Streams

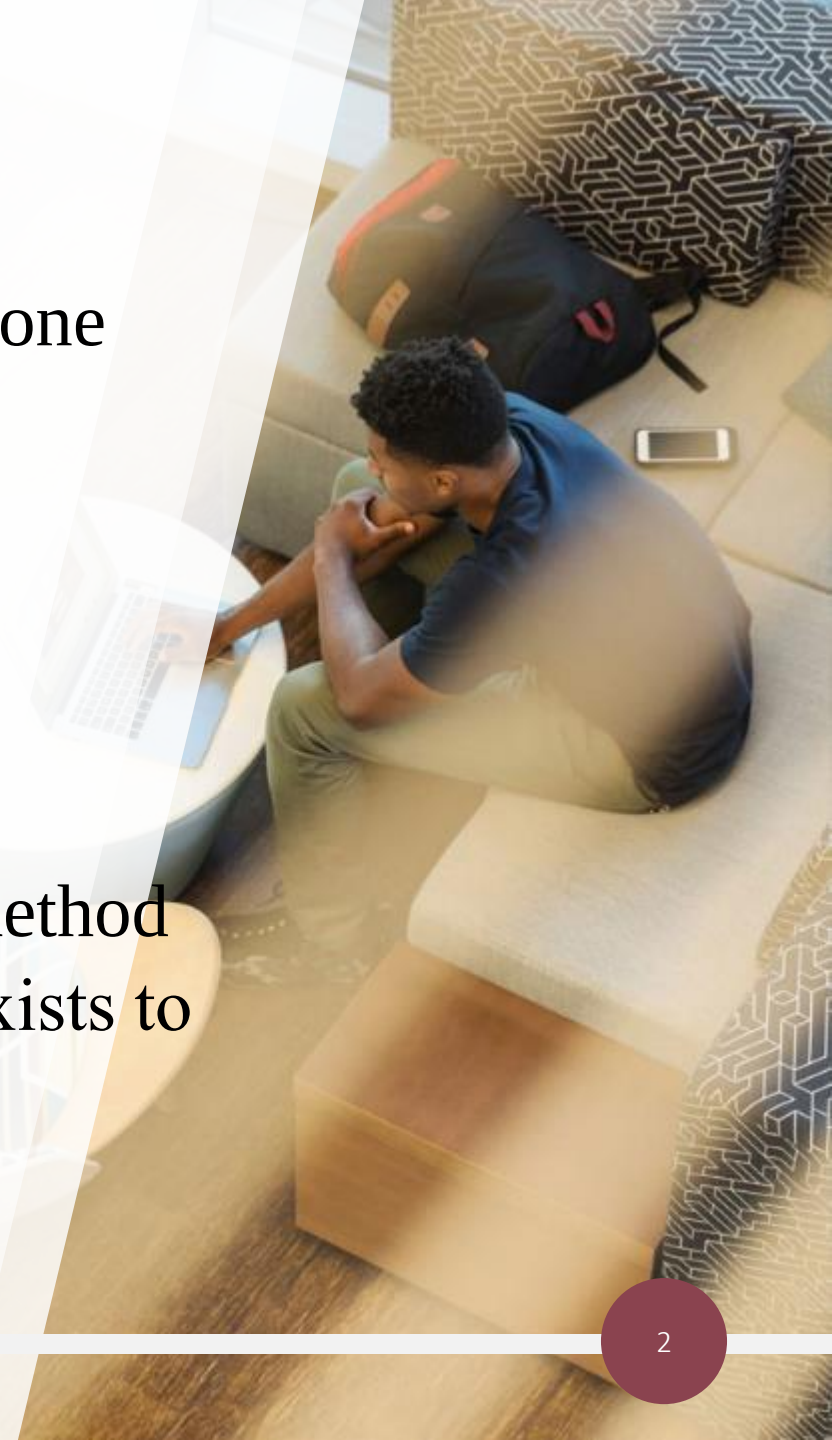
Terminal Operations
collect() using API Collectors

Terminal Operations
collect() – using API-defined Collectors

- Now we will look at the other version *collect()* – the one that accepts pre-defined collectors from the API.

- We access these collectors via static methods on the *Collectors* interface.

- It is important to pass the *Collector* to the *collect()* method
  - a *Collector* does not do anything on it's own. It exists to help collect elements.



