnota() > 8)
    .collect(Collectors.toList());
```
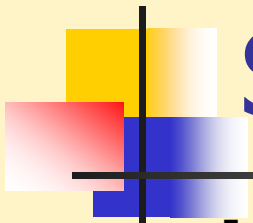
[Cerven/9, Cerven/10, Cerven/11]

```
List<Karta> vacsieKarty4 = karty
    .stream()
    .parallel()
    .filter(k -> k.getHodnota() > 8)
    .sequential()
    .collect(Collectors.toList());
```

[Cerven/9, Cerven/10]

MapFilter.java

Sekvenčný a paralelný stream

IntStream je podtrieda Stream pre hodnoty Integer

```java
// vyrobí stream obsahujuci 0..99
Stream<Integer> stream = IntStream.range(0, 100).boxed();
```

Každá z týchto operácií prebehne `stream` a vyčerpá ho:

- `List<Integer> lst = stream.collect(Collectors.toList());`
- `System.out.println(stream.count());`   100
- `stream.forEach(e -> System.out.println(e+e));`
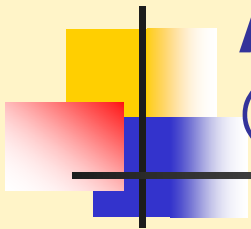- `stream.forEach(System.out::println);`

preto, ak urobíte **dve** na tom istom streame, výsledná chyba je

`Exception: stream has already been operated upon or closed`

```java
// toto už nedostaneme v poradí 0, 1, ...
stream.parallel().forEach(e -> System.out.println(e+e));
```

MapFilter.java

130
132
134
124
126
128
62
...

Aké metódy ma Stream

(Jshell pozná autocompletion)

- Kliknite na TAB

map/filter
(existuje/neexistuje/pre všetky)

```java
lst.
    stream().
    filter(e -> (e % 2 == 0)).
    forEach(System.out::print);      // 0246810121416182022…
lst.
    stream().
    map(e -> e*e).
    forEach(System.out::print);      // 01491625364964 …


∃   lst.stream().anyMatch(e -> (e == 51))       // true
    lst.stream().anyMatch(e -> (e * e == e))    // true
∄   lst.stream().noneMatch(e -> (e > 100))      // true
    lst.stream().noneMatch(e -> (e + e == e))   // false
∀   lst.stream().allMatch(e -> e>0 )            // false
    lst.stream().filter(e -> e>0 ).count()      // 99
```
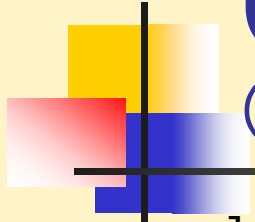
MapFilter.java

Optional
(buď existuje alebo neexistuje)

z predošlého príkladu:
List<Integer> lst = 0, 1, … 99

```
lst.stream().findFirst()              // Optional[0]
lst.stream().findFirst().isPresent() // true
lst.stream().findFirst().get()       // 0

lst.parallelStream().findAny().get() // 56,65,… nejednoznačné

lst.stream().min(Integer::compare).get()          // 0
lst.stream().min(Integer::compare).isPresent()    // true
lst.stream().max(Integer::compare).get()          // 99

lst.stream().map(i->i%10).sorted().forEach(System.out::print);
0000000000111111111122222222223333333333444444444455555555556666
6666667777777777888888888999999999

lst.stream().map(i->i%10).distinct().forEach(System.out::print);
0123456789
```

MapFilter.java

Lenivosť
(laziness)

```
lst.stream().map(e -> { System.out.print(e); return e+e;});

lst.stream().filter(e -> {System.out.print(e);return true;});

lst.stream().map(e -> { System.out.print(e); return e+e;}).
        findFirst().get();                              0

lst.stream().map(e -> { System.out.print(e); return e+e;}).
        collect(Collectors.toList());
```

