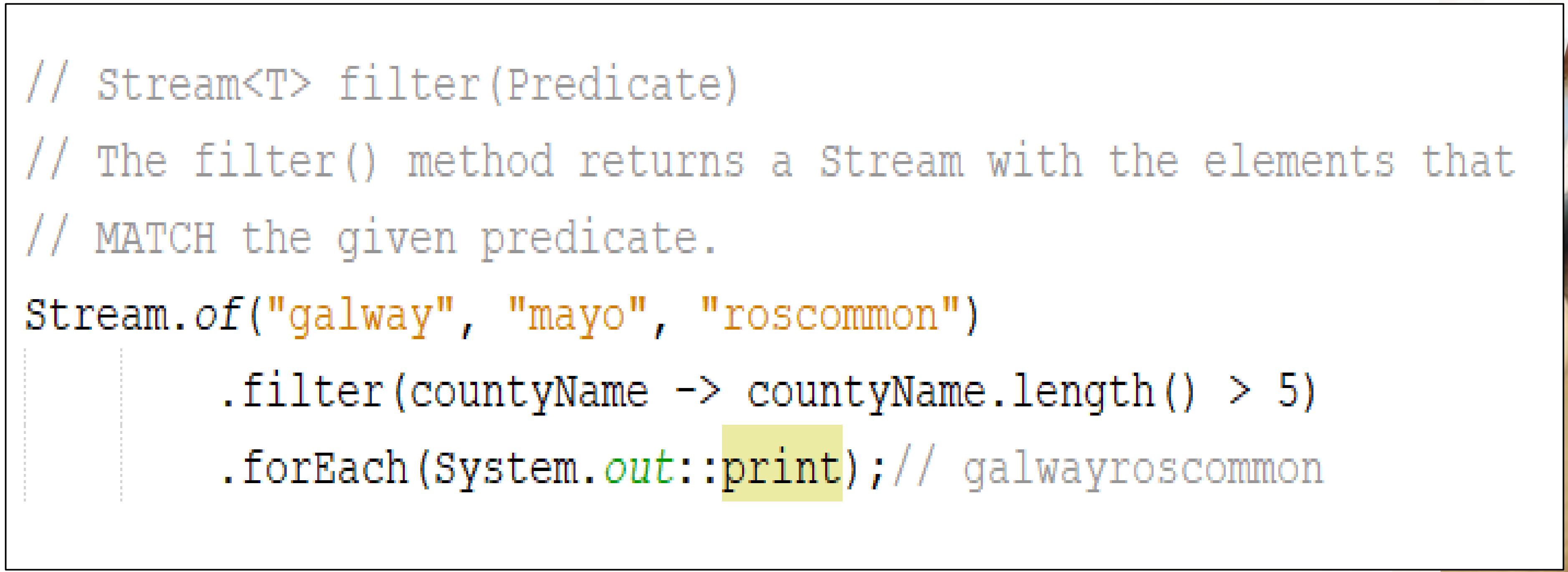
Streams

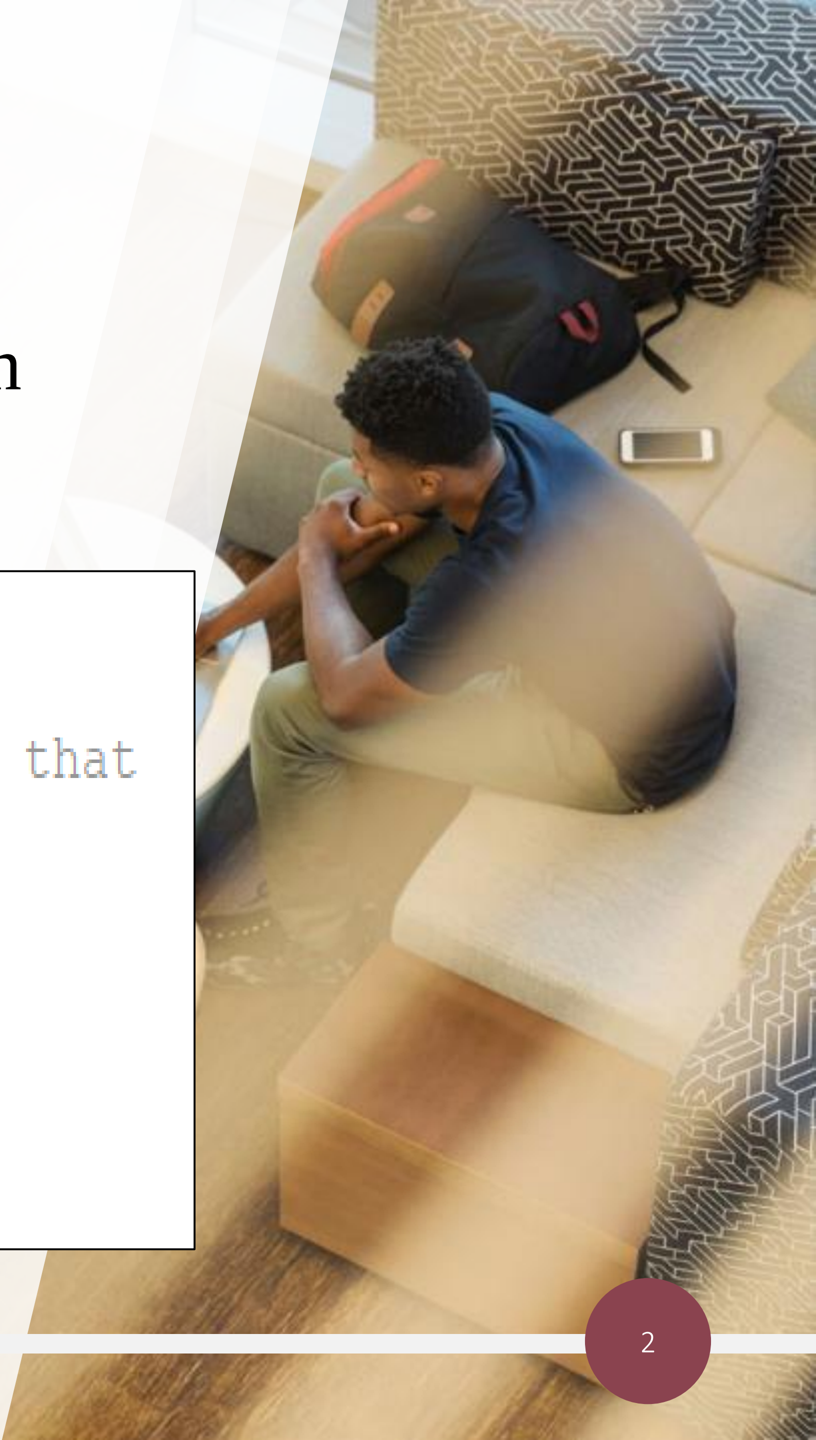
Intermediate Operations

Intermediate Operations
filter()

- Unlike a terminal operation, an intermediate operation produces a stream as a result.



```
// Stream<T> filter(Predicate)
// The filter() method returns a Stream with the elements that
