

A photograph of four students in a library setting. A young man in a grey t-shirt is smiling and looking at a laptop. A young woman with glasses is looking at the laptop. Another young woman is looking at a book. A young man is looking at the laptop. The background is filled with bookshelves. The image has a blue and red overlay.

Streams

Introduction and Stream Pipelines

What is a Stream?

- Like lambdas and functional interfaces, streams were another new addition in Java 8.
- A *stream* is a sequence of data that can be processed with operations.
- Streams are **not** another way of organising data, like an array or a *Collection*. Streams do not hold data; streams are all about processing data efficiently.



What is a Stream?

- While streams make code more concise, their big advantage is that streams, by using a pipeline, can, in certain situations, greatly improve the efficiency of data processing.
- The real power of streams comes from the multiple intermediate operations you can perform on the stream.



The Pipeline

- A *stream pipeline* consists of the operations that run on a stream to produce a result.
- There are 3 parts to a stream pipeline:
 - a) Source – where the stream comes from e.g. array, collection or file.
 - b) Intermediate operations – transforms the stream into another one. There can be as few or as many as required.
 - c) Terminal operation – required to start the whole process and produces the result. Streams can only be used once i.e. streams are no longer usable after a terminal operation completes (re-generate the stream if necessary).



The Pipeline

- *filter()* is an intermediate operation and as such can filter the stream and pass on the filtered stream to the next operation (another intermediate operation or a terminal operation).
- *count()* and *forEach()* are both terminal operations that end the stream.
- The pipeline operations are the way in which we specify how and in what order we want the data in the source manipulated. Remember, streams don't hold any data.




```
/* Output:
```

```
98.4
```

```
100.2
```

```
100.2
```

```
87.9
```

```
102.8
```

```
102.8
```

```
Number of temps > 100 is: 2
```

```
*/
```

```
List<Double> temps = Arrays.asList(98.4, 100.2, 87.9, 102.8);
```

```
System.out.println("Number of temps > 100 is: "+
```

```
    temps
```

```
        .stream() // create the stream
```

```
        .peek(System.out::println) // show the value
```

```
        .filter(temp -> temp > 100) // filter it
```

```
        .peek(System.out::println) // show the value
```

```
        .count()); // 2
```

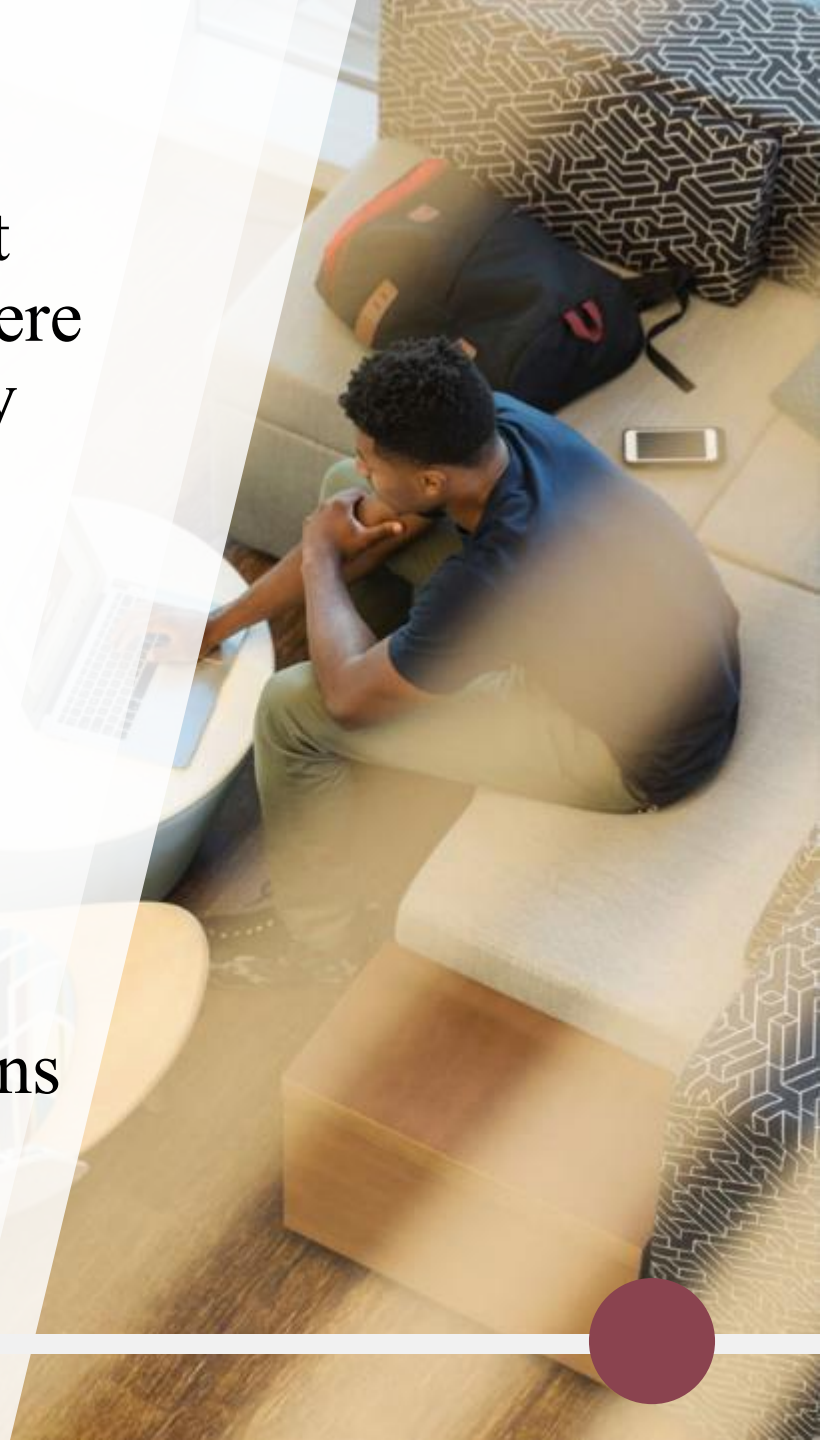
A group of four students are sitting at a table in a library, surrounded by bookshelves. They are looking at a laptop and some papers, appearing to be in a collaborative study session. The image has a semi-transparent blue overlay on the left side and a semi-transparent red overlay on the right side.

