

# Streams – základní příklady

# Výpis souborů v adresáři

```
import java.io.File;

import static java.util.stream.Collectors.*;

import java.util.stream.Stream;

public class Sample {

    public static void main(String[] args) {

        File dir = new File("c:/");

        File[] children = dir.listFiles();

        if (children != null) {

            for (int i = 0; i < children.length; i++) {

                System.out.print(children[i].getName());

                if (i != children.length - 1)

                    System.out.print(", ");

            }

            System.out.println();

        }

    }

}
```

To samé



```
System.out.println(
    Stream.of(children)
        .map(File::getName)
        .collect(joining(", "))
);
```



Statická metoda v  
`java.util.stream.Collectors`  
Vysvětlení viz.  
následující snímek

# Vysvětlení

```
System.out.println(  
    Stream.of(children)  
        .map(File::getName)  
        .collect(joining(", "))  
);
```

`Stream.of(children)` -> převede 2D pole „array“ na stream

`stream.map(File::getName)` -> převede stream objektů na stream jiných objektů, které jsou definovány tím, co funkce vrací (musí to být funkce objektu, který je ve streamu). V tomto případě se převede stream objektů typu `File` na stream názvů souborů.

`stream.collect()` -> uloží prvky streamu do výsledku (`String`, `List`, `Map`, `Set`, ...)

# Filter

- Můžeme chtít ze streamu získat pouze vybrané záznamy, k tomu slouží `filter()`:

```
System.out.println(
```

```
Stream.of(children)
```

```
    .filter(file -> !file.isDirectory())
```

```
    .map(File::getName)
```

```
    .collect(joining(", ")))
```

```
);
```

Tady jsem použil lambda výraz




Poznámka: Pokud stream obsahuje *n* prvků, pak filter vrátí 0 .. *n* prvků, vyhovující podmínce filtru.

Poznámka: Dovnitř filtru se vkládá **Predicate**, což je funkce, která má metodu `boolean test(T t)`

# Predicate I.

- Nebo:

```
Predicate<File> isFilePred = file -> !file.isDirectory();  
  
if (children != null) {  
    System.out.println(  
        Stream.of(children)  
            .filter(isFilePred)  
            .map(File::getName)  
            .collect(joining(", "))  
    );  
}
```



# Predicate II.

- Nebo:

```
Predicate<File> isDirPred = file -> file.isDirectory();  
  
if (children != null) {  
    System.out.println(  
        Stream.of(children)  
            .filter(isDirPred.negate())  
            .map(File::getName)  
            .collect(joining(", "))  
    );  
}
```

# Predicate III.

- Predicate je také možné řetězit:

```
Predicate<File> isDirPred = file -> file.isDirectory();
```

```
Predicate<File> isHiddenPred = file -> file.isHidden();
```

```
if (children != null) {
```

```
    System.out.println(
```

```
        Stream.of(children)
```

```
            .filter(isDirPred.and(isHiddenPred))
```

```
            .map(File::getName)
```

```
            .collect(joining(", ")))
```

```
    );
```

```
}
```

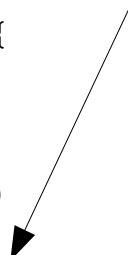
Je možné použít and() nebo or()



# Function I.

- S funkcemi můžete dělat to samé:

```
Function<File, String> getNameFunc = file -> file.getName();  
  
if (children != null) {  
    System.out.println(  
        Stream.of(children)  
            .map(getNameFunc)  
            .collect(joining(", "))  
    );  
}
```



Mimochodem: Funkce můžete volat i standalone:


```
System.out.println(getNameFunc.apply(new File("c:/test")));
```



# Function II.

- Funkce je také možné řetězit:

```
Function<File, String> getNameFunc = file -> file.getName();  
Function<String, String> toLowerCaseFunc = string -> string.toLowerCase();  
  
if (children != null) {  
    System.out.println(  
        Stream.of(children)  
            .map(getNameFunc.andThen(toLowerCaseFunc))  
            .collect(joining(", "))  
    );  
}
```



