B.Sc. In Software Development. Year 3. Applications Programming. Streams and Filters
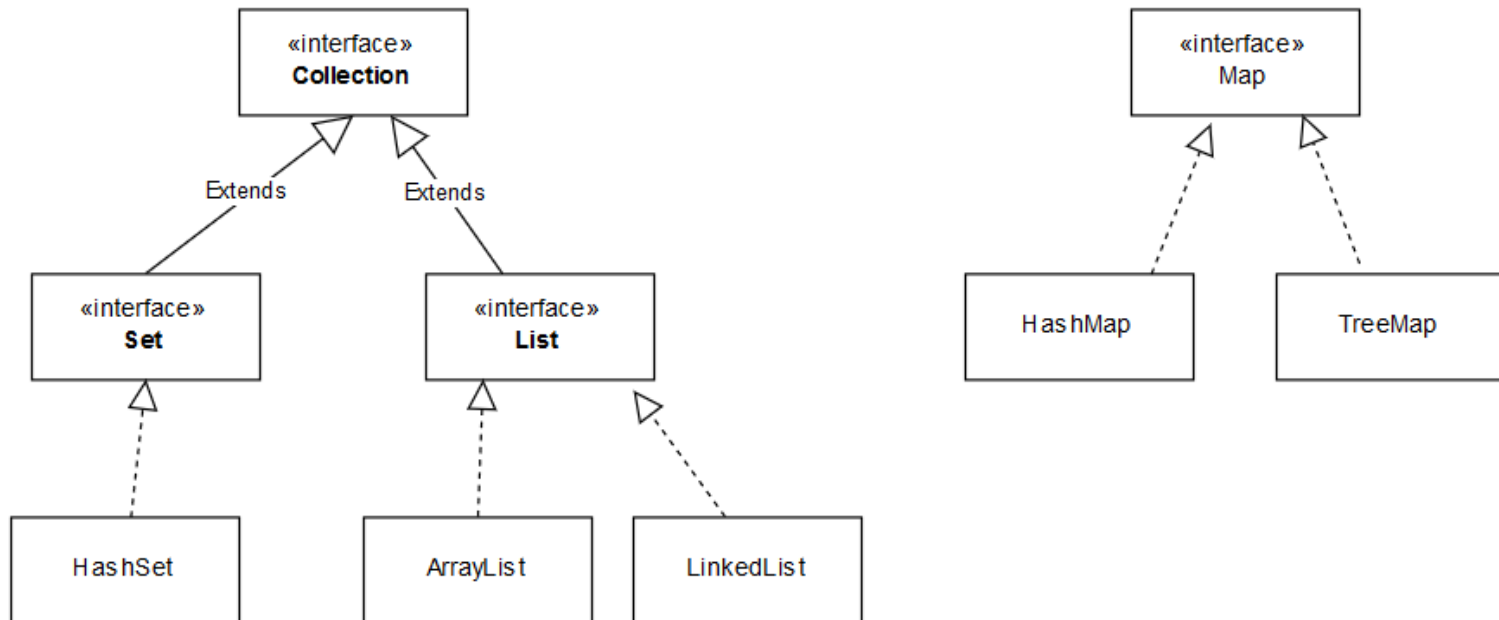
Introduction

- Recall: A collection is an in-memory data structure to hold values.

- Unlike arrays, collections aren't part of the language itself.

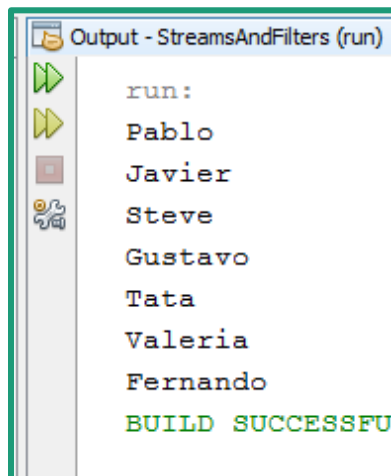- Collections are classes that are available from the Java API.

Introduction

- Java Streams do not store data.

- Streams operate on the source data structure (collection/array) and produce pipelined data that we can use and perform specific operations.

- You can create a stream from the list and filter it based on a condition.

Introduction

- Streams may seem similar to a collection.