Streams

Streams are Lazy

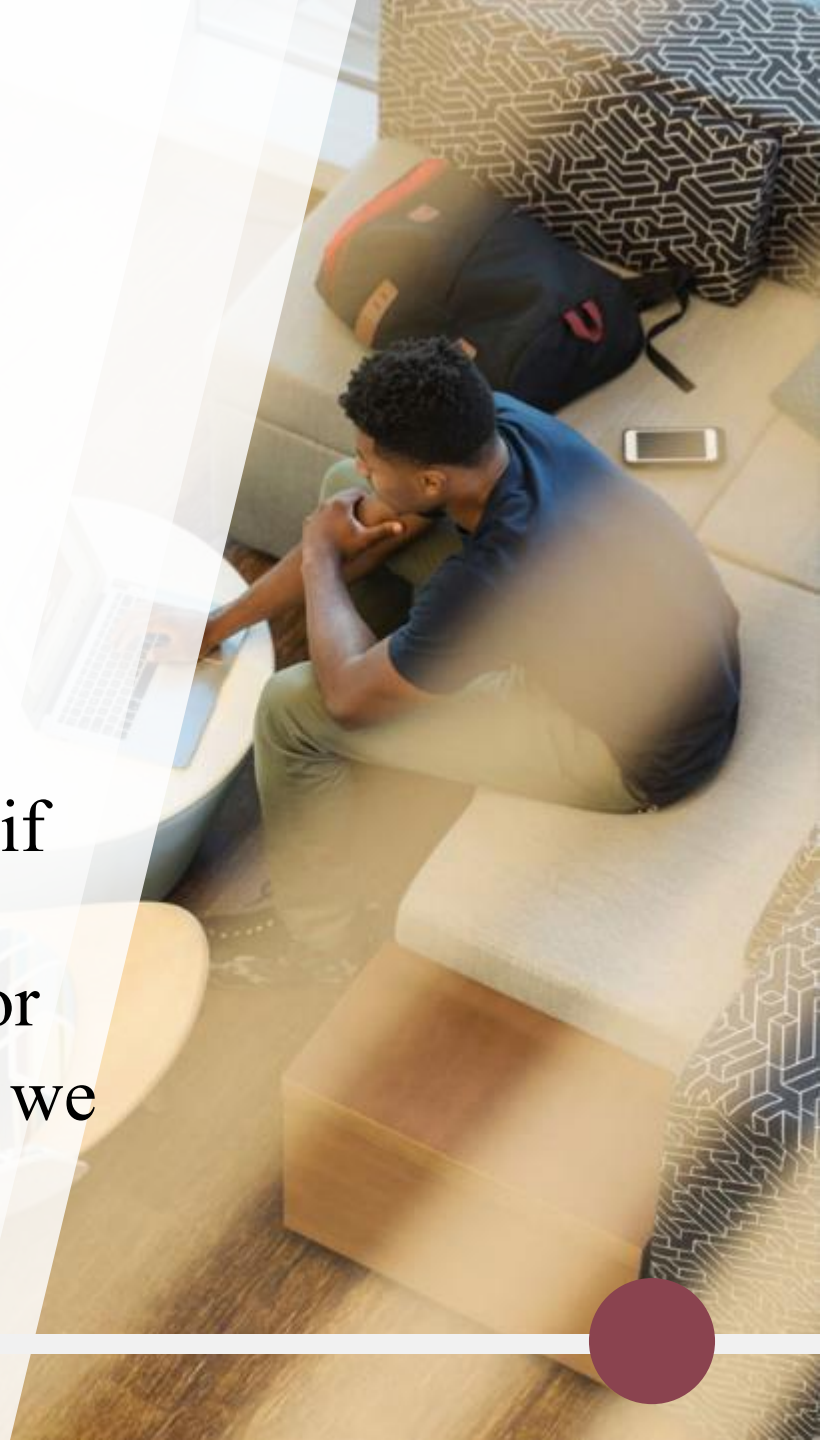
Streams are Lazy

- The principle of “lazy” evaluation is that you get what you need only when you need it. For example, if you were displaying 10,000 records to a user, the principle of lazy evaluation would be to retrieve 50 and while the user is viewing these, retrieve another 50 in the background.
- “Eager” evaluation would be to retrieve all 10,000 records in one go.
- With regard to streams, this means that nothing happens until the terminal operation occurs.



Streams are Lazy

- The pipeline specifies what operations we want performed (on the source) and in which order.
- This enables the JDK to reduce operations whenever possible.
- For example, why run an operation on a piece of data if the operation is not required:
 - we have found the data element we were looking for
 - we may have a limit set on the number of elements we want to operate on



```
/* Each element moves along the chain vertically:
```

```
    filter: Alex
```

```
    forEach: Alex
```

```
    filter: David
```

```
    forEach: David
```

```
    filter: April
```

```
    forEach: April
```

```
    filter: Edward
```

```
    forEach: Edward */
```

```
Stream.of("Alex", "David", "April", "Edward")
```

```
    .filter(s -> {
```

```
        System.out.println("filter: "+s);
```

```
        return true;
```

```
    })
```

```
    .forEach(s -> System.out.println("forEach: "+s));
```

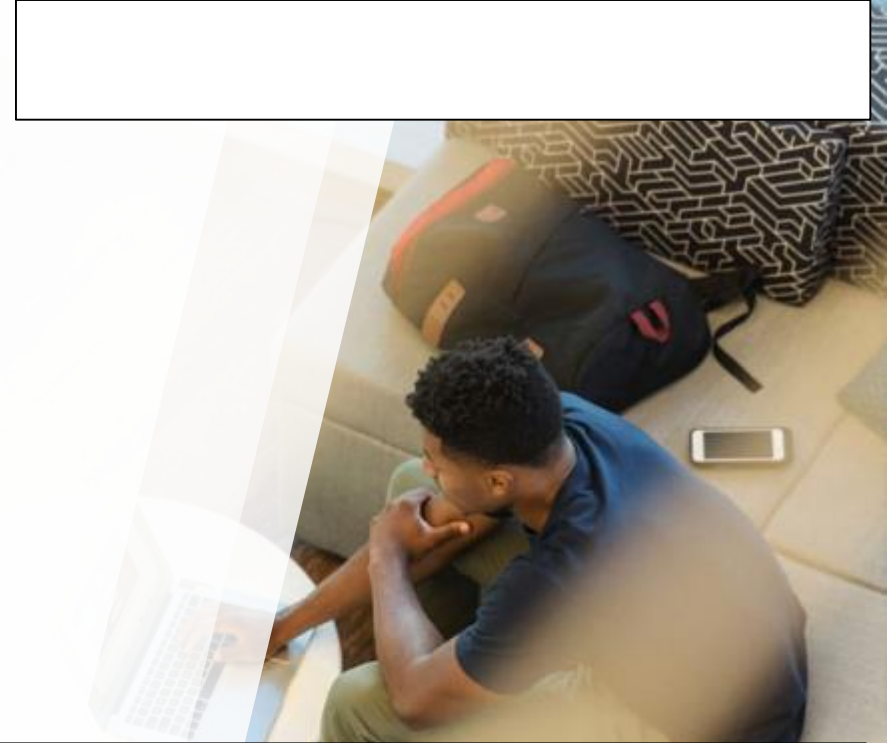
Streams are Lazy


```
/* This can help in reducing the actual number of operations - instead of  
   mapping "Alex", "David", "April" and "Edward" and then anyMatch() on  
   "Alex" (5 operations in total), we process the elements vertically resulting in  
   only 2 operations. While this is a small example, it shows the benefits to be  
   had if we had millions of data elements to be processed.
```

```
    map: Alex  
    anyMatch: ALEX    */  
Stream.of("Alex", "David", "April", "Edward")  
    .map(s -> {  
        System.out.println("map: "+s);  
        return s.toUpperCase();  
    })  
    .anyMatch(s -> { // ends when first true is returned (Alex)  
        System.out.println("anyMatch: "+s);  
        return s.startsWith("A");  
    });
```

Streams are Lazy

```
List<String> names =  
    Arrays.asList("April", "Ben", "Charlie",  
        "David", "Benildus", "Christian");  
names.stream()  
    .peek(System.out::println)  
    .filter(s -> {  
        System.out.println("filter1 : "+s);  
        return s.startsWith("B") || s.startsWith("C"); } )  
    .filter(s -> {  
        System.out.println("filter2 : "+s);  
        return s.length() > 3; } )  
    .limit(1) // intermediate operation Stream<T> limit(long)  
    .forEach(System.out::println); // terminal operation
```



April	- peek
filter1 : April	- filter1 removes April
Ben	- peek
filter1 : Ben	- filter1 passes Ben on
filter2 : Ben	- filter2 removes Ben
Charlie	- peek
filter1 : Charlie	- filter1 passes Charlie on
filter2 : Charlie	- filter2 passes Charlie on
Charlie	- forEach()

Note: limit(1) means David, Benildus or Christian are not processed at all i.e. none of them appear in the output via "peek() "

A photograph of four students in a library setting. A young woman with long dark hair is on the left, smiling. Next to her is a young man with dark hair, also smiling. To his right is a young woman with glasses and a dark polka-dot top, looking at a laptop. On the far right is a young man with dark hair, seen from the back/side, looking at the laptop. They are all gathered around a table with a laptop, books, and papers. The background is filled with bookshelves. A semi-transparent blue diagonal band runs across the image, and a semi-transparent red horizontal band is at the bottom.