// MATCH the given predicate.
Stream.of("galway", "mayo", "roscommon")
        .filter(countyName -> countyName.length() > 5)
        .forEach(System.out::print);// galwayroscommon
```

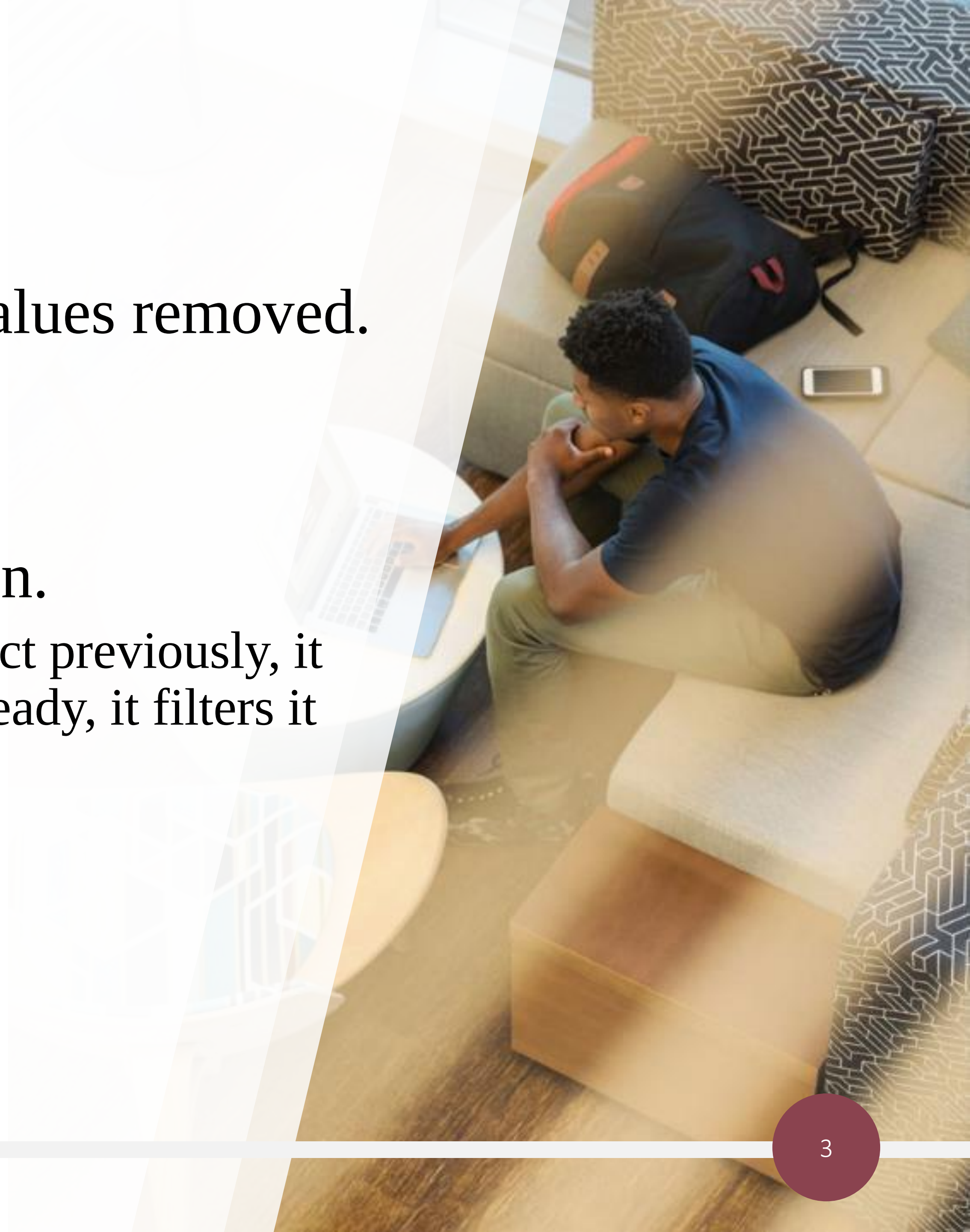


Copyright © Seán Kennedy

Intermediate Operations
distinct()