```
Java(TM) Platform SE binary                                    _  □  x

jshell> lst.stream().map(e -> { System.out.print(e); return e+e;}).
   ...> collect(Collectors.toList());
0123456789101112131415161718192021222324252627282930313233343536373839404142434445464
7484950515253545556575859606162636465666768697071727374757677787980818283848586878889
909192939495969798990$83 ==> [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28
, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70
, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98, 100, 102, 104, 106, 108, 1
10, 112, 114, 116, 118, 120, 122, 124, 126, 128, 130, 132, 134, 136, 138, 140, 142,
144, 146, 148, 150, 152, 154, 156, 158, 160, 162, 164, 166, 168, 170, 172, 174, 176,
 178, 180, 182, 184, 186, 188, 190, 192, 194, 196, 198]
```

MapFilter.java
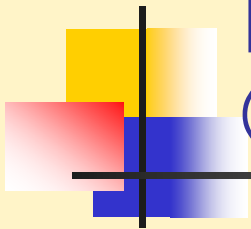
ParallelStream

(komutatívnosť)

```
lst.parallelStream().
    map(e -> e+e).                          // zdvojí čísla
    filter(e -> (e % 3 > 0)).               // nedeliteľné 3
    forEach(e -> System.out.println(e))


lst.parallelStream().
    filter(e -> (e % 3 > 0)).               // nedeliteľné 3
    map(e -> e+e).                          // zdvojí čísla
    forEach(e -> System.out.println(e))


lst.parallelStream().
    map(e -> e+e).                          // zdvojí čísla
    filter(e -> (e % 3 > 0)).               // nedeliteľné 3
    collect(Collectors.toList()).size()  // koľko je výsledok
```

MapFilter.java
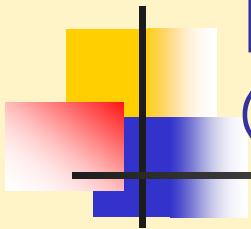
ParallelStream

(skladanie funkcií)

```
lst.parallelStream().
    map(e -> f1(e)).   // čo vieme povedať o kompozícii ?
    map(e -> f2(e)).
    collect(Collectors.toList())


lst.parallelStream().
    map(e -> f2(f1(e))).   // čo vieme povedať o kompozícii ?
    collect(Collectors.toList())


static Integer f1(Integer e) { return e+e; }
static static Integer f2(Integer e) { return 5*e; }
```

MapFilter.java

ParallelStream

(funkcie so side-effect)

Funkcie poznáme *slušné* a iné:

Slušná funkcia (referenčne transparentná) vždy pre rovnaký vstup vráti rovnaký výsledok, t.j. nerobí žiaden side-effect, nepoužíva globálnu premennú, súbor, …
Programovací jazyk je *slušný*, ak v ňom môžete písať len slušné funkcie.

Príklad (neslušný):

```
lst.parallelStream().
    map(e->funWithSideEffect(e)).
    filter(e -> (e % 3 > 0)).
    sorted().
    collect(Collectors.toList());

static Integer globalVariable = 0;
static Integer funWithSideEffect(Integer n) {
    return n+n + (++globalVariable);
}
```

MapFilter.java

Globálne premenné
(sú identifikovaná *smrť*)

V praxi: funkcia sa môže javiť ako slušná, a pri tom ňou nie je ... ☹

```
Java(TM) Platform SE binary

jshell> globalVariable = 0
globalVariable ==> 0

jshell> lst.parallelStream().
   ...> map(e->funWithSideEffect(e)).
   ...>    filter(e -> (e % 3 > 0)).
   ...> sorted().
   ...> collect(Collectors.toList());
$27 ==> [47, 73, 83, 112, 115, 115, 118, 118, 118, 121, 122, 122, 125, 125, 127, 127, 128, 130, 130,
5, 146, 149, 152, 166, 167, 169, 170, 172, 175, 175, 179, 179, 182, 185, 185, 194, 220, 224, 224, 232

jshell> globalVariable = 0
globalVariable ==> 0

jshell> lst.parallelStream().
   ...> map(e->funWithSideEffect(e)).
   ...>    filter(e -> (e % 3 > 0)).
   ...> sorted().
   ...> collect(Collectors.toList());
$29 ==> [34, 38, 41, 44, 46, 47, 47, 49, 50, 50, 53, 53, 100, 103, 107, 110, 113, 115, 118, 119, 119,
51, 152, 154, 154, 155, 157, 158, 158, 161, 161, 163, 163, 166, 166, 169, 169, 184, 187, 190, 202, 20
 241, 241, 242, 244]

jshell>
```
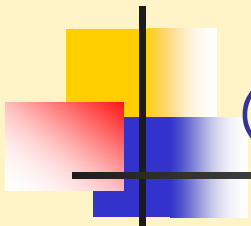
Trochu novej syntaxe

(pripomína Java collections syntax sugar JDK9)

- **Stream obsahujúci pár hodnôť**