  - They're not.

- Differences include:

  1. Streams do not store data.

  2. Streams don't mutate their source.

  3. Stream operations are lazy.

Simple Example – Without Streams

```java
        List<String> names = new ArrayList<>();
        names.add("Pablo");
        names.add("Javier");
        names.add("Steve");
        names.add("Gustavo");
        names.add("Tata");
        names.add("Valeria");
        names.add("Fernando");

        for (String str : names) {
            System.out.println(str);
        }
```

Output - StreamsAndFilters (run)

```
    run:
    Pablo
    Javier
    Steve
    Gustavo
    Tata
    Valeria
    Fernando
    BUILD SUCCESSFU
```
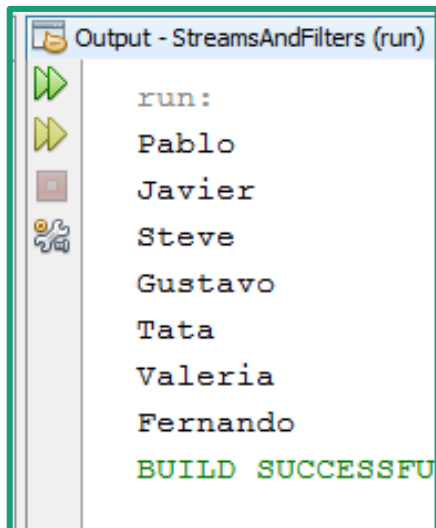
Simple Example – Using Streams

```java
List<String> names = new ArrayList<>();
names.add("Pablo");
names.add("Javier");
names.add("Steve");
names.add("Gustavo");
names.add("Tata");
names.add("Valeria");
names.add("Fernando");

names.forEach(System.out::println);
//alternatively
//names.stream().forEach(s -> System.out.println(s));
```

```
Output - StreamsAndFilters (run)

run:
Pablo
Javier
Steve
Gustavo
Tata
Valeria
Fernando
BUILD SUCCESSFU
```

Typical Workflow When Working With Streams

1. **Create** a stream.

2. Specify intermediate operations for **transforming** the initial stream into others, possibly into multiple steps.

3. Apply a **terminal operation** to produce a result.

Characteristics of a Stream

- [Streams](#) represents a sequence of objects from a source, which supports aggregate operations.

  - **Source** − Stream takes Collections, Arrays, or I/O resources as input source.

  - **Aggregate operations** − Stream supports aggregate operations like filter, map, limit, reduce, find, match etc.

  - **Pipelining** − Most of the stream operations return streams themselves so their result can be pipelined.
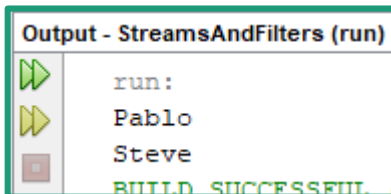
Creating a Stream

- With Java 8, the [Collection](#) interface has two methods to generate a Stream.

    - `stream()` – Returns a serial/sequential stream. With these streams process elements one element after the next.

    - `parallelStream()` – Returns a parallel stream. Parallel streams, in contrast, can take full advantage of multicore processors by breaking its elements into two or more smaller streams, performing operations on them, and then recombining the separate streams to create the final result

Transforming a Stream –filtering

- A stream transformation produces a stream whose elements are derived from another.

- A simple transformation using `filter`.
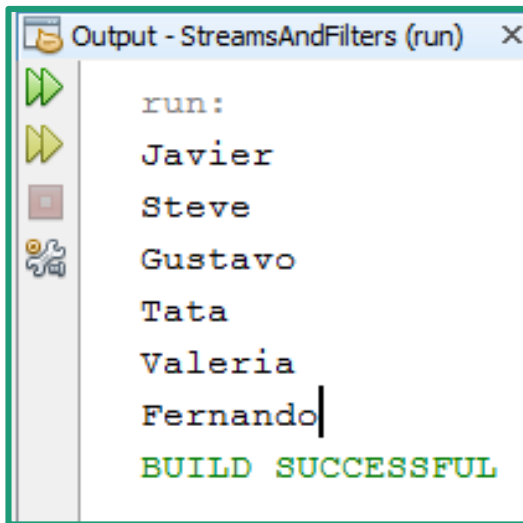
```
13          List<String> names = new ArrayList<>();
14          names.add("Pablo");
15          names.add("Javier");
16          names.add("Steve");
17          names.add("Gustavo");
18          names.add("Tata");
19          names.add("Valeria");
20          names.add("Fernando");
21
22          Stream<String> filteredList = names.stream().filter(s -> s.length() ==5);
23          filteredList.forEach(System.out::println);
24
25          //alternatively
26          //names.stream().filter(s -> s.length() ==5).forEach(System.out::println);
```

```
Output - StreamsAndFilters (run)
  run:
  Pablo
  Steve
  BUILD SUCCESSFUL
```