Na výsledek getNameFunc (čímž je String) se aplikuje funkce toLowerCaseFunc. Protože tato funkce nedělá nic zajímavého, je možné ji nahradit `String::toLowerCase`

# map

- Již jsem zmínil `stream.map()`
- Pár doplnění:
  - Operace `map()` transformuje objekt ze streamu na jiný objekt (může být stejného nebo dokonce i jiného typu, jakého typu bude záležít na tom, jaký typ objektu při transformaci vrátíme).
  - Nemění počet objektů ve streamu.
  - Dvnitř `map()` vstupuje objekt typu **Function**, který má metodu `R apply(T t)`

# Group operace I.

- Když ze streamu potřebujeme získat mapu objektů, pak můžeme lehce použít `groupingBy()`:

```
Map<Boolean, List<File>> map = Stream.of(children)
    .collect(groupingBy(File::isDirectory)
);

System.out.println("Number of Files: " + children.length);

System.out.println("Number of directories: " + map.get(true).size());


System.out.println("Number of files: " + map.get(false).size());
```

- Tato funkce rozdělí stream do skupin podle kritéria (funkce), které se stane klíčem v mapě. V tomto případě získám mapu, ve které budou dva klíče typu boolean: `true`, `false`, na které bude navázán list objektů typu `File` (prakticky tím lehce získám list souborů a adresářů).

# Group operace II.

- Nemusím do výsledné mapy ukládat zdrojové objekty, ale jenom část z nich:

```
Map<Boolean, List<String>> map =  
    Stream.of(children)  
        .collect(groupingBy(File::isDirectory,  
                             mapping(File::getName, toList())))  
    ;
```



```
System.out.println("Number of files: " + children.length);
```

```
System.out.println("Number of directories: " + map.get(true).size());
```

```
System.out.println("Number of files: " + map.get(false).size());
```

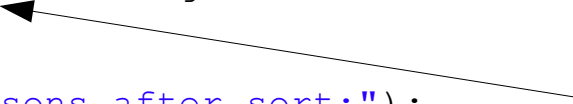
# Utřídění kolekce I.: Třída Person

```
public class Person {  
  
    private String name;  
  
    public Person() { }  
  
    public Person(String name) { this.name = name; }  
  
    public String getName() { return name; }  
  
    public void setName(String name) { this.name = name; }  
  
    @Override  
    public String toString() {  
        return "Person [name=" + name + " ]";  
    }  
}
```

# Utrídění kolekce II.: Test I.

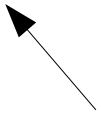
```
public static void main(String[] args) {  
    List<Person> persons = new ArrayList<>();  
    persons.add(new Person("Xavier"));  
    persons.add(new Person("Michal"));  
    persons.add(new Person("Jirka"));  
  
    System.out.println("persons before sort:");  
    persons.stream().forEach(System.out::println);  
    persons.sort(comparing(Person::getName));  
  
    System.out.println("persons after sort:");  
    persons.stream().forEach(System.out::println);  
}
```

Statická funkce z  
java.util.Comparator  
viz. další snímek



# Utřídění kolekce III.

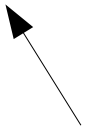
```
persons.sort(comparing(Person::getName));
```



Vrací objekt typu `Comparator`,  
který má další užitečné metody:  
`reversed()` - provede třídění sestupně  
`thenComparing()` - třídění podle více kritérií

... metoda `sort()` by se neměla používat,  
místo toho je lepší:

```
persons.stream().sorted(comparing(Person::getName)).forEach(System.out::println);
```



Nebo je možné kolekci utřídit pomocí `.stream().sorted(Comparator)`  
což má tu výhodu, že tato metoda vrací `Stream`  
a tudíž je na ní možné volat například `forEach()`

# reduce()

- Metoda `reduce()` provede pro každý záznam nějakou operaci a výsledek operace použije v následujícím kroku. Výsledkem může být i jedna hodnota.
- Příklad:


```
Integer[] pole = new Integer[] { 1, 2, 3, 4, 5 };
```

```
int totalSum = Stream.of(pole).reduce(0, (sum, e) -> sum + e);
```

```
System.out.println(totalSum);
```



Počáteční  
hodnota  
totalSum



Operace, která se provede  
pro každý prvek streamu,  
její výsledek se uloží  
do totalSum.