Terminal Operations
collect(Collector)

```
String s = Stream.of("cake", "biscuits", "apple tart")
                 .collect(Collectors.joining(", "));
System.out.println(s);  // cake, biscuits, apple tart
```

```
Double avg = Stream.of("cake", "biscuits", "apple tart")
                   // averagingInt(ToIntFunction) functional method is:
                   //      int applyAsInt(T value);
                   .collect(Collectors.averagingInt(s -> s.length()));
System.out.println(avg);  // 7.333333333333333
```

Terminal Operations
Collectors.toMap()

- Collecting into Maps:
  - ➢ two Functions required: the first function tells the collector how to create the **key**; the second function tells the collector how to create the **value.**

```
// We want a map: dessert name -> number of characters in dessert name
// Output:
//   {biscuits=8, cake=4, apple tart=10}
Map<String, Integer> map =
        Stream.of("cake", "biscuits", "apple tart")
        .collect(
            Collectors.toMap(s -> s,          // Function for the key
                             s -> s.length())// Function for the value
        );
System.out.println(map);
```

Terminal Operations

Collectors.toMap() – opposite of previous example


```
// We want a map: number of characters in dessert name -> dessert name
// However, 2 of the desserts have the same length (cake and tart) and as
// length is our key and we can't have duplicate keys, this leads to an
// exception as Java does not know what to do...
//    IllegalStateException: Duplicate key 4 (attempted merging values cake and tart)
// To get around this, we can supply a merge function, whereby we append the
// colliding keys values together.
Map<Integer, String> map =
        Stream.of("cake", "biscuits", "tart")
        .collect(
            Collectors.toMap(s -> s.length(),// key is the length
                             s -> s,         // value is the String
                             (s1, s2) -> s1 + "," + s2)// Merge function - what to
                                                       // do if we have duplicate keys
                                                       //   - append the values
        );
System.out.println(map);// {4=cake,tart, 8=biscuits}
```

Terminal Operations
Collectors.toMap()

```
// The maps returned are HashMaps but this is not guaranteed. What if we wanted
// a TreeMap implementation so our keys would be sorted. The last argument
// caters for this.
TreeMap<String, Integer> map =
        Stream.of("cake", "biscuits", "apple tart", "cake")
        .collect(
            Collectors.toMap(s -> s,            // key is the String
                             s -> s.length(),   // value is the length of the String
                            (len1, len2) -> len1 +len2, // what to do if we have
                                                        // duplicate keys
                                                        //    - add the *values*
                            () -> new TreeMap<>() ));// TreeMap::new works
System.out.println(map);// {apple tart=10, biscuits=8, cake=8} Note: cake maps to 8
System.out.println(map.getClass());// class java.util.TreeMap
```

Terminal Operations
Collectors.groupingBy()

- *groupingBy()* tells *collect()* to group all of the elements into a *Map*.

- *groupingBy()* takes a *Function* which determines the keys in the *Map*.

- Each value is a *List* of all entries that match that key. The *List* is a default, which can be changed.

Terminal Operations
Collectors.groupingBy()

```
Stream<String> names = Stream.of("Joe", "Tom", "Tom", "Alan", "Peter");
Map<Integer, List<String>> map =
        names.collect(
            // passing in a Function that determines the
            // key in the Map
            Collectors.groupingBy(String::length) // s -> s.length()
        );
System.out.println(map);// {3=[Joe, Tom, Tom], 4=[Alan], 5=[Peter]}
```

Terminal Operations
Collectors.groupingBy()

- What if we wanted a *Set* instead of a *List* as the value in the map (to remove the duplication of "Tom") ?