```
Stream.of(0,1,2,3,4,5,6,7,9).
    collect(Collectors.toList())
[0, 1, 2, 3, 4, 5, 6, 7, 9]


Stream.of("Palo", "Peter", "Jano", "Jana").
    collect(Collectors.toList())
[Palo, Peter, Jano, Jana]
```

- **Konverzia poľa na Stream**

```
Arrays.stream(new Integer[]{0,1,2,3,4,5,6,7,8,9}).
    collect(Collectors.toList())
[0, 1, 2, 3, 4, 5, 6, 7, 9]
```

- **IntStream a range**

```
IntStream.range(0,100).forEach(e -> System.out.print(e));
01234567891011121314151617181920212223 24…
```

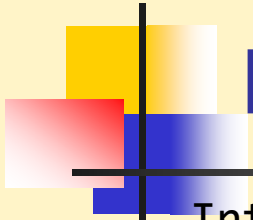MapFilter.java

Collectors
(groupingBy, partitioningBy, reducing)

```
Map<Integer, List<Integer>>map = lst.parallelStream().collect(
    Collectors.groupingBy( e -> (String.valueOf(e).length()) ));
{1=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9], 2=[10, 11, 12, ... , 94, 95, 96, 97, 98, 99]}

map.forEach((len, list) -> System.out.println(len + ", "+ list));
1, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
2, [10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, …

Map<Boolean, List<Integer>>partitions = lst.parallelStream().
    collect(Collectors.partitioningBy( e-> e % 3 == 0) );
{false=[1, 2, 4, 5, 7, 8, 10, ...], true=[0, 3, 6, 9, 12, 15, 18, ...]}

Long count = lst.parallelStream().collect(
    Collectors.reducing(0L, e -> 1L, Long::sum));    // 100
Long sum = lst.parallelStream().collect(
    Collectors.reducing(0L, e -> new Long(e), Long::sum));  // 4950
int sumInt = lst.parallelStream().reduce(0, Integer::sum); //4950
```

mapToObj

```
IntStream.range(0,10).mapToObj(e -> (char)('@'+e)).
    forEach(System.out::print);
```
@ABCDEFGHI

```
IntStream.range(0,10).
    mapToObj(e -> IntStream.range(0, e)).
    forEach(row -> System.out.print(row.count()));
```
0123456789

```
IntStream.range(0,10).
    mapToObj(e -> IntStream.range(0, e)).
    forEach(row -> System.out.println(
        row.boxed().collect(Collectors.toList(
```
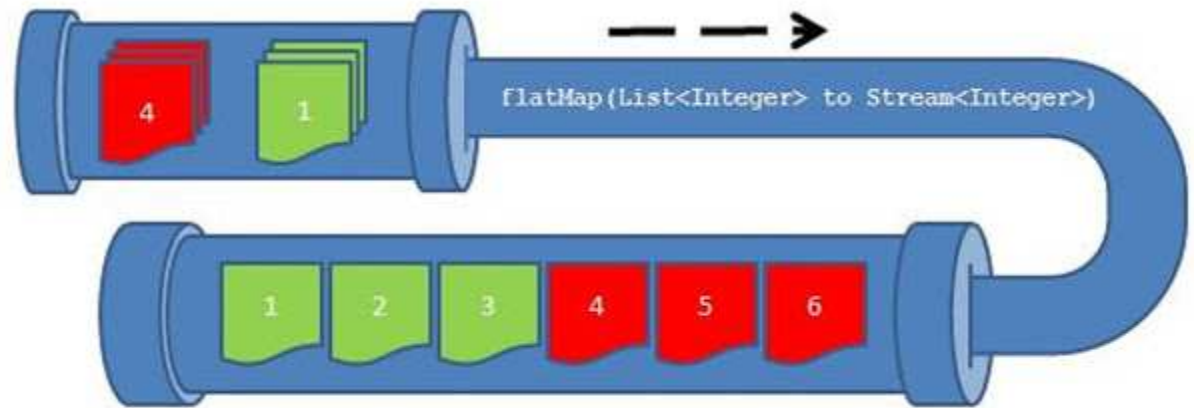
```
[]
[0]
[0, 1]
[0, 1, 2]
[0, 1, 2, 3]
[0, 1, 2, 3, 4]
[0, 1, 2, 3, 4, 5]
[0, 1, 2, 3, 4, 5, 6]
[0, 1, 2, 3, 4, 5, 6, 7]
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