Transforming a Stream –filtering

```java
11          List<String> names = new ArrayList<>();
12          names.add("Pablo");
13          names.add("Javier");
14          names.add("Steve");
15          names.add("Gustavo");
16          names.add("Tata");
17          names.add("Valeria");
18          names.add("Fernando");
19
20          List<String> filteredNames = names.stream().filter(string -> !string.startsWith("Pa")).collect(Collectors.toList());
21
22          filteredNames.forEach(System.out::println);
```
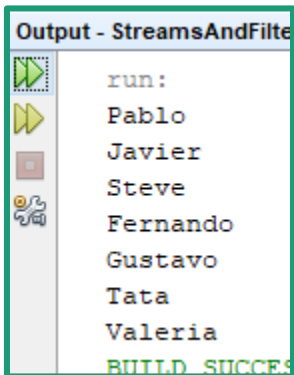
Output - StreamsAndFilters (run) ✕

```
run:
Javier
Steve
Gustavo
Tata
Valeria
Fernando
BUILD SUCCESSFUL
```
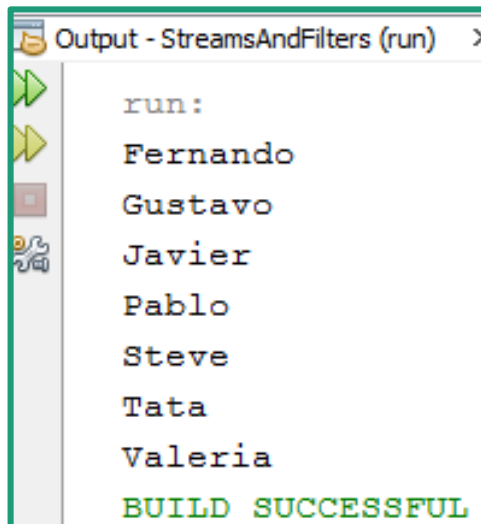
Transforming a Stream – using distinct

```
13          List<String> names = new ArrayList<>();
14          names.add("Pablo");
15          names.add("Javier");
16          names.add("Steve");
17          names.add("Fernando");
18          names.add("Gustavo");
19          names.add("Tata");
20          names.add("Valeria");
21          names.add("Fernando");
22          names.add("Valeria");
23
24          Stream<String> distinctList = names.stream().distinct();
25          distinctList.forEach(System.out::println);
```

Output - StreamsAndFilte

```
run:
Pablo
Javier
Steve
Fernando
Gustavo
Tata
Valeria
BUILD SUCCES
```

Transforming a Stream – sorting

```java
10          List<String> names = new ArrayList<>();
11          names.add("Pablo");
12          names.add("Javier");
13          names.add("Steve");
14          names.add("Gustavo");
15          names.add("Tata");
16          names.add("Valeria");
17          names.add("Fernando");
18
19          names.stream().sorted().forEach(System.out::println);
```
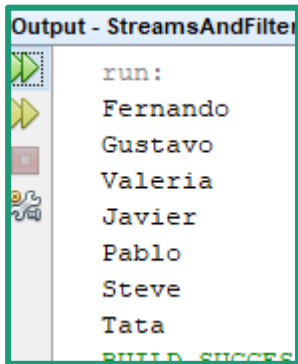
```
Output - StreamsAndFilters (run)        >

    run:
    Fernando
    Gustavo
    Javier
    Pablo
    Steve
    Tata
    Valeria
    BUILD SUCCESSFUL
```