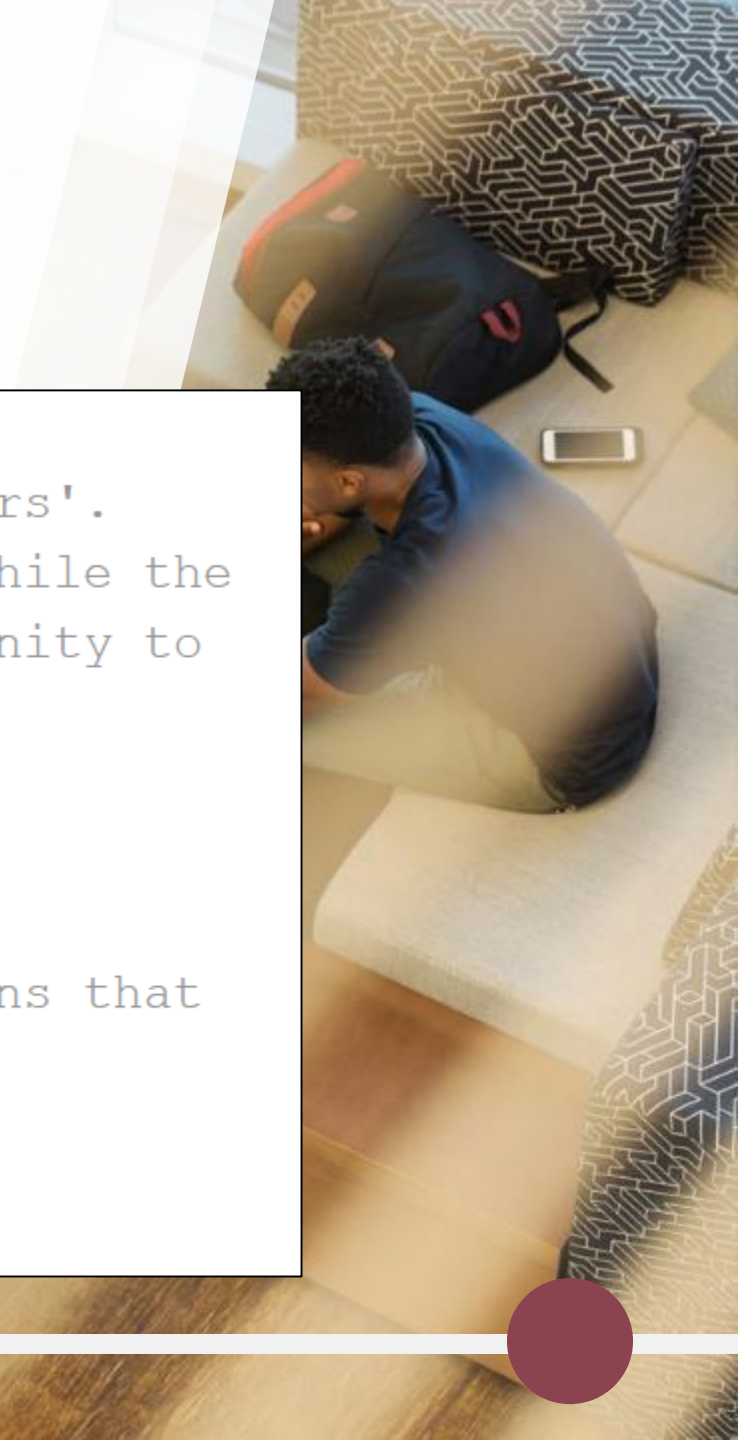
Streams

Creating Streams

Creating a Stream from an Array

- *Arrays.stream()* can be used to stream an array.

```
Double[] numbers = {1.1, 2.2, 3.3};  
// Arrays.stream() creates a stream from the array 'numbers'.  
// The array is considered the source of the stream and while the  
// data is flowing through the stream, we have an opportunity to  
// operate on the data.  
Stream<Double> stream1 = Arrays.stream(numbers);  
  
// lets perform an operation on the data  
// note that count() is a "terminal operation" - this means that  
// you cannot perform any more operations on the stream.  
long n = stream1.count();  
System.out.println("Number of elements: "+n); // 3
```



Creating a Stream from a Collection

- The default *Collection* interface method *stream()* is used.

```
List<String> animalList = Arrays.asList("cat", "dog", "sheep");
// using stream() which is a default method in Collection interface
Stream<String> streamAnimals = animalList.stream();
System.out.println("Number of elements: "+streamAnimals.count()); // 3

// stream() is a default method in the Collection interface and therefore
// is inherited by all classes that implement Collection. Map is NOT one
// of those i.e. Map is not a Collection. To bridge between the two, we
// use the Map method entrySet() to return a Set view of the Map (Set
// IS-A Collection).
Map<String, Integer> namesToAges = new HashMap<>();
namesToAges.put("Mike", 22);namesToAges.put("Mary", 24);namesToAges.put("Alice", 31);
System.out.println("Number of entries: "+
    namesToAges
        .entrySet() // get a Set (i.e. Collection) view of the Map
        .stream()   // stream() is a default method in Collection
        .count()); // 3
```

Creating a Stream with *Stream.of()*

- *Stream.of()* is a static generically-typed utility method that accepts a varargs parameter and returns an ordered stream of those values.

```
import java.util.stream.Stream;
```

```
class Dog{}
```

```
static <T> Stream<T> of(T... values)
```

```
public class BuildStreams {
```

```
    public static void main(String []args){
```

```
        Stream<Integer> streamI = Stream.of(1,2,3);
```

```
        System.out.println(streamI.count()); // 3
```

```
        Stream<String> streamS = Stream.of("a", "b", "c", "d");
```

```
        System.out.println(streamS.count()); // 4
```

```
        Stream<Dog> streamD = Stream.of(new Dog());
```

```
        System.out.println(streamD.count()); // 1
```

```
    }
```

