STREAMS

USING JAVA 8 STREAMS

Czym jest Stream?

Nie myląc z I/O Streams, które są przede wszystkim związane z operacjami na plikach, stream w największym skrócie oznacza sekwencję elementów. Można go nazwać "generalizacją" kolekcji.

Jak to działa?

Załóżmy, że mamy listę obiektów typu Person i chcemy otrzymać listę zwierząt należących to osób, które są starsze niż 35 lat. Zanim wprowadzono Javę 8, rozwiązalibyśmy to w taki sposób:

```
List<Pet> pets = new ArrayList<>();
for (Person p : people)  {
   if (p.getAge() > 35) {
pets.add(p.getPet());
   }
}
```

, a korzystając ze streamów w taki:

```
 List<Pet> pets = people.stream()
    . filter((p) -> p.getAge() > 35)
    . map(x->x.getPet())
    . collect(Collectors.toList());
```

Zauważmy, że poszczególne metody streama przyjmują lambdy, które już poznaliśmy. Zamiast bezpośrednio operować na elementach listy, "podajemy" streamowi funkcję, którą ma zastosować.

```
List<Pet> pets = people.stream()
   . filter((p) -> p.getAge() > 35)
   . map(x->x.getPet())
   . collect(Collectors.toList());
```

filter(Predicate<T>)- przyjmuje Predicate<T>
map(Function<T,R>) - przyjmuje Function<T,R>
collect(Collector<T,A,R>) przyjmuje Collector<T,A,R>

.map(Function<T,R>

.map używamy po to, by "mapować" obiekty, czyli przekształcać jeden obiekt w inny.

```
List<String> letters = Arrays.asList("a", "b", "c", "d");
List<String> collect = letters.stream().map(String::toUpperCase).collect(Collectors.toList());
System.out.println(collect); //[A, B, C, D]
```

W tym przykładzie stworzyliśmy nową listę, w której Stringi zostały zmapowane na Stringi pisane wielką literą.

.filter(Predicate<T>)

Jak sama nazwa wskazuje, filter() będziemy używać do "filtrowania" naszego streama, czyli wybierania tylko pożądanych elementów.

```
List<Person> persons = …
Stream<Person> personsOver18 = persons.stream().filter(p -> p.getAge() > 18);
```

W tym przykładzie otrzymaliśmy stream z osobami powyżej 18 lat.