Konverzie IntStream <-> Stream<Integer>

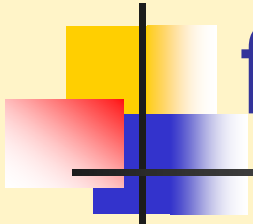- `Stream<Integer> intStream.boxed()`
- `IntStream stream.mapToInt()`

MapFilter.java

flatMap

- The `flatMap` operation

`flatMap(List<Integer> to Stream<Integer>)`

```
List<Integer> together = Stream.of(asList(1, 2, 3), asList(4, 5, 6))
        .flatMap(numbers -> numbers.stream())
        .collect(toList());
assertEquals(asList(1, 2, 3, 4, 5, 6), together);
```

```
List<List<String>> l2 = List.of(
            List.of("Palo", "Jana"),
            List.of("Peter", "Kamil", "Martina"));
```

```
[[Palo, Jana], [Peter, Kamil, Martina]]
l2.stream().flatMap(lst -> lst.stream()).
        collect(Collectors.toList());
[Palo, Jana, Peter, Kamil, Martina]
```

http://www.java67.com/2016/03/how-to-use-flatmap-in-java-8-stream.html

flatMap

```
IntStream.range(0,10).
    flatMap(e -> IntStream.range(0, e)).
    forEach(System.out::print);
00101201230123401234501234560123456701234 5678

IntStream.range(0,10).
    flatMap(e -> IntStream.range(0, e).
                    filter(i->i%2==0)).
    forEach(System.out::print);
0002020240240246024602468
```

Collectors

(rozdeliť stream na úseky, kde platí predikát)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20

```java
static boolean isPrime(Integer n) {
    return IntStream.range(2, 1+(int)Math.floor(Math.sqrt(n)))
                    .allMatch(i -> n % i != 0);
}
```

[[4], [6], [8, 9, 10], [12], [14, 15, 16], [18], [20, 21, 22], ...

```java
int[] splitters = Stream.of( //[-1, 0,1,2,3,5,7,11,13,17,19, 100]
    IntStream.of(-1),
    IntStream.range(0,lst.size()).filter(i->isPrime(lst.get(i))),
    IntStream.of(lst.size()))
  .flatMapToInt(s -> s).toArray();
List<List<Integer>> chunks =
  IntStream.range(0, splitters.length - 1)
  .mapToObj(i -> lst.subList(splitters[i]+1, splitters[i+1]))
  .filter(chunk -> chunk.size() > 0)
  .collect(Collectors.toList());
```

MapFilter.java

FOR FREE
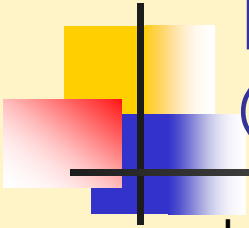
Binárne vektory {0,1}
(klasické riešenie)

```
List<String> binaries(int n) {
    if (n == 0) {
        return Arrays.asList("");
    } else {
        List<String> result = new ArrayList<>();
        for (String s : binaries(n-1)) {
                result.add(s + "0");
                result.add(s + "1");
        }
        return result;
    }
}
binaries(4)
[0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111]
```

Kombinatorika.java

Binárne vektory {0,1}

(streamové riešenie)

Počet = 2^n

```java
Stream<String> binaries1(int n) {
    if (n == 0) {
        return Stream.of("");
    } else {
        return
                binaries1(n-1).
                flatMap(s -> Stream.of(s + "0", s + "1"));
    }
}
Stream<String> binaries1(int n) {
    return (n == 0)?Stream.of(""):
    binaries1(n-1).flatMap(s -> Stream.of(s + "0", s + "1"));
}
binaries1(4).collect(Collectors.toList())
[0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111]
```
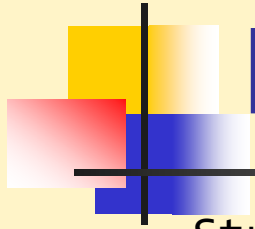
Kombinatorika.java