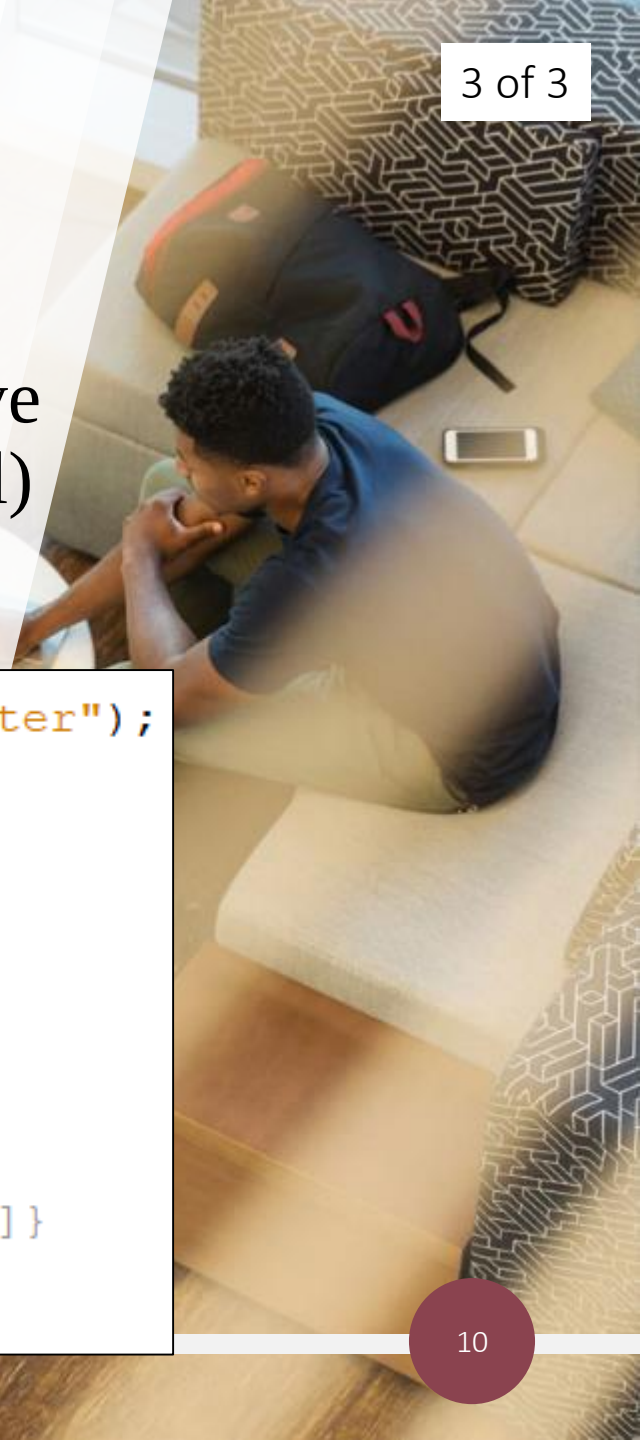
- *groupingBy()* is overloaded to allow us to pass down a "downstream collector". This is a collector that does something special with the **values**.

```
Stream<String> names = Stream.of("Joe", "Tom", "Tom", "Alan", "Peter");
Map<Integer, Set<String>> map =
        names.collect(
            Collectors.groupingBy(
                    String::length,      // key Function
                    Collectors.toSet()) // what to do with the values
        );
System.out.println(map);// {3=[Joe, Tom], 4=[Alan], 5=[Peter]}
```

Terminal Operations
Collectors.groupingBy()