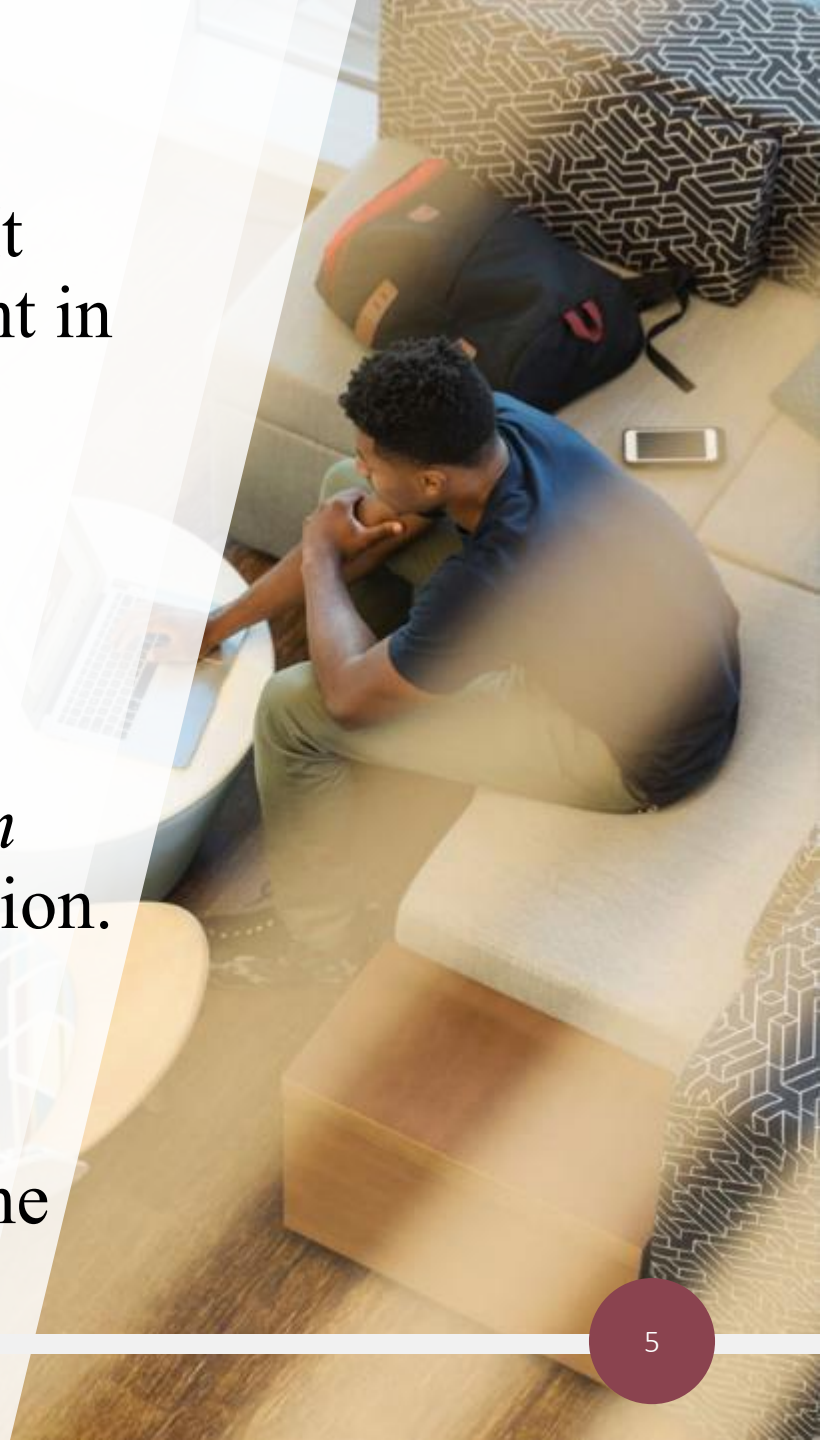
```
}
```


Creating a Stream from a File

- The *Files.lines()* method can be used to stream a file. It provides one line at a time from the file as a data element in the stream.

```
public static Stream<String> lines(Path path)  
    throws IOException
```

- To process the data from the stream, we use the *Stream* interfaces' *forEach()* method, which is a terminal operation.
- Similar to the *forEach()* for collections, it takes a *Consumer*, which enables us to process each line from the file.



Creating a Stream from a File

```
class Cat{  
    private String name, colour;  
    Cat(String name, String colour) {  
        this.name = name;  
        this.colour = colour;  
    }  
    @Override  
    public String toString() {  
        return "Cat{" + "name=" + name + ", colour=" + colour + "}";  
    }  
}
```

```

23 public class ProcessFile {
24     public static void main(String []args){
25         List<Cat> cats = loadCats("Cats.txt");
26         cats.forEach(System.out::println); // just print the Cat
27     }
28     public static List<Cat> loadCats(String filename){
29         List<Cat> cats = new ArrayList<>();
30         try(Stream<String> stream = Files.lines(Paths.get(filename))){
31             stream.forEach(line -> {
32                 String[] catsArray = line.split("/");
33                 cats.add(new Cat(catsArray[0], catsArray[1]));
34             });
35         } catch (IOException ioe) {
36             ioe.printStackTrace();
37         }
38         return cats;
39     }
40 }

```

Fido/Black
Lily/White

run:
Cat{name=Fido, colour=Black}
Cat{name=Lily, colour=White}
BUILD SUCCESSFUL (total time: 3 seconds)

Output

Note that inside the lambda expression, variables from the enclosing scope are either *final* or *effectively final*. This means that while we can add elements to 'cats' we cannot change what 'cats' refers to i.e. we cannot say `cats=new ArrayList<>();`

Infinite Streams

- Infinite streams can be created in the following ways:

```
// infinite stream of random unordered numbers
// between 0..9 inclusive
//      Stream<T> generate(Supplier<T> s)
//      Supplier is a functional interface:
//      T get()
Stream<Integer> infStream = Stream.generate(() -> {
    return (int) (Math.random() * 10);
});
// keeps going until I kill it.
infStream.forEach(System.out::println);
```

1 of 2



```
// infinite stream of ordered numbers
//    2, 4, 6, 8, 10, 12 etc...
// iterate(T seed, UnaryOperator<T> fn)
//    UnaryOperator is-a Function<T, T>
//    T apply(T t)
Stream<Integer> infStream = Stream.iterate(2, n -> n + 2);

// keeps going until I kill it.
infStream.forEach(System.out::println);
```

Infinite Streams

- Infinite streams can be turned into finite streams with operations such as *limit(long)* :

```
// finite stream of ordered numbers
// 2, 4, 6, 8, 10, 12, 14, 16, 18, 20
Stream
    .iterate(2, n -> n + 2)
    // limit() is a short-circuiting stateful
    // intermediate operation
    .limit(10)
    // forEach(Consumer) is a terminal operation
    .forEach(System.out::println);
```



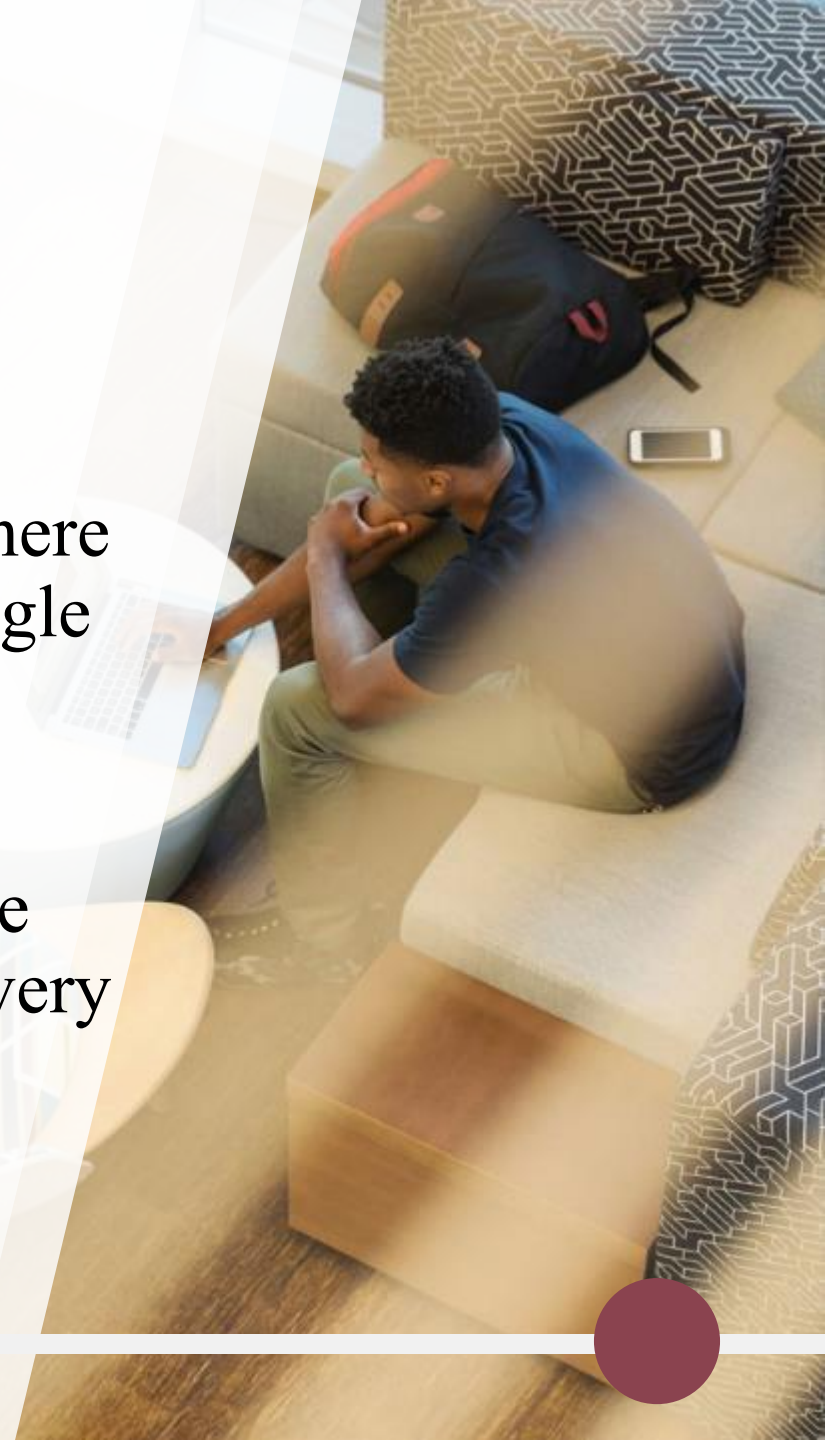
A group of four students are gathered around a table in a library, looking at a laptop screen. The background is filled with bookshelves. The image has a semi-transparent blue overlay on the left side and a semi-transparent red overlay at the bottom.