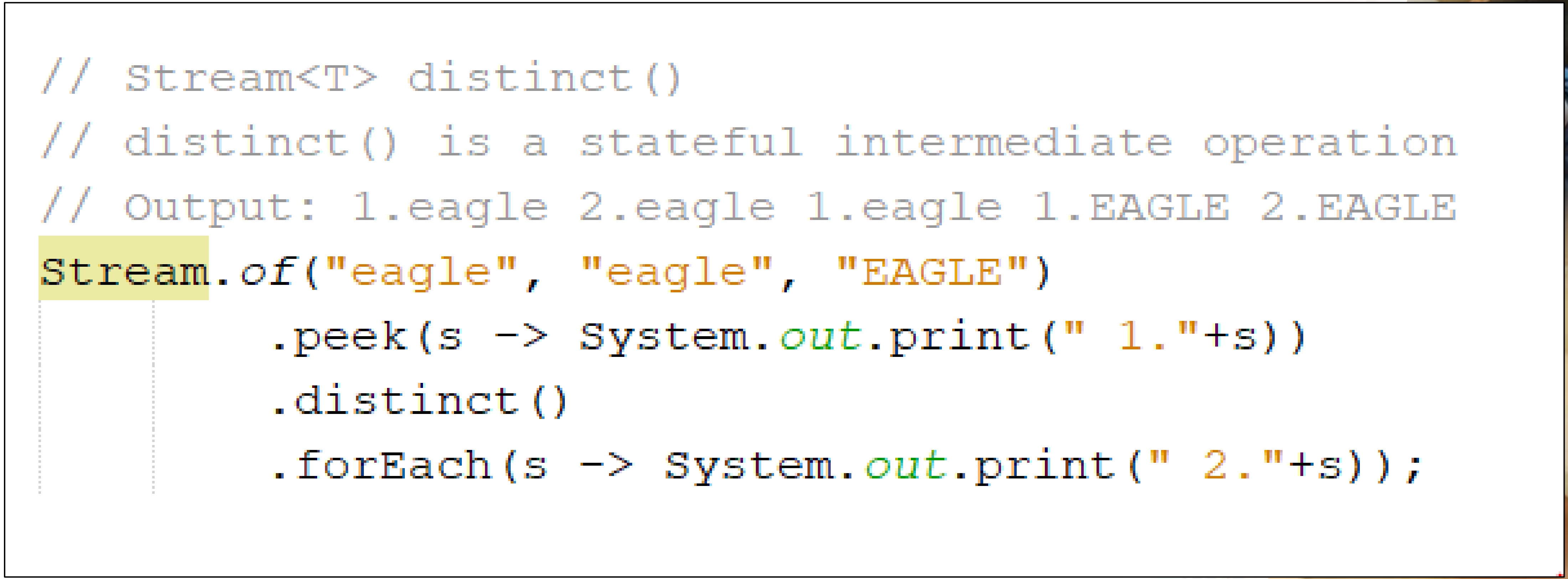
- *distinct()* returns a stream with duplicate values removed.
  - *equals()* is used i.e. case sensitive.

- *distinct()* is a <u>stateful</u> intermediate operation.
  - it behaves like a filter – if it has not seen the object previously, it passes it on and remembers it; if it has seen it already, it filters it out.



Copyright © Seán Kennedy

Intermediate Operations
distinct()



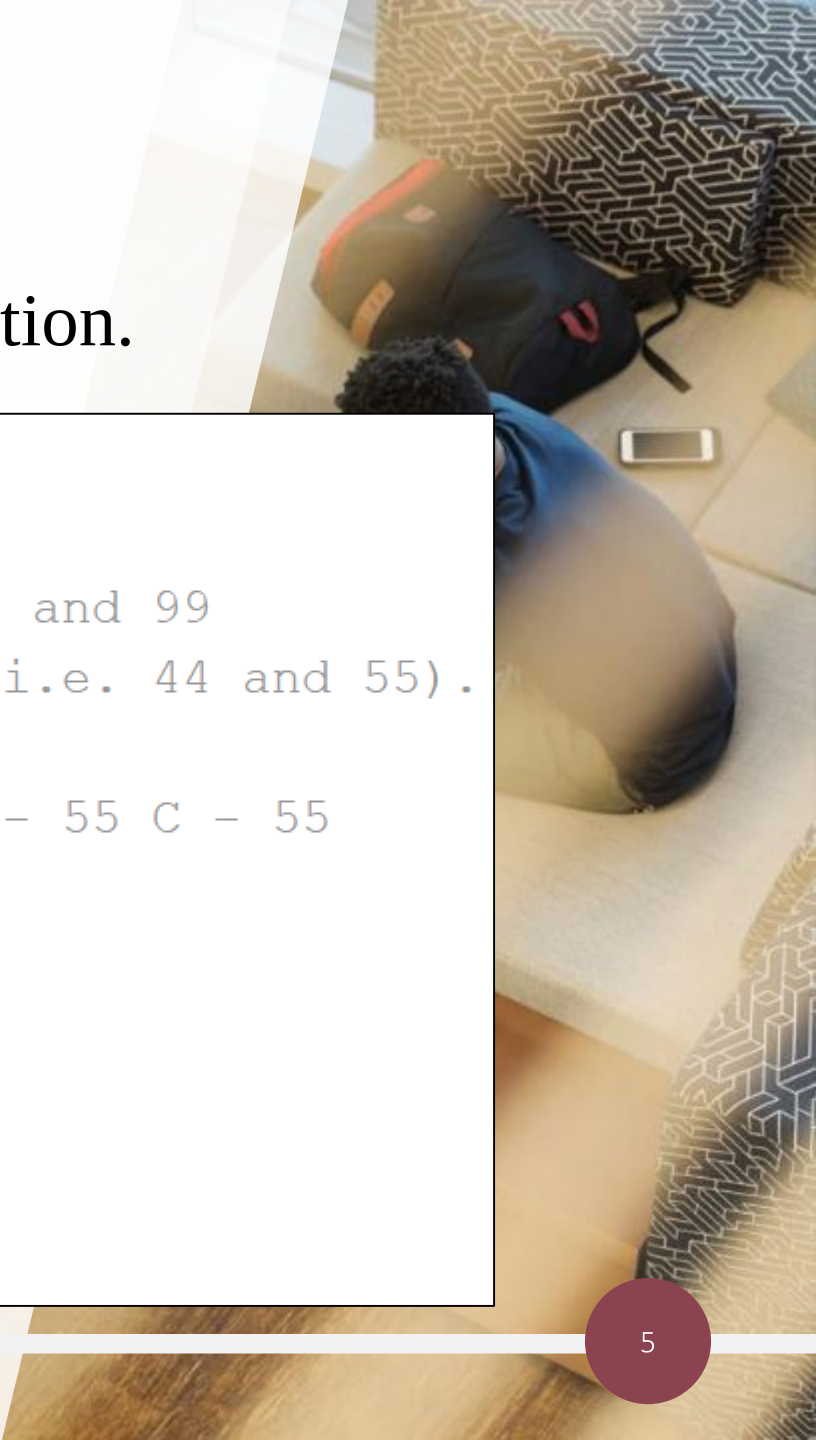
```
// Stream<T> distinct()
// distinct() is a stateful intermediate operation
// Output: 1.eagle 2.eagle 1.eagle 1.EAGLE 2.EAGLE
Stream.of("eagle", "eagle", "EAGLE")
        .peek(s -> System.out.print(" 1."+s))
        .distinct()
        .forEach(s -> System.out.print(" 2."+s));
```