- Hlavička funkce: `reduce(T, BinaryOperator<T>)`
- `BinaryOperator` funkce má metodu `apply(T, U)`



# Specializované reduce() funkce I.: sum()

- Pro sčítání hodnot ve streamu můžeme použít funkci sum(), k jejímu použití ale musíme převést prvky streamu na int, double nebo long:

```
Integer[] pole = new Integer[] { 1, 2, 3, 4, 5 };  
  
int totalSum = Stream.of(pole).mapToInt(Integer::intValue).sum();  
  
System.out.println(totalSum);
```



Funkce: mapToInt,  
mapToDouble, mapToLong

# Specializované reduce() funkce II.: collect()

- Další funkce pro redukci je collect(). Její použití jsme již viděli. Pomocí collect() funkce můžeme výsledek uložit:
  - toList()
  - toSet()
  - toMap()
  - Do stringu pomocí joining()
  - Do mapy pomocí groupingBy()
- Poznámka: Všechny metody jsou statické metody v `java.util.stream.Collectors`.

# Optional

- Pokud potřebujeme ze streamu získat první záznam (například provedeme vyfiltrování záznamů a chceme získat první záznam ve výsledném streamu), pak můžeme použít metodu `findFirst()`.
  - Tato metoda vrací `Optional<T>`
    - `Optional` může nebo nemusí obsahovat výsledek.
    - Je to velice užitečná třída, díky které nemusíme po získání nějakého objektu dělat podmínku jestli to, co jsme získali není null.
    - V Java 9 bude vylepšené:
      - <http://blog.codefx.org/java/dev/java-9-optional/>

# Stream Performance

- „Streams are lazy“
- Operace streamu se dají rozdělit do dvou kategorií:
  - Intermediate (filter, map, ...)
  - Terminal (forEach, collect, ...)
- Stream začne vykonávat intermediate operace až když dojde na terminal operaci, do té doby s daty streamu vůbec nepracuje.
- Při průchodu kolekcí se vždy provádí všechny intermediate operace na každém prvku najednou.
- Streamy jsou vždy pomalejší než když bychom si to naprogramovali ručně, ale zase na druhou stranu jsou přehlednější.

# Další zajímavé metody streamu

- `sorted()`
  - Utřídí elementy streamu
- `distinct()`
  - Vyloučí duplicity ze streamu

# Infinite Stream

- Můžete také vytvořit nekonečný stream:

```
Stream.iterate(100, e -> e + 1).forEach(System.out::println);
```

- Nekonečný stream s předčasným ukončením na základě programově definované podmínky:

```
AtomicInteger i = new AtomicInteger();  
IntStream  
    .generate(() -> i.getAndIncrement())  
    .peek(System.out::println)  
    .allMatch(e -> e < 10_000);
```

- V Java 9 to bude výrazně vylepšené:
  - <http://blog.codefx.org/java/dev/java-9-stream/>

# Náhrada for(int i = 0; ...)

- Když potřebujete vytvořit for cyklus, pak je možné použít IntStream
  - <http://www.deadcoderising.com/2015-05-19-java-8-replace-traditional-for-loops-with-intstreams/>

- Jedno ze zajímavých využití: získat páry objektů v listu:

```
IntStream.range(1, arrayList.size())  
    .mapToObj(i -> new Pair(arrayList.get(i-1), arrayList.get(i)))  
    .forEach(System.out::println);
```