Streams

Terminal Operations

Terminal Operations

- Terminal operations can be performed without any intermediate operations but not the other way around.
- *Reductions* are a special type of terminal operation where all of the contents of the stream are combined into a single primitive or Object (Collection).
- We will have a look at the most common ones over the coming slides. Note that the comments in the code are very important.



Stream terminal operations

<u>Method</u>	<u>Return value</u>	<u>Reduction¹</u>
count()	long	Yes
min(), max()	Optional<T> - stream may be empty	Yes
findAny(), findFirst()	Optional<T>	No – may not look at all of the elements
allMatch(), anyMatch(), noneMatch()	boolean	No – may not look at all of the elements
forEach()	void	No (as it does not return anything)
reduce()	varies	Yes
collect()	varies	Yes

¹ Reductions are a special type of terminal operation where ALL of the contents of the stream are combined into a single primitive or Object e.g. long or Collection.

Terminal Operations

count(), min(), max()

```
long count = Stream.of("dog", "cat")
                    .count();
System.out.println(count); // 2
```

```
// Optional<T> min(Comparator)
// Optional<T> max(Comparator)
// Optional introduce in Java 8 to replace 'null'. If the stream is
// empty then the Optional will be empty (and we won't have to
// deal with null).
Optional<String> min = Stream.of("deer", "horse", "pig")
                             .min((s1, s2) -> s1.length()-s2.length());
min.ifPresent(System.out::println); // pig

Optional<Integer> max = Stream.of(4, 6, 2, 12, 9)
                             .max((i1, i2) -> i1-i2);
max.ifPresent(System.out::println); // 12
```

Terminal Operations

findAny(), findFirst()

```
// Optional<T> findAny()
// Optional<T> findFirst()
// These are terminal operations but not reductions
// as they sometimes return without processing all
// the elements in the stream. Reductions reduce the
// entire stream into one value.
Optional<String> any = Stream.of("John", "Paul")
    .findAny();
any.ifPresent(System.out::println); // John (usually)

Optional<String> first = Stream.of("John", "Paul")
    .findFirst();
first.ifPresent(System.out::println); // John
```

Terminal Operations


anyMatch(), allMatch(), noneMatch()

```
// boolean anyMatch(Predicate)
// boolean allMatch(Predicate)
// boolean noneMatch(Predicate)
List<String> names = Arrays.asList("Alan", "Brian", "Colin");
Predicate<String> pred = name -> name.startsWith("A");
System.out.println(names.stream().anyMatch(pred)); // true (one does)
System.out.println(names.stream().allMatch(pred)); // false (two don't)
System.out.println(names.stream().noneMatch(pred)); // false (one does)
```


Terminal Operations forEach()

```
// void forEach(Consumer)
// As there is no return value, forEach() is not a reduction.
// As the return type is 'void', if you want something to
// happen, it has to happen inside the Consumer (side-effect).
Stream<String> names = Stream.of("Cathy", "Pauline", "Zoe");
names.forEach(System.out::print); //CathyPaulineZoe

// Notes: forEach is also a method in the Collection interface.
//         Streams cannot be the source of a for-each loop
//         because streams do not implement the Iterable interface.
Stream<Integer> s = Stream.of(1);
for(Integer i : s){} // error: required array or Iterable
```



```
// The reduce() method combines a stream into a single object.
// It is a reduction, which means it processes all elements.
// The most common way of doing a reduction is to start with
// an initial value and keep merging it with the next value.

// T reduce(T identity, BinaryOperator<T> accumulator)
//     BinaryOperator<T> functional method:
//     T apply(T, T);
// The "identity" is the initial value of the reduction and also
// what is returned if the stream is empty. This means that there
// will always be a result and thus Optional is not the return type
// (on this version of reduce()).
// The "accumulator" combines the current result with the
// current value in the stream.
String name = Stream.of("s", "e", "a", "n")
    .filter(s -> s.length() > 2)
    .reduce("nothing", (s, c) -> s + c);
//     .reduce("", (s, c) -> s + c);
System.out.println(name); // sean

Integer product = Stream.of(2, 3, 4)
    .reduce(1, (a, b) -> a * b);
System.out.println(product); // 24
```

Terminal Operations reduce()



```
// Optional<T> reduce(BinaryOperator<T> accumulator)
// When you leave out the identity, an Optional is
// returned because there may not be any data (all the
// elements could have been filtered out earlier). There are
// 3 possible results:
//     a) empty stream => empty Optional returned
//     b) one element in stream => that element is returned
//     c) multiple elements in stream => accumulator is applied
BinaryOperator<Integer> op = (a,b) -> a+b;
Stream<Integer> empty          = Stream.empty();
Stream<Integer> oneElement     = Stream.of(6);
Stream<Integer> multipleElements = Stream.of(3, 4, 5);
empty.reduce(op).ifPresent(System.out::println);           //
oneElement.reduce(op).ifPresent(System.out::println);      // 6
multipleElements.reduce(op).ifPresent(System.out::println); // 12
// Why not just require the identity and remove this method?
// Sometimes it is nice to know if the stream is empty as opposed
// to the case where there is a value returned from the accumulator
// that happens to match the identity (however unlikely).
Integer val = Stream.of(1,1,1)
    .filter(n -> n > 5) // val is 1 this way
    .reduce(1, (a, b) -> a ); // val is 1 this way too
System.out.println(val); // 1
```

Terminal Operations reduce()

Terminal Operations

reduce()

```
// <U> U reduce (U identity,  
//             BiFunction accumulator,  
//             BinaryOperator combiner)  
// We use this version when we are dealing with different types,  
// allowing us to create intermediate reductions and then combine  
// them at the end. This is useful when working with parallel  
// streams - the streams can be decomposed and reassembled by  
// separate threads. For example, if we wanted to count the length  
// of four 1000-character strings, the first 2 values and the last  
// two values could be calculated independently. The intermediate  
// results (2000) would then be combined into a final value (4000).  
// Example: we want to count the number of characters in each String  
Stream<String> stream = Stream.of("car", "bus", "train", "aeroplane");  
int length = stream.reduce( 0, // identity  
                           (n, str) -> n + str.length(), // n is Integer  
                           (n1, n2) -> n1 + n2); // both are Integers  
System.out.println(length); // 20
```

Terminal Operations

collect()

- This is a special type of reduction called a *mutable reduction* because we use the same mutable object while accumulating. This makes it more efficient than regular reductions.
- Common mutable objects are *StringBuilder* and *ArrayList*.



Terminal Operations


collect()

- It is a really useful method as it lets us get data out of streams and into other forms e.g. *Map*'s, *List*'s and *Set*'s.
- There are two versions. We will look at one version now but later on, we will look at the more important one (the one that works with pre-defined collectors).



Terminal Operations

collect()



```
// StringBuilder collect(Supplier<StringBuilder> supplier,  
//                        BiConsumer<StringBuilder,String> accumulator  
//                        BiConsumer<StringBuilder,StringBuilder> combiner)  
// This version is used when you want complete control over  
// how collecting should work. The accumulator adds an element  
// to the collection e.g. the next String to the StringBuilder.  
// The combiner takes two collections and merges them. It is useful  
// in parallel processing.  
StringBuilder word = Stream.of("ad", "jud", "i", "cate")  
    .collect(() -> new StringBuilder(),           // StringBuilder::new  
            (sb, str) -> sb.append(str),          // StringBuilder::append  
            (sb1, sb2) -> sb1.append(sb2));       // StringBuilder::append  
System.out.println(word); // adjudicate
```

A background image of four students in a library. A young woman with long dark hair is on the left, smiling. Next to her is a young man with dark hair, also smiling. To his right is a young woman with glasses and dark hair, looking towards the right. On the far right is a young man with dark hair, seen from the back/side, looking towards the others. They are gathered around a table with a laptop and some papers. Bookshelves filled with books are in the background.