Copyright © Seán Kennedy

Intermediate Operations
limit()

- *limit()* is a short-circuiting stateful intermediate operation.

```
// Stream<T> limit(long maxSize)
// limit is a short-circuiting stateful
// intermediate operation. Lazy evaluation - 66, 77, 88 and 99
// are not streamed as they are not needed (limit of 2 i.e. 44 and 55).
// Output:
//  A - 11 A - 22 A - 33 A - 44 B - 44 C - 44 A - 55 B - 55 C - 55
Stream.of(11,22,33,44,55,66,77,88,99)
          .peek(n -> System.out.print(" A - "+n))
          .filter(n -> n > 40)
          .peek(n -> System.out.print(" B - "+n))
          .limit(2)
          .forEach(n -> System.out.print(" C - "+n));
```



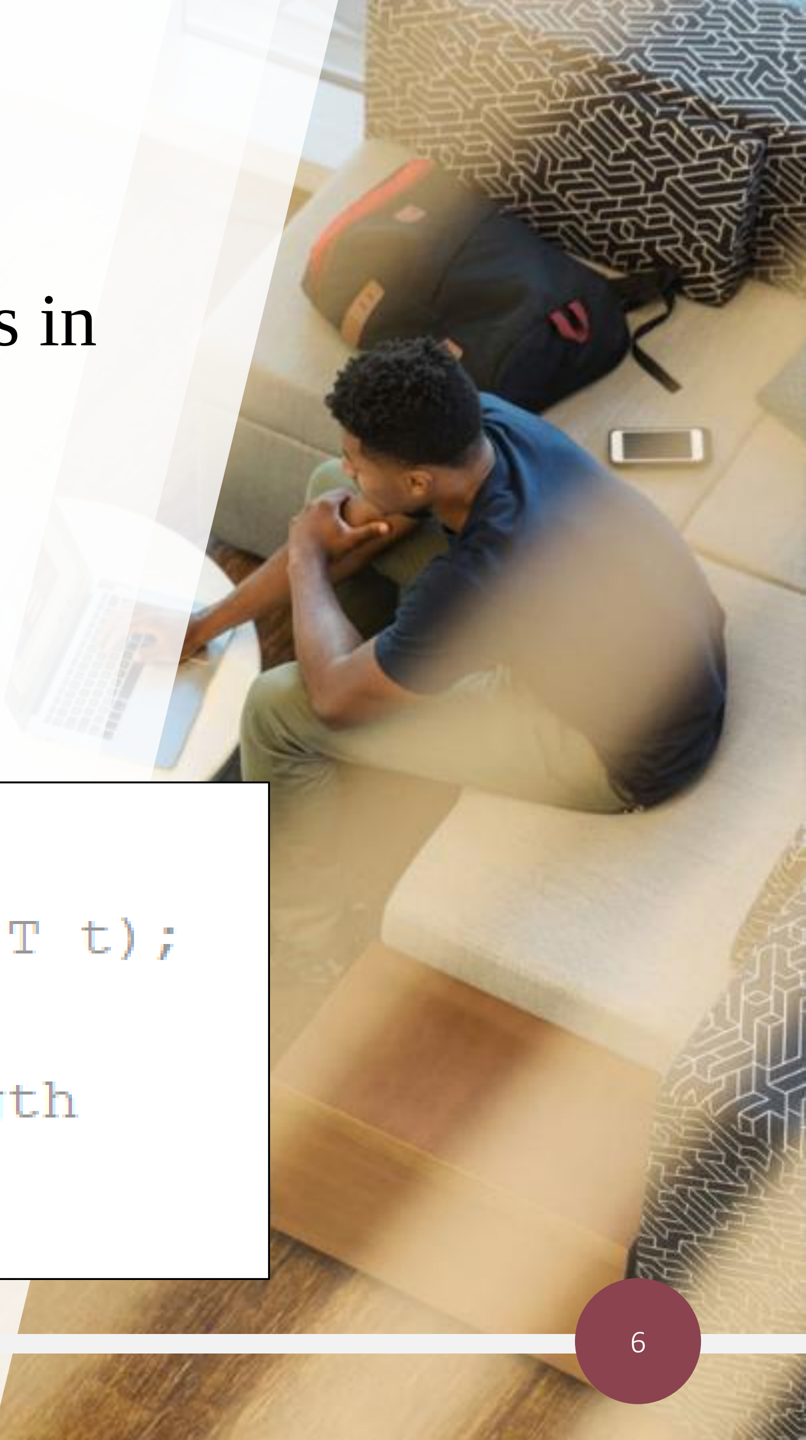
Copyright © Seán Kennedy

Intermediate Operations
map()

- *map()* creates a one-to-one mapping between elements in the stream and elements in the next stage of the stream.