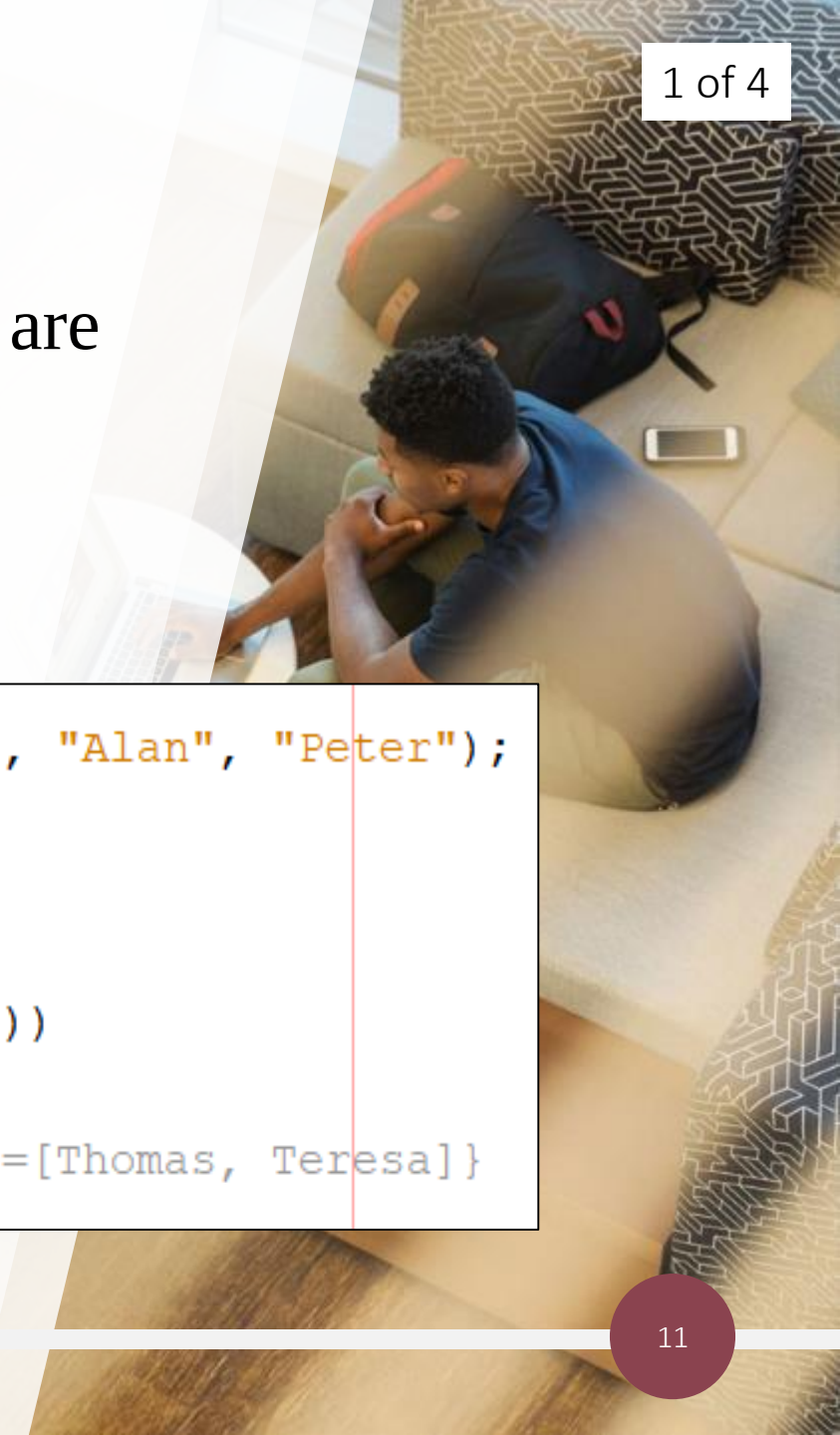
- There are no guarantees on the type of Map returned.
- What if we wanted to ensure we got back a *TreeMap* but leave the values as a *List*? We can achieve this by using the (optional) map type *Supplier* while passing down the *toList()* collector.

```
Stream<String> names = Stream.of("Joe", "Tom", "Tom", "Alan", "Peter");
TreeMap<Integer, List<String>> map =
        names.collect(
            Collectors.groupingBy(
                    String::length,
                    TreeMap::new,          // map type Supplier
                    Collectors.toList())// downstream collector
        );
System.out.println(map);// {3=[Joe, Tom, Tom], 4=[Alan], 5=[Peter]}
System.out.println(map.getClass());// class java.util.TreeMap
```



Terminal Operations
Collectors.partitioningBy()

- Partitioning is a special case of grouping where there are only two possible groups – true and false.
- The keys will be the booleans *true* and *false*.