Transforming a Stream – sorting

```
13          List<String> names = new ArrayList<>();
14          names.add("Pablo");
15          names.add("Javier");
16          names.add("Steve");
17          names.add("Gustavo");
18          names.add("Tata");
19          names.add("Valeria");
20          names.add("Fernando");
21
22          Stream<String> revStream = names.stream().sorted(Comparator.comparing(String::length).reversed());
23          revStream.forEach(System.out::println);
24          //alternatively
25          //names.stream().sorted(Comparator.comparing(String::length).reversed()).forEach(System.out::println);
```
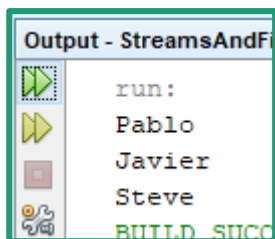
Output - StreamsAndFilter
```
run:
Fernando
Gustavo
Valeria
Javier
Pablo
Steve
Tata
BUILD SUCCES
```

Transforming a Stream – limiting

```java
13          List<String> names = new ArrayList<>();
14          names.add("Pablo");
15          names.add("Javier");
16          names.add("Steve");
17          names.add("Gustavo");
18          names.add("Tata");
19          names.add("Valeria");
20          names.add("Fernando");
21
22          Stream<String> str = names.stream().limit(3);
23          str.forEach(System.out::println);
24
25          //alternatively
26          //names.stream().limit(3).forEach(System.out::println);
```

```
Output - StreamsAndFi
  run:
  Pablo
  Javier
  Steve
  BUILD SUCC
```

Reductions

- Essentially getting data from streams.

- They are terminal operations.

  - They reduce the stream to a non-stream value that can be used in your program. E.G.

```java
13         List<String> names = new ArrayList<>();
14         names.add("Pablo");
15         names.add("Javier");
16         names.add("Steve");
17         names.add("Gustavo");
18         names.add("Tata");
19         names.add("Valeria");
20         names.add("Fernando");
21
22         long totalCount = names.stream().count();
23         System.out.println("Total count " + totalCount);
24
25         long filteredCount = names.stream().filter(aName -> aName.length() > 6).count();
26         System.out.println("Filtered Count " + filteredCount);
```
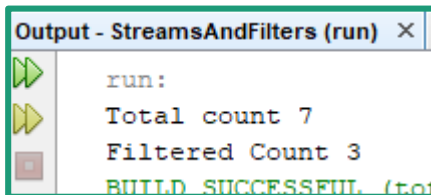
```
Output - StreamsAndFilters (run)  ✕
    run:
    Total count 7
    Filtered Count 3
    BUILD SUCCESSFUL (to
```

Reductions – max and min

```java
List<String> names = new ArrayList<>();
names.add("Pablo");
names.add("Javier");
names.add("Steve");
names.add("Gustavo");
names.add("Tata");
names.add("Valeria");
names.add("Fernando");

String minName = names.stream().min(Comparator.comparing(String::valueOf)).get();
System.out.println(minName);

String maxName = names.stream().max(Comparator.comparing(String::valueOf)).get();
System.out.println(maxName);

List<Integer> numList = Arrays.asList(-9, -18, 0, 25, 4, 8, 85, 7, 5);

int minNum = numList.stream().min(Comparator.comparing(Integer::valueOf)).get();
System.out.println(minNum);

int maxNum = numList.stream().max(Comparator.comparing(Integer::valueOf)).get();
System.out.println(maxNum);
```

Output - StreamsAndF
```
run:
Fernando
Valeria
-18
85
BUILD SUCC
```

Reductions – using Optional
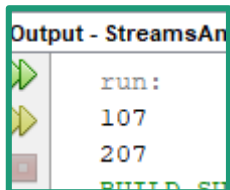
```java
13        List<String> names = new ArrayList<>();
14        names.add("Pablo");
15        names.add("Javier");
16        names.add("Steve");
17        names.add("Gustavo");
18        names.add("Tata");
19        names.add("Valeria");
20        names.add("Fernando");
21
22        Optional<String> minName = names.stream().min(Comparator.comparing(String::valueOf));
23
24        if (minName.isPresent())
25            System.out.println(minName.get());
26        else
27            System.out.println("not present");
28
29        Optional<String> maxName = names.stream().max(Comparator.comparing(String::valueOf));
30
31        System.out.println(maxName);
32
33        List<Integer> numList = Arrays.asList();
34
35        Optional<Integer> minNum = numList.stream().min(Comparator.comparing(Integer::valueOf));
36        System.out.println(minNum.orElse(-1));
37
38        Optional<Integer> maxNum = numList.stream().max(Comparator.comparing(Integer::valueOf));
39        System.out.println(maxNum);
```

```
Output - StreamsAndFilters (run)  ✕

    run:
    Fernando
    Optional[Valeria]
    -1
    Optional.empty
    BUILD SUCCESSFUL (t
```