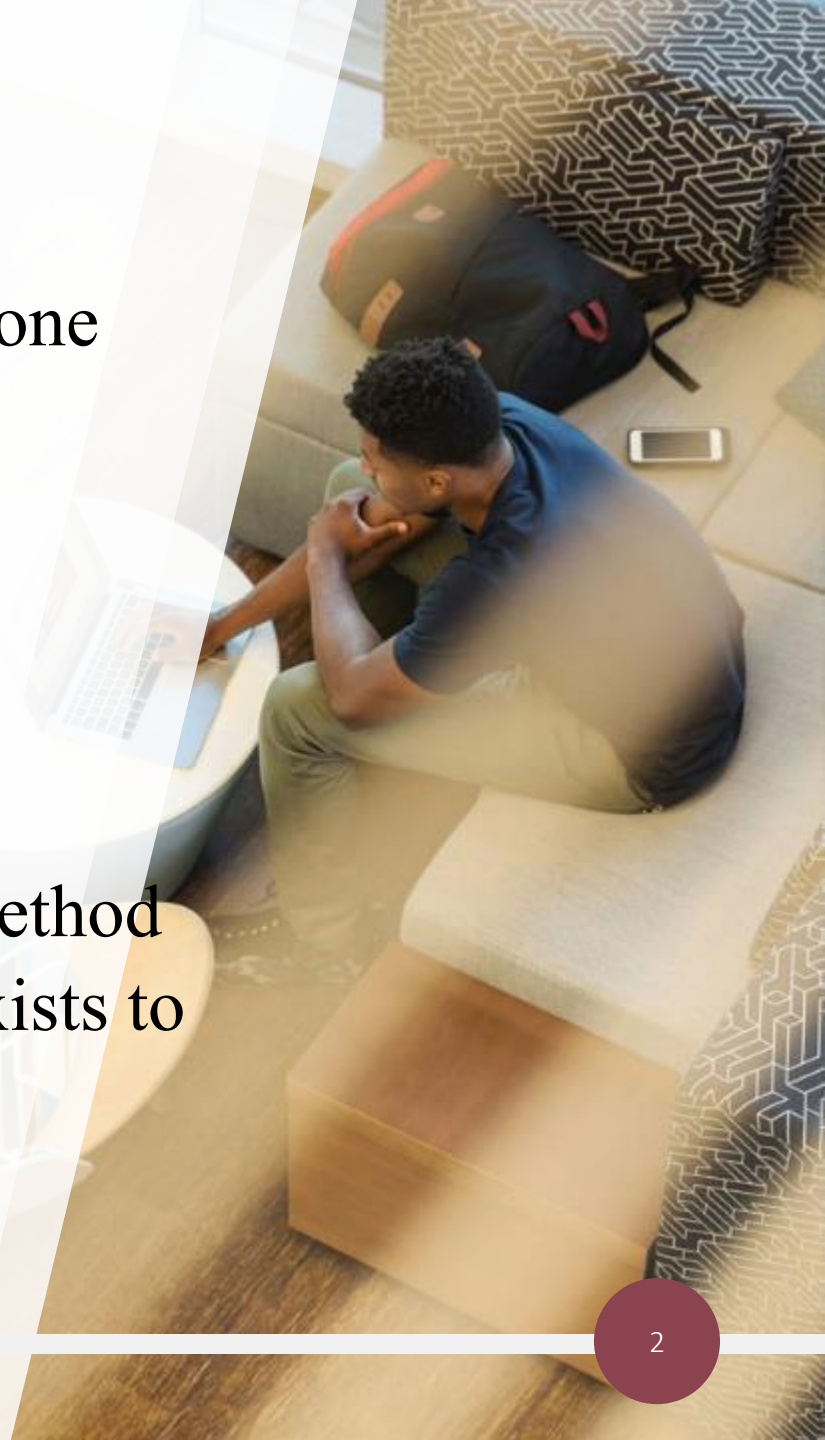
Streams

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]}
```


A photograph of four students in a library setting. A young man in a grey t-shirt is smiling and looking at a laptop. A young woman with glasses is looking at the laptop. Another young woman is looking at a book. A young man is looking at the laptop. They are all sitting at a table. The background is filled with bookshelves. There are semi-transparent blue and red geometric shapes overlaid on the image.

Streams

Intermediate Operations

Intermediate Operations

filter()

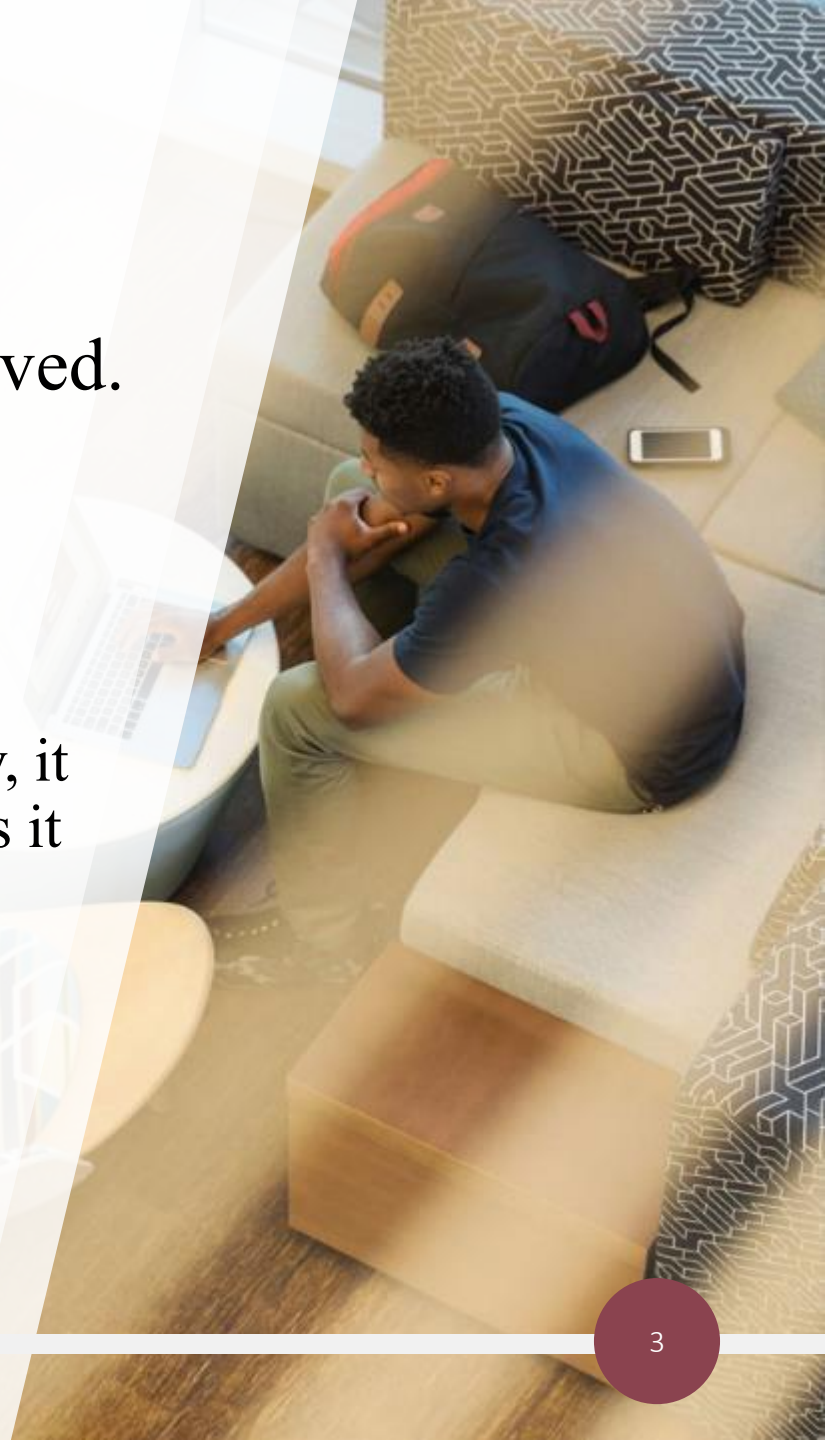
- Unlike a terminal operation, an intermediate operation produces a stream as a result.