- *map()* is for transforming data.

```
// <R> Stream<R> map(Function<T,R> mapper)
//     Function's functional method: R apply(T t);
Stream.of("book", "pen", "ruler")
        .map(s -> s.length()) // String::length
        .forEach(System.out::print);// 435
```



Copyright © Seán Kennedy

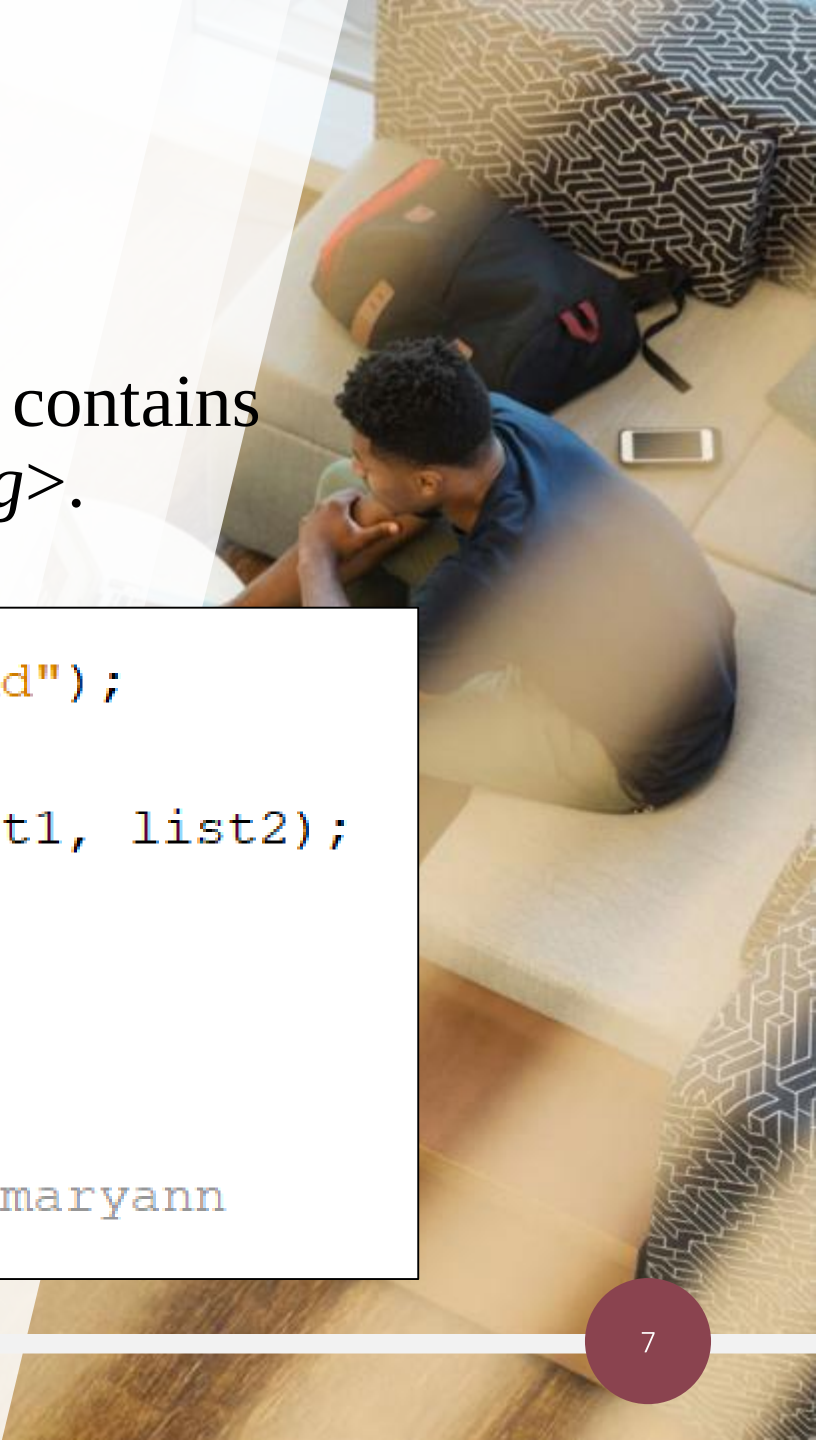
Intermediate Operations
flatMap()

✓

- *flatMap()* takes each element in the stream e.g. Stream<*List*<*String*>> and makes any elements it contains top-level elements in a single stream e.g. Stream<*String*>.

```
List<String> list1 = Arrays.asList("sean", "desmond");
List<String> list2 = Arrays.asList("mary", "ann");
Stream<List<String>> streamOfLists = Stream.of(list1, list2);

// flatMap(Function(T, R)) IN:T OUT:R
//  flatMap(List<String>, Stream<String>)
streamOfLists.flatMap(list -> list.stream())
        .forEach(System.out::print);// seandesmondmaryann
```