Reductions – reduce

```java
List<Integer> numList = Arrays.asList(-9, -18, 0, 25, 4, 8, 85, 7, 5);

int total = numList.stream().reduce((x, y) -> x + y).get();
//alternatively
//int total = numList.stream().reduce(Integer::sum).get();
System.out.println(total);

//Set start value. Result will be start value + sum of array.
int startValue = 100;
int sum = numList.stream().reduce(startValue, (x, y) -> x + y);
//alternatively
//int sum = numList.stream().reduce(startValue, Integer::sum);
System.out.println(sum);
```
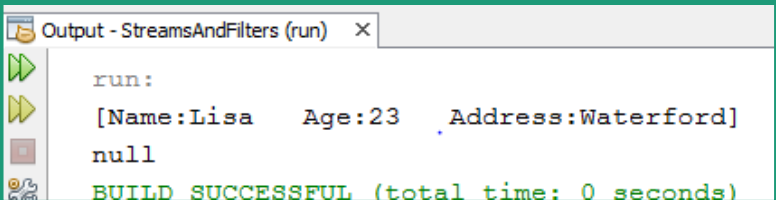
```
Output - StreamsAn
  run:
  107
  207
  BUILD SU
```

Reductions – findAny and OrElse

```java
4    public class Person {
5        private String id;
6        private String name;
7        private int age;
8        private String address;
9
10       public Person(String id, String name, int age, String address) {
11           this.id = id;
12           this.name = name;
13           this.age = age;
14           this.address = address;
15       }
16
17       public String getId() {
```
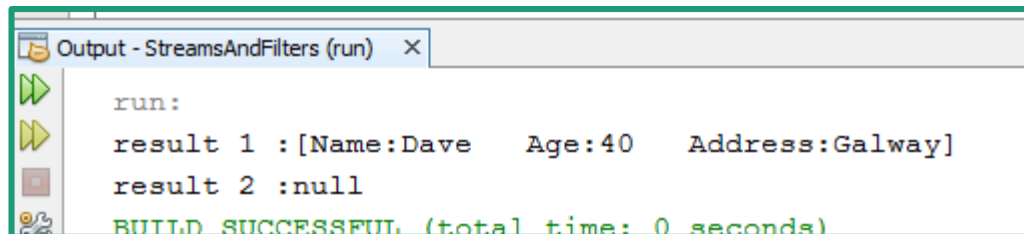
```java
19           List<Person> persons = Arrays.asList(
20               new Person("D94086", "Dave", 40, "Galway"),
21               new Person("L87454", "Lisa", 23 , "Waterford"),
22               new Person("T93677", "Trevor", 60, "Dublin")
23           );
24
25           Person result1 = persons.stream()
26               .filter(x -> "Lisa".equals(x.getName()))
27               .findAny()
28               .orElse(null);
29
30           System.out.println(result1);
31
32           Person result2 = persons.stream()
33               .filter(x -> "Alan".equals(x.getName()))
34               .findAny()
35               .orElse(null);
36
37           System.out.println(result2);
```

Output - StreamsAndFilters (run) ✕

```
run:
[Name:Lisa    Age:23    Address:Waterford]
null
BUILD SUCCESSFUL (total time: 0 seconds)
```