```
// 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
```


Intermediate Operations

`distinct()`

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



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));
```

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));
```

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
```


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
```

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.



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
```



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

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



Intermediate Operations sorted(Comparator)

```
// 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);
```



A photograph of four students in a library setting. A young woman with long dark hair is on the left, smiling. Next to her is a young man with dark hair, also smiling. To his right is a young woman with glasses and a dark polka-dot top, looking at a laptop. On the far right is a young man with dark hair, seen from the back/side, looking at the laptop. They are all gathered around a table with a laptop, books, and papers. The background is filled with bookshelves. A semi-transparent blue diagonal band runs across the image, and a semi-transparent red horizontal band is at the bottom.

Streams

Primitive Streams

Primitive Streams

- As opposed to `Stream<T>` e.g. `Stream<Integer>`, `Stream<Double>` and `Stream<Long>`; Java actually provides other stream classes that you can use to work with primitives:

- *IntStream* – for primitive types *int*, *short*, *byte* and *char*
- *DoubleStream* – for primitive types *double* and *float*
- *LongStream* – for primitive type *long*

- *IntStream* `Stream<Integer>`
- *DoubleStream* `Stream<Double>`
- *LongStream* `Stream<Long>`



Creating Primitive Streams

```
int[] ia    = {1,2,3};
double[] da = {1.1, 2.2, 3.3};
long[] la   = {1L, 2L, 3L};

IntStream iStream1    = Arrays.stream(ia);
DoubleStream dStream1 = Arrays.stream(da);
LongStream lStream1   = Arrays.stream(la);
System.out.println(iStream1.count() + ", " +
                   dStream1.count() + ", " + lStream1.count()); // 3, 3, 3

IntStream iStream2    = IntStream.of(1, 2, 3);
DoubleStream dStream2 = DoubleStream.of(1.1, 2.2, 3.3);
LongStream lStream2    = LongStream.of(1L, 2L, 3L);
System.out.println(iStream2.count() + ", " +
                   dStream2.count() + ", " + lStream2.count()); // 3, 3, 3
```

Primitive Streams

- The primitive streams, in addition to containing many of the *Stream* methods, also contain specialised methods for working with numeric data.
- The primitive streams know how to perform certain common operations automatically e.g. *min()*, *max()*, *sum()* and *average()*.



```
// 1. Using Stream<T> and reduce(identity, accumulator)
Stream<Integer> numbers = Stream.of(1,2,3);
// Integer reduce(Integer identity, BinaryOperator accumulator)
//   BinaryOperator extends BiFunction<T,T,T>
//     T apply(T,T)
// starting the accumulator with 0
//     n1 + n2
//     0 + 1 == 1 (n1 now becomes 1)
//     1 + 2 == 3 (n1 now becomes 3)
//     3 + 3 == 6
System.out.println(numbers.reduce(0, (n1, n2) -> n1 + n2)); // 6

// 2. Using IntStream and sum()
// IntStream mapToInt(ToIntFunction)
//   ToIntFunction is a functional interface:
//     int applyAsInt(T value);
IntStream intS = Stream.of(1,2,3)
    .mapToInt(n -> n); // unboxed
int total = intS.sum();
System.out.println(total); // 6
```


method		primitive stream
OptionalDouble	average()	IntStream
		LongStream
		DoubleStream
OptionalInt	max()	IntStream
OptionalLong		LongStream
OptionalDouble		DoubleStream
OptionalInt	min()	IntStream
OptionalLong		LongStream
OptionalDouble		DoubleStream
int	sum()	IntStream
long	sum()	LongStream
double	sum()	DoubleStream



```
OptionalInt max = IntStream.of(10, 20, 30)
    .max(); // terminal operation
max.ifPresent(System.out::println); // 30

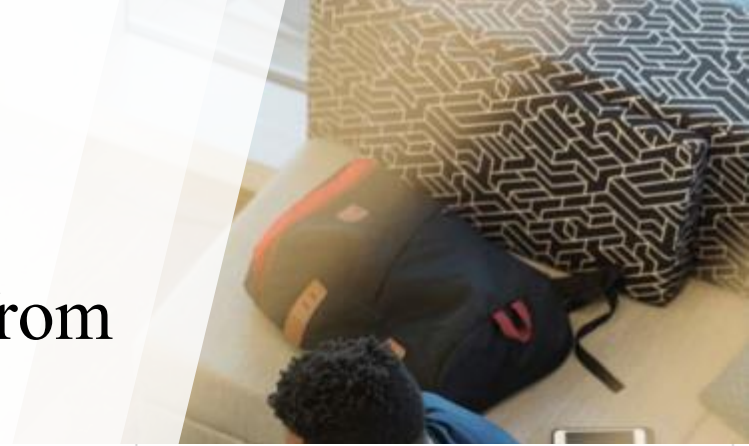
OptionalDouble min = DoubleStream.of(10.0, 20.0, 30.0)
    .min(); // terminal operation
// NoSuchElementException is thrown if no value present
System.out.println(min.orElseThrow()); // 10.0

OptionalDouble average = LongStream.of(10L, 20L, 30L)
    .average(); // terminal operation
System.out.println(average.orElseGet(() -> Math.random())); // 20.0
```

```
stats(IntStream.of(5, 10, 15, 20));
stats(IntStream.empty());
}
public static void stats(IntStream numbers) {
    IntSummaryStatistics intStats =
        numbers.summaryStatistics(); // terminal op.
    int min = intStats.getMin();
    System.out.println(min); // 5 (2147483647 if nothing in stream)
    int max = intStats.getMax();
    System.out.println(max); // 20 (-2147483648 if nothing in stream)
    double avg = intStats.getAverage();
    System.out.println(avg); // 12.5 (0.0 if nothing in stream)
    long count = intStats.getCount();
    System.out.println(count); // 4 (0 if nothing in stream)
    long sum = intStats.getSum();
    System.out.println(sum); // 50 (0 if nothing in stream)
}
```


Supplier<T>	T get()		Function<T, R>	R apply(T)
DoubleSupplier	double getAsDouble()		BiFunction<T,U,R>	R apply(T, U)
IntSupplier	int getAsInt()		DoubleFunction<R>	R apply(double)
LongSupplier	long getAsLong()		IntFunction<R>	R apply(int)
Consumer<T>	void accept(T)		LongFunction<R>	R apply(long)
BiConsumer<T, U>	void accept(T, U)		UnaryOperator<T>	T apply(T)
DoubleConsumer	void accept(double)		BinaryOperator<T>	T apply(T, T)
IntConsumer	void accept(int)		DoubleUnaryOperator	double applyAsDouble(double)
LongConsumer	void accept(long)		IntUnaryOperator	int applyAsInt(int)
Predicate<T>	boolean test(T)		LongUnaryOperator	long applyAsLong(long)
BiPredicate<T,U>	boolean test(T, U)		DoubleBinaryOperator	double applyAsDouble(double, double)
DoublePredicate	boolean test(double)		IntBinaryOperator	int applyAsInt(int, int)
IntPredicate	boolean test(int)		LongBinaryOperator	long applyAsLong(long, long)
LongPredicate	boolean test(long)			

Mapping Streams



- Another way to create a primitive stream is by mapping from another stream type.