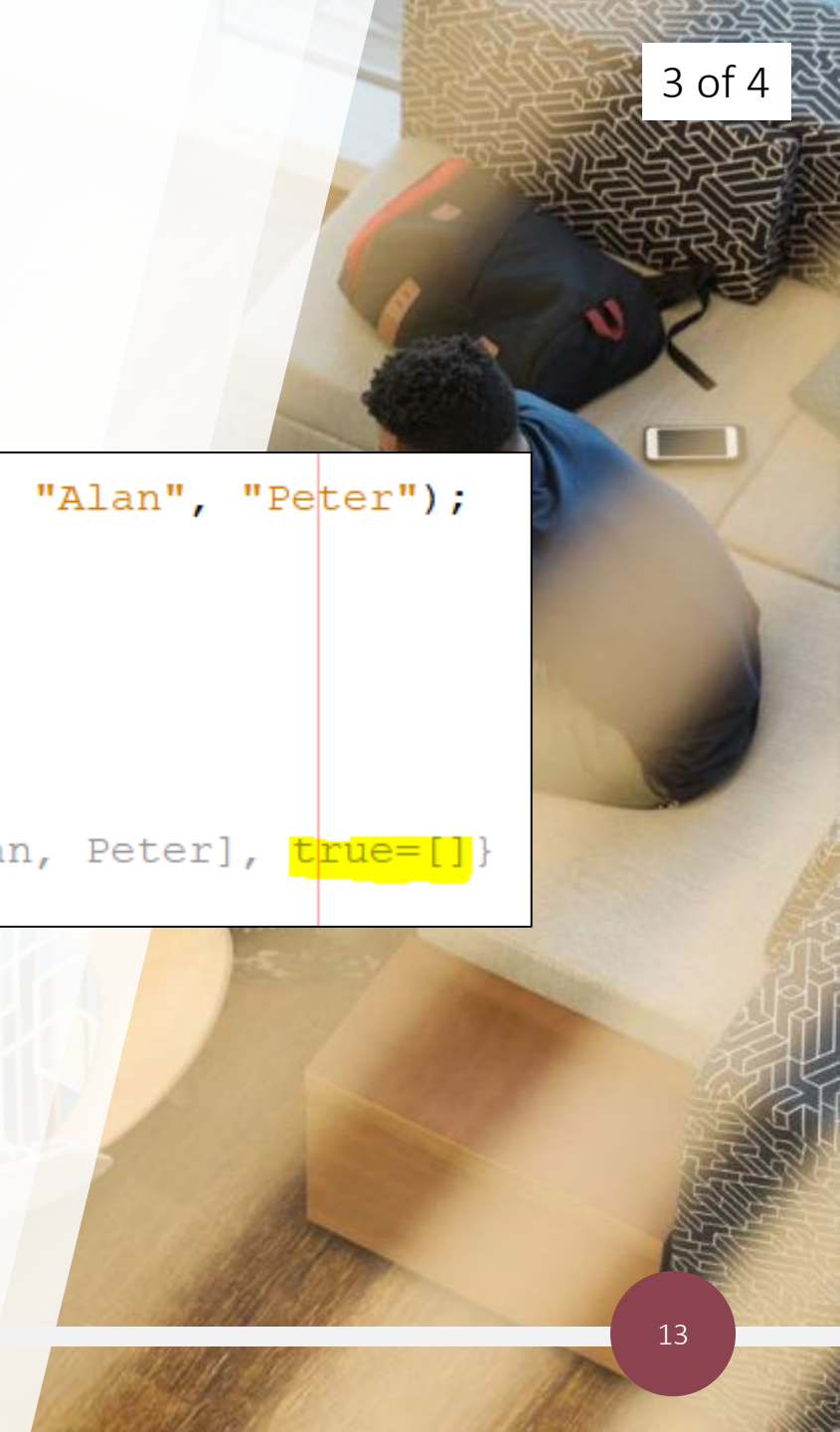
```
Stream<String> names = Stream.of("Thomas", "Teresa", "Mike", "Alan", "Peter");
Map<Boolean, List<String>> map =
        names.collect(
            // pass in a Predicate
            Collectors.partitioningBy(s -> s.startsWith("T"))
        );
System.out.println(map);// {false=[Mike, Alan, Peter], true=[Thomas, Teresa]}
```

Terminal Operations
Collectors.partitioningBy()


```
Stream<String> names = Stream.of("Thomas", "Teresa", "Mike", "Alan", "Peter");
Map<Boolean, List<String>> map =
        names.collect(
            // pass in a Predicate
            Collectors.partitioningBy(s -> s.length() > 4)
        );
System.out.println(map);// {false=[Mike, Alan], true=[Thomas, Teresa, Peter]}
```

Terminal Operations
Collectors.partitioningBy()


```
Stream<String> names = Stream.of("Thomas", "Teresa", "Mike", "Alan", "Peter");
Map<Boolean, List<String>> map =
        names.collect(
            // forcing an empty list
            Collectors.partitioningBy(s -> s.length() > 10)
        );
System.out.println(map);// {false=[Thomas, Teresa, Mike, Alan, Peter], true=[]}
```



Terminal Operations
Collectors.partitioningBy()

- As with *groupingBy()*, we can change the values type from *List* to *Set*.

```
Stream<String> names = Stream.of("Alan", "Teresa", "Mike", "Alan", "Peter");
Map<Boolean, Set<String>> map =
        names.collect(
            Collectors.partitioningBy(
                s -> s.length() > 4,// predicate
                Collectors.toSet()    )
        );
System.out.println(map);// {false=[Mike, Alan], true=[Teresa, Peter]}
```