Reductions – multiple conditions

```java
15        List<Person> persons = Arrays.asList(
16                new Person("D94086", "Dave", 40, "Galway"),
17                new Person("L87454", "Lisa", 23 , "Waterford"),
18                new Person("T93677", "Trevor", 60, "Dublin")
19        );
20
21
22        Person result1 = persons.stream()
23                .filter((p) -> "dave".equalsIgnoreCase(p.getName()) && 40 == p.getAge())
24                .findAny()
25                .orElse(null);
26
27        System.out.println("result 1 :" + result1);
28
29        //or like this
30        Person result2 = persons.stream()
31                .filter(p -> {
32                    if ("Alan".equals(p.getName()) && 21 == p.getAge()) {
33                        return true;
34                    }
35                    return false;
36                }).findAny()
37                .orElse(null);
38
39        System.out.println("result 2 :" + result2);
```
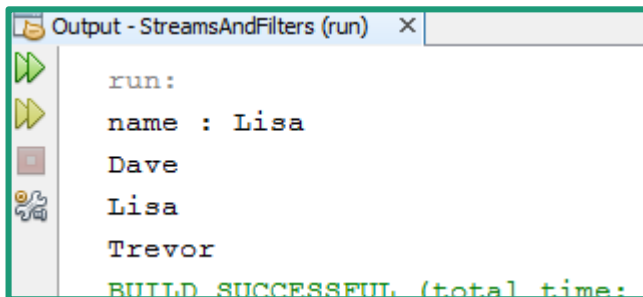
```
Output - StreamsAndFilters (run)  ✕

    run:
    result 1 :[Name:Dave    Age:40    Address:Galway]
    result 2 :null
    BUILD SUCCESSFUL (total time: 0 seconds)
```

Reductions – map and collect

```
16          List<Person> persons = Arrays.asList(
17                  new Person("D94086", "Dave", 40, "Galway"),
18                  new Person("L87454", "Lisa", 23 , "Waterford"),
19                  new Person("T93677", "Trevor", 60, "Dublin")
20          );
21
22
23      String name = persons.stream()
24              .filter(x -> "lisa".equalsIgnoreCase(x.getName()))
25              .map(Person::getName)
26              .findAny()
27              .orElse("Nothing Found");
28
29      System.out.println("name : " + name);
30
31      List<String> collect = persons.stream()
32              .map(Person::getName)
33              .collect(Collectors.toList());
34
35      collect.forEach(System.out::println);
```

Output - StreamsAndFilters (run) ✕

```
run:
name : Lisa
Dave
Lisa
Trevor
BUILD SUCCESSFUL (total time:
```

Collecting Results

- When you are done with a stream, you often want to look at the results:

```
stream.forEach(System.out.println);
```

- More often than not, you will want to collect the stream in a data structure.

```java
13          List<String> names = new ArrayList<>();
14          names.add("Pablo");
15          names.add("Javier");
16          names.add("Steve");
17          names.add("Gustavo");
18          names.add("Tata");
19          names.add("Valeria");
20          names.add("Fernando");
21
22          Object[] filteredArray = names.stream().filter(aName -> aName.length() > 6).toArray();
23          System.out.println(Arrays.toString(filteredArray));
24
25          String[] result = names.stream().filter(aName -> aName.length() > 6).toArray(String[]::new);
26          System.out.println(Arrays.toString(result));
```
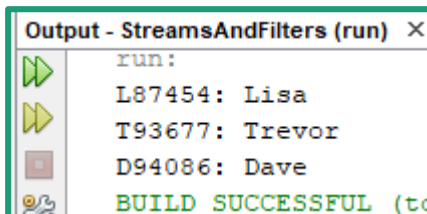
```
Output - StreamsAndFilters (run)  ✕
run:
[Gustavo, Valeria, Fernando]
[Gustavo, Valeria, Fernando]
BUILD SUCCESSFUL (total time:
```

Collecting Results into Maps

- What if you have a Stream<Person> and you want to collect the elements into a map.