Source stream class	To create <i>Stream<T></i>	To create <i>DoubleStream</i>	To create <i>IntStream</i>	To create <i>LongStream</i>
Stream<T>	map(Function<T,R>) R apply(T value)	mapToDouble(ToDoubleFunction<T>) double applyAsDouble(T value)	mapToInt(ToIntFunction<T>) int applyAsInt(T value)	mapToLong(ToLongFunction<T>) long applyAsLong(T value)
DoubleStream	mapToObj(DoubleFunction<R>) R apply(double value)	map(DoubleUnaryOperator) double applyAsDouble(double)	mapToInt(DoubleToIntFunction) int applyAsInt(double)	mapToLong(DoubleToLongFunction) long applyAsLong(double)
IntStream	mapToObj(IntFunction<R>) R apply(int value)	mapToDouble(IntToDoubleFunction) double applyAsDouble(int)	map(IntUnaryOperator) int applyAsInt(int)	mapToLong(IntToLongFunction) long applyAsLong(int)
LongStream	mapToObj(LongFunction<R>) R apply(long value)	mapToDouble(LongToDoubleFunction) double applyAsDouble(long)	mapToInt(LongToIntFunction) int applyAsInt(long)	map(LongUnaryOperator) long applyAsLong(long)

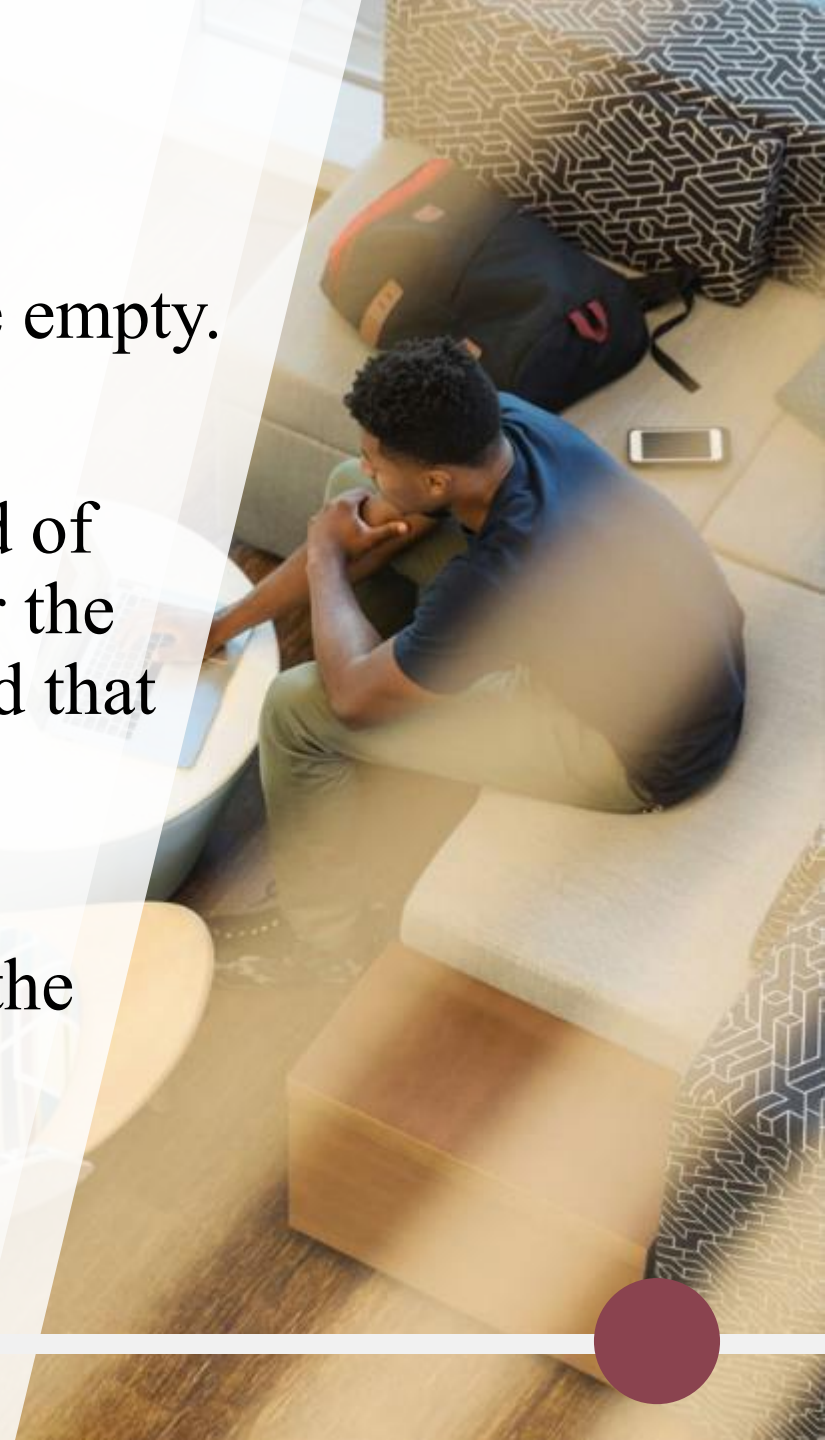
A photograph of four students in a library setting. A young man in a grey t-shirt is smiling and looking at a laptop. A young woman with glasses is looking at the laptop. Another young woman is looking at a book. A young man is looking at the laptop. The background is filled with bookshelves. There are semi-transparent geometric overlays: a large blue triangle on the left and a red triangle on the right.

Streams

Optionals

Optionals

- Think of an *Optional* as a box that may or may not be empty.
- Before Java 8, programmers would return *null* instead of *Optional*. Returning an *Optional* is now a clear way for the API to state that there might not be a value in there (and that the programmer must deal with that).
- In addition to *Optional*<*T*>, there are *Optional*'s for the primitive types:
 - *OptionalDouble*, *OptionalInt* and *OptionalLong*



```
// a long way to calculate average (just for showing Optional)
public static Optional<Double> calcAverage(int... scores){
    if(scores.length == 0) return Optional.empty();
    int sum=0;
    for(int score:scores) sum += score;
    return Optional.of((double)sum / scores.length);
}
```

```
Optional<Double> optAvg = calcAverage(50, 60, 70);
// if you do a get() and the Optional is empty you get:
//    NoSuchElementException: No value present
// boolean isPresent() protects us from that.
if(optAvg.isPresent()){
    System.out.println(optAvg.get()); // 60.0
}
// void ifPresent(Consumer c)
optAvg.ifPresent(System.out::println); // 60.0
// T orElse(T t)
System.out.println(optAvg.orElse(Double.NaN)); // 60.0

Optional<Double> optAvg2 = calcAverage(); // will return an empty Optional
System.out.println(optAvg2.orElse(Double.NaN)); // NaN
// T orElseGet(Supplier<T> s)
System.out.println(optAvg2.orElseGet(() -> Math.random())); // 0.8524556508038182
```



```
public static void doOptionalNull() {  
    Optional<String> optSK = howToDealWithNull("SK");  
    optSK.ifPresent(System.out::println); // SK  
    Optional<String> optNull = howToDealWithNull(null);  
    System.out.println(  
        optNull.orElseGet(  
            () -> "Empty optional")); // Empty optional  
}  
  
public static Optional<String> howToDealWithNull(String param) {  
    // Optional optReturn = param == null ? Optional.empty() : Optional.of(param);  
    Optional optReturn = Optional.ofNullable(param); // same as previous line  
    return optReturn;  
}
```

```
public static void doOptionalPrimitiveAverage() {  
    OptionalDouble optAvg = IntStream.rangeClosed(1, 10)  
                                     .average();  
    // DoubleConsumer - functional interface; functional method is:  
    //     void accept(double value)  
    optAvg.ifPresent((d) -> System.out.println(d)); // 5.5  
    System.out.println(optAvg.getAsDouble()); // 5.5  
    // DoubleSupplier - functional interface; functional method is:  
    //     double getAsDouble()  
    System.out.println(optAvg.orElseGet(() -> Double.NaN)); // 5.5  
}
```


A photograph of four students in a library setting. A young woman with long dark hair is on the left, smiling. Next to her is a young man with dark hair, also smiling. To his right is a young woman with glasses and dark hair, looking towards the right. On the far right is a young man with dark hair, seen from the back/side, looking at a laptop. They are all gathered around a table with books and a laptop. The background is filled with bookshelves. A semi-transparent blue diagonal band runs across the image, and a semi-transparent red horizontal band is at the bottom.

Streams

Parallel Streams