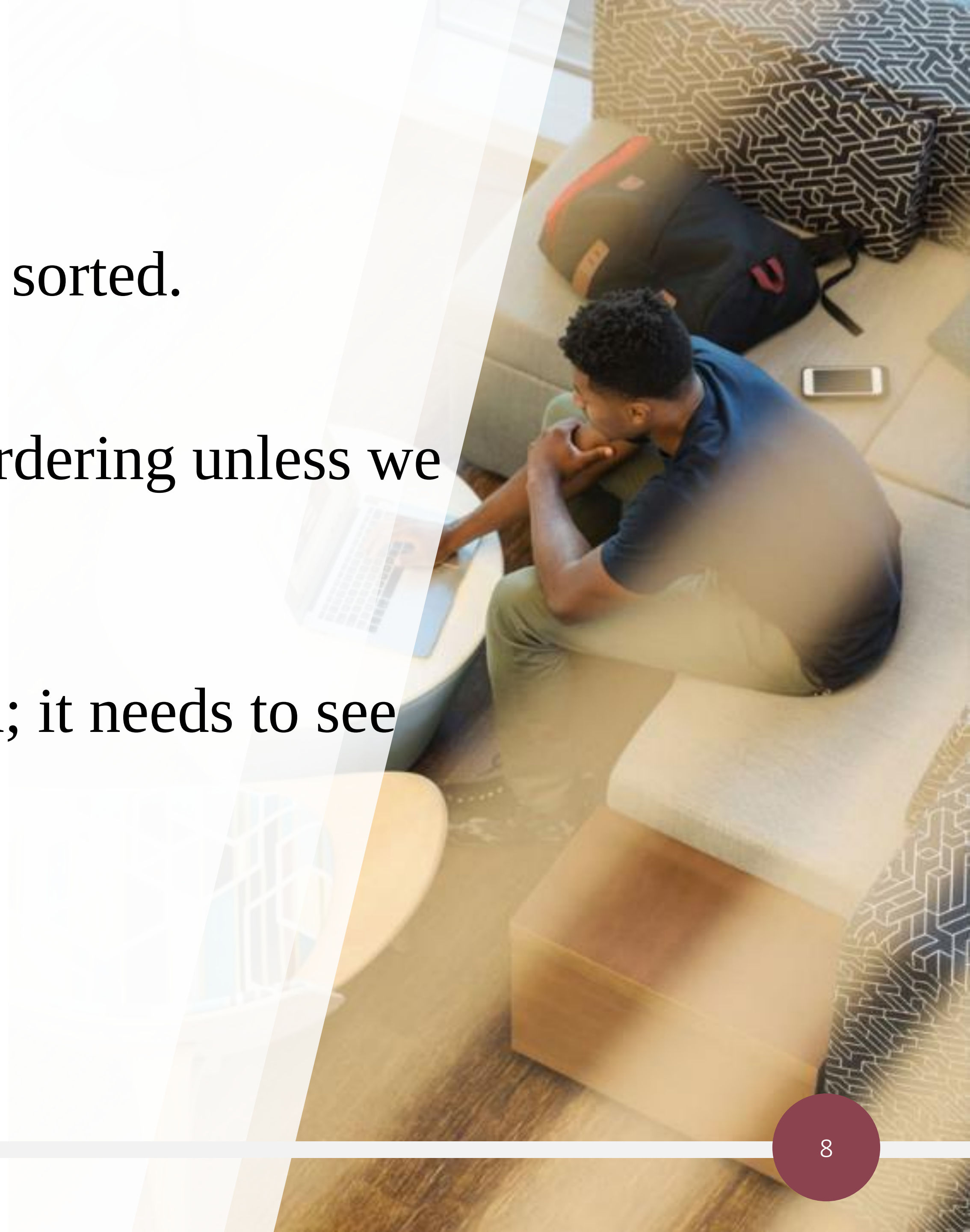
Copyright © Seán Kennedy

Intermediate Operations
sorted()

- *sorted()* returns a stream with the elements sorted.

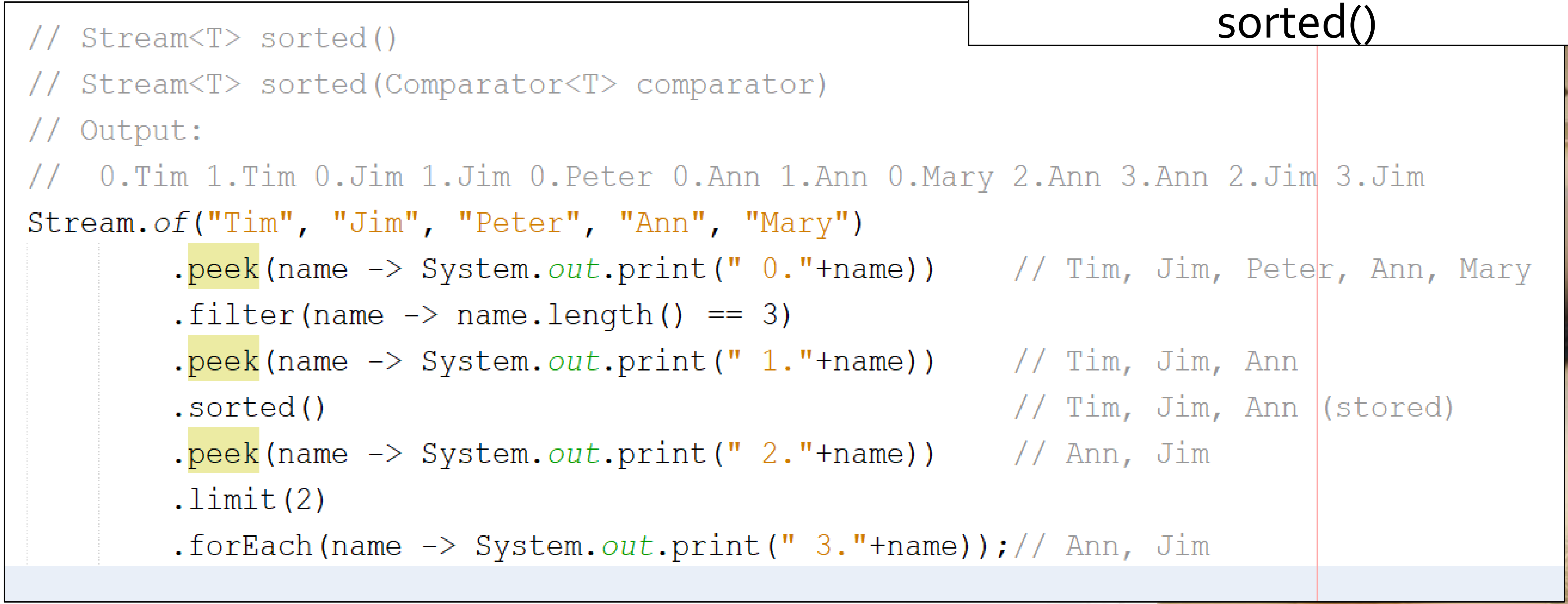
- Just like sorting arrays, Java uses natural ordering unless we provide a comparator.