  - You can then lookup people by their ID.

```
20          List<Person> persons = Arrays.asList(
21                  new Person("D94086", "Dave", 40, "Galway"),
22                  new Person("L87454", "Lisa", 23 , "Waterford"),
23                  new Person("T93677", "Trevor", 60, "Dublin")
24          );
25
26          Map<String, String> idToName = persons.stream().collect(Collectors.toMap(Person::getId, Person::getName));
27
28          for (Map.Entry<String, String> entry : idToName.entrySet()) {
29              System.out.println(entry.getKey() + ": " + entry.getValue());
30          }
```
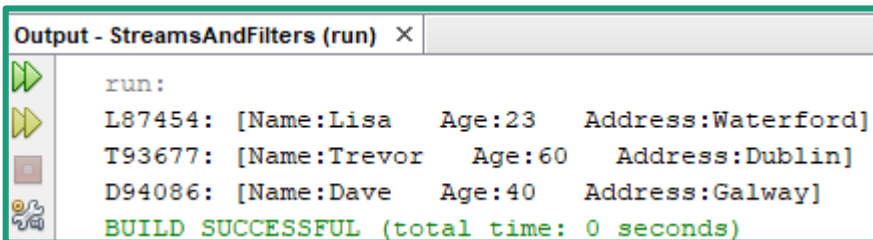
```
Output - StreamsAndFilters (run)  ✕
  run:
  L87454: Lisa
  T93677: Trevor
  D94086: Dave
  BUILD SUCCESSFUL (to
```

Collecting Results into Maps

- What if you want the actual Person objects to be the values and the ID's to be the keys?

```java
21        List<Person> persons = Arrays.asList(
22                new Person("D94086", "Dave", 40, "Galway"),
23                new Person("L87454", "Lisa", 23 , "Waterford"),
24                new Person("T93677", "Trevor", 60, "Dublin")
25        );
26
27
28        Map<String, Person> idToObj = persons.stream().collect(Collectors.toMap(Person::getId, Function.identity()));
29
       for (Map.Entry<String, Person> entry : idToObj.entrySet()) {
31            System.out.println(entry.getKey() + ": " + entry.getValue());
32        }
```

```
Output - StreamsAndFilters (run) ×

    run:
    L87454: [Name:Lisa    Age:23    Address:Waterford]
    T93677: [Name:Trevor   Age:60    Address:Dublin]
    D94086: [Name:Dave    Age:40    Address:Galway]
    BUILD SUCCESSFUL (total time: 0 seconds)
```

Primitive Type Streams

- Thus far all integers have been collected in a `Stream<Integer>`.

  - This is inefficient.

  - Same is true for all primitive types.

- The stream library has specialised types IntStream, LongStream and DoubleStream.

  - They store primitive values directly.

Primitive Type Streams - Examples

```
11          int arr1[] = IntStream.of(1,2,3,4,5,6).toArray();
12          System.out.println("arr1 " + Arrays.toString(arr1));
13
14          int arr2[] = IntStream.range(0, 10).toArray(); //upperbound is excluded
15          System.out.println("arr2 " + Arrays.toString(arr2));
16
17          int arr3[] = IntStream.rangeClosed(0, 10).toArray(); //upperbound is excluded
18          System.out.println("arr3 " + Arrays.toString(arr3));
19
20          List<String> names = new ArrayList<>();
21          names.add("Pablo");
22          names.add("Javier");
23          names.add("Steve");
24          names.add("Gustavo");
25          names.add("Tata");
26          names.add("Valeria");
27          names.add("Fernando");
28
29          //process the lengths of each string in the list as ints
30          int arr4[] = names.stream().mapToInt(String::length).toArray();
31          System.out.println("arr4 " + Arrays.toString(arr4));
32
33          int arr5[] = new Random().ints(10, 0, 1000).toArray();
34          System.out.println("arr5 " + Arrays.toString(arr5));
```

```
Output - StreamsAndFilters (run) ×

    run:
    arr1 [1, 2, 3, 4, 5, 6]
    arr2 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    arr3 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    arr4 [5, 6, 5, 7, 4, 7, 8]
    arr5 [856, 699, 619, 72, 615, 185, 28, 582, 343, 321]
    BUILD SUCCESSFUL (total time: 0 seconds)
```

Parallel Streams

- Stream make it easy to parallelise bulk operations.

  - Process is automatic once you have a parallel stream.

- When stream operations run in parallel the intent is that the same result is returned as if they had run serially.

- There is a substantial overhead to using parallel streams that will only pay off for very large data sets.

References

Cay S. Horstmann (2018) *Core Java SE 9 For the Impatient. 2/E*. ISBN-13 978-0-13-469472-6 (Link)

Murach J., and Urban M. (2014) *Murach's Beginning Java with NetBeans*, Mike Murach and Associates, Inc. (Link)

https://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/

https://www.baeldung.com/java-8-streams