Permutácie

FOR FREE
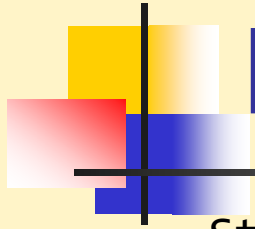
Počet = n!

```java
Stream<String> perms(int n) {
    if (n <= 0) {
        return Stream.of("");
    } else {
        return

                perms(n-1).
                flatMap(s->IntStream.range(0, n).
                    mapToObj(i -> insert(i, n, s)) );

    }
}
String insert(int i, int n, String s) {
    return      s.substring(0,i) +
                String.valueOf(n) +
                s.substring(i, s.length());

}
```

Kombinatorika.java

Kombinácie bez opakovania

<div style="text-align:right">Počet = n nad k</div>
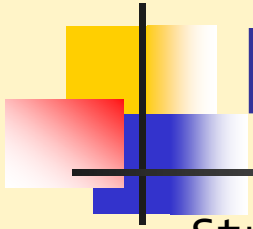
```java
Stream<String> kbo(int k, int n) {
    if (k > n) {
        return Stream.of();
    } else if (k == 0) {
        return Stream.of("");
    } else {
        return Stream.concat(        // zreťazenie dvoch streamov
                kbo(k, n-1),
                kbo(k-1, n-1).map(s -> s + String.valueOf(n-1)));
    }
}
kbo(3,6).collect(Collectors.toList())
[012, 013, 023, 123, 014, 024, 124, 034, 134, 234, 015, 025, 125, 035, 135, 235, 045, 145, 245, 345]
```
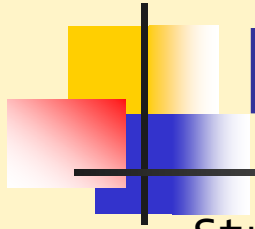
Kombinácie s opakovaním

```java
Stream<String> kso(int k, int n) {
    if (n == 0) {
        return Stream.of();
    } else if (k == 0) {
        return Stream.of("");
    } else {
        return Stream.concat(        // zreťazenie dvoch streamov
                kso(k, n-1),
                kso(k-1, n).map(s -> s + String.valueOf(n-1)));
    }
}
kso(2,6).collect(Collectors.toList())
[01, 11, 02, 12, 22, 03, 13, 23, 33, 04, 14, 24, 34, 44, 05, 15, 25, 35, 45, 55]
```

Počet = (n+k-1) nad k

Kombinácie s opakovaním

```
Stream<String> kso(int k, int n) {
    if (k > n) {
        return Stream.of();
    } else if (k == 0) {
        return Stream.of("");
    } else {
        return Stream.concat(
                kso(k, n-1),
                kso(k-1, n).map(s -> s + String.valueOf(n-1)));
    }
}
kso(2,6).collect(Collectors.toList())
[01, 11, 02, 12, 22, 03, 13, 23, 33, 04, 14, 24, 34, 44, 05, 15, 25, 35, 45, 55]
```

Počet = n^k

FOR FREE

Kombinatorika.java

Variácie bez opakovania

Počet = n(n-1)…(n-k+1)

```
Stream<String> vbo(int k, int n) {
    if (k > n) {
        return Stream.of();
    } else if (k == 0) {
        return Stream.of("");
    } else {
        return Stream.concat(
                vbo(k, n-1),
                vbo(k-1, n-1).
                flatMap(s -> IntStream.range(0, k).
                        mapToObj(i -> insert(i, n-1, s)))));
    }
}
vbo(3,4).collect(Collectors.toList())
```

[210, 120, 102, 201, 021, 012, 310, 130, 103, 301, 031, 013, 320, 230, 203, 302, 032, 023, 321, 231, 213, 312, 132, 123]

Kombinatorika.java

Bonmot

Ak by vám (v 1.semestri) neprezradili priradenie (=) a cyklus (for/while),

tak tu máme spústu šikovných funkcionálnych programátorov…