- *sorted()* is a stateful intermediate operation; it needs to see all of the data before it can sort it.



Copyright © Seán Kennedy

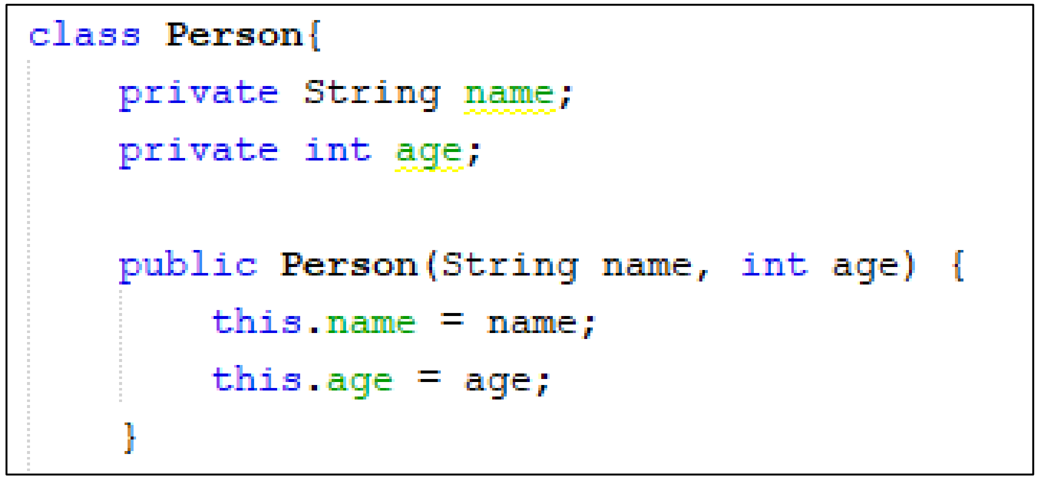
Intermediate Operations
sorted()



```
// Stream<T> sorted()
// Stream<T> sorted(Comparator<T> comparator)
// Output:
//  0.Tim 1.Tim 0.Jim 1.Jim 0.Peter 0.Ann 1.Ann 0.Mary 2.Ann 3.Ann 2.Jim 3.Jim
Stream.of("Tim", "Jim", "Peter", "Ann", "Mary")
        .peek(name -> System.out.print(" 0."+name))    // Tim, Jim, Peter, Ann, Mary
        .filter(name -> name.length() == 3)
        .peek(name -> System.out.print(" 1."+name))    // Tim, Jim, Ann
        .sorted()                                      // Tim, Jim, Ann (stored)
        .peek(name -> System.out.print(" 2."+name))    // Ann, Jim
        .limit(2)
        .forEach(name -> System.out.print(" 3."+name));// Ann, Jim
```