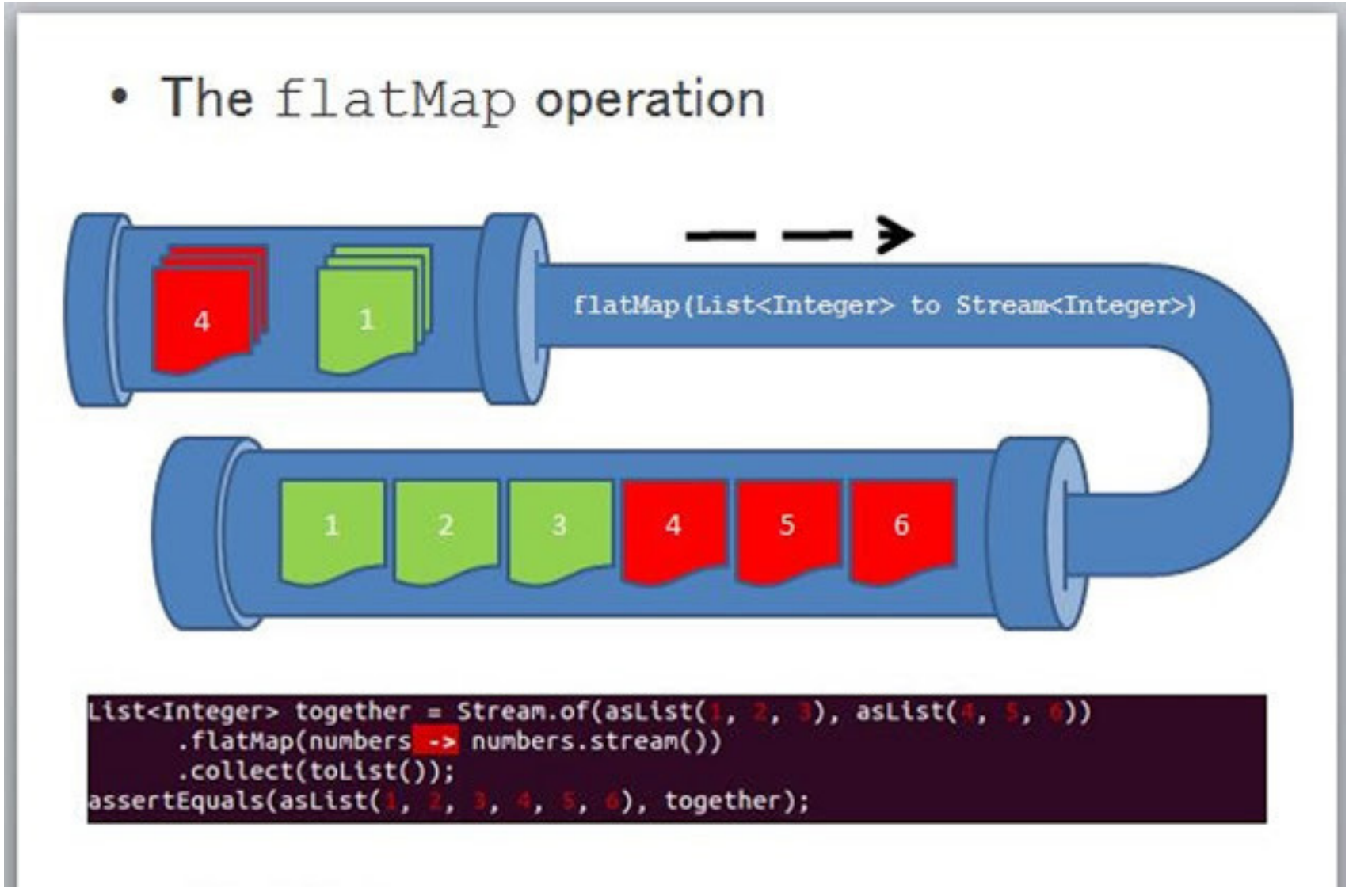
.flatMap(Function<T,R>

Ta metoda używana jest do "spłaszczania" streama streamów (to nie pomyłka).

```
List<String> pinkFloyd = Arrays.asList("Gilmour", "Waters", "Wright","Mason","Barrett");
    List<String> ledZeppelin = Arrays.asList("Page", "Plant", "Jones","Bonham");
    List<List<String>> woodstockBands = new ArrayList<>();
    woodstockBands.add(pinkFloyd);
    woodstockBands.add(ledZeppelin);
    List<String> lineUp = woodstockBands.stream()
                                .flatMap(x->x.stream())
                                .collect(Collectors.toList());
```

W tym przykładzie "spłaszczyliśmy" listę list na jedną listę. Wynikiem jest lista Stringów zawierająca wszystkie Stringi z listy źródłowej

Całkiem dobry obrazek, który tłumaczy działanie flatMap:



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

collect(Collector<T,A,R>)

Tej metody można użyć do "zebrania" streama w kolekcję, która nas interesuje. Stream trzeba w jakiś sposób zamknąć. Najszczęściej będziemy oczekiwać kolekcji, ale mogą to być pojedyncze obiekty. Poniżej przykłady:

```
List<String> result = givenList.stream()
 .collect(toList());
Set<String> result = givenList.stream()
 .collect(toSet());
List<String> result = givenList.stream()
 .collect(toCollection(LinkedList::new))
String result = givenList.stream()
 .collect(joining());
String result = givenList.stream()
 .collect(joining(" "));
Long result = givenList.stream()
 .collect(counting());
```

Do roboty!