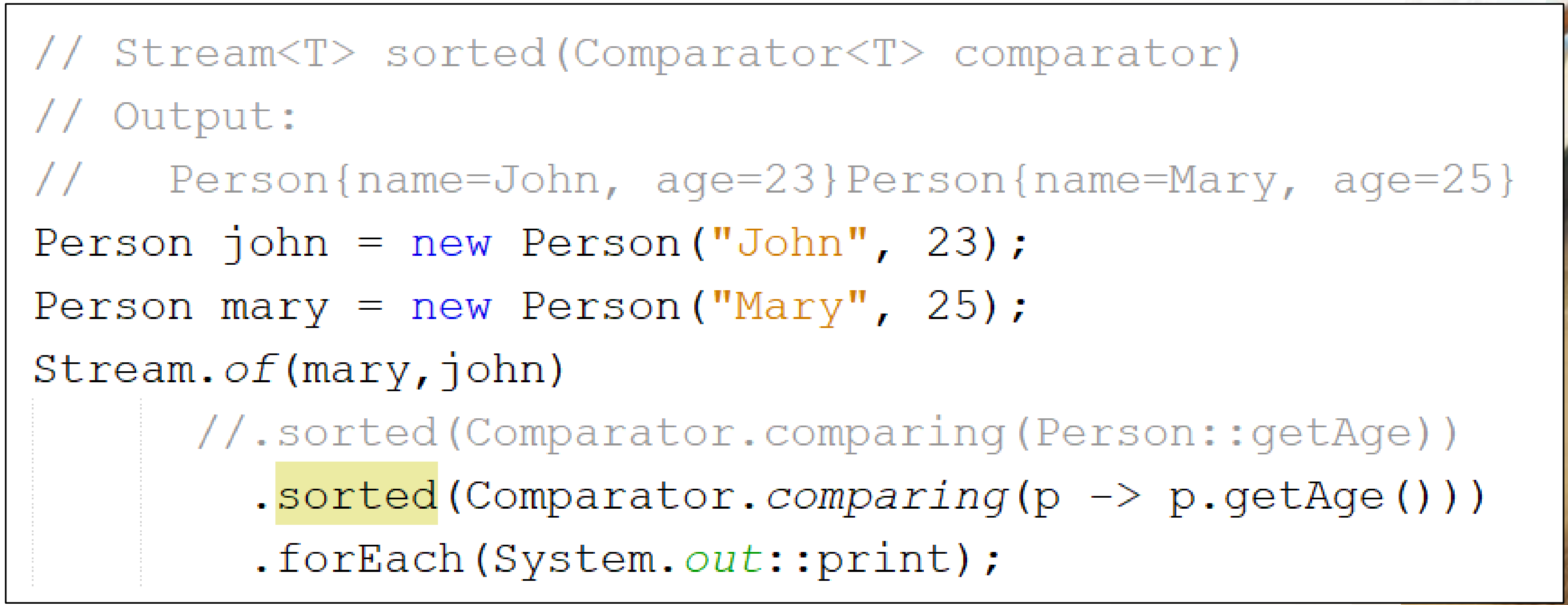
Copyright © Seán Kennedy

Intermediate Operations
sorted(Comparator)

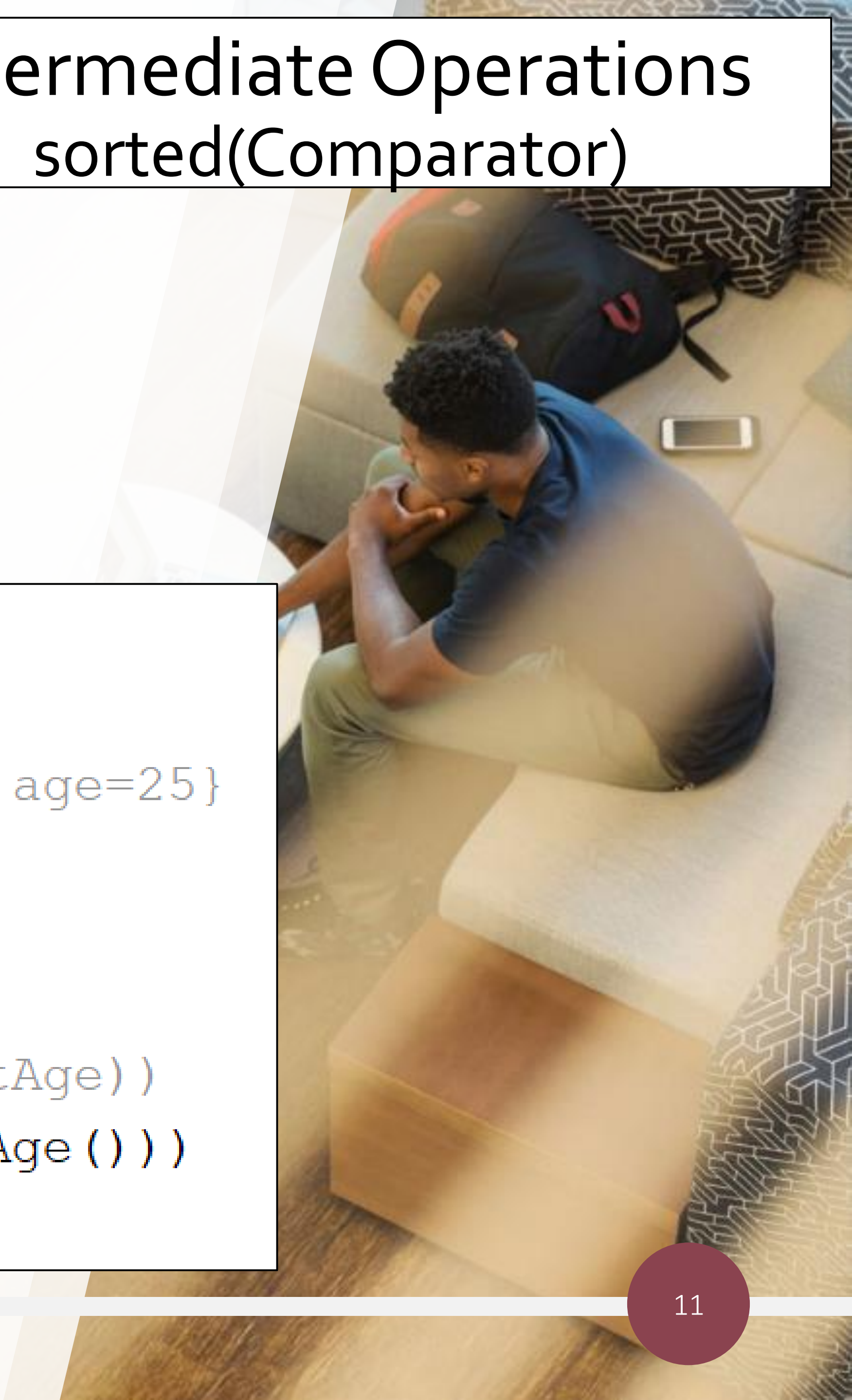


```java
class Person{
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
```



```java
// Stream<T> sorted(Comparator<T> comparator)
// Output:
//   Person{name=John, age=23}Person{name=Mary, age=25}
Person john = new Person("John", 23);
Person mary = new Person("Mary", 25);
Stream.of(mary,john)
      //.sorted(Comparator.comparing(Person::getAge))
        .sorted(Comparator.comparing(p -> p.getAge()))
        .forEach(System.out::print);
```



Copyright © Seán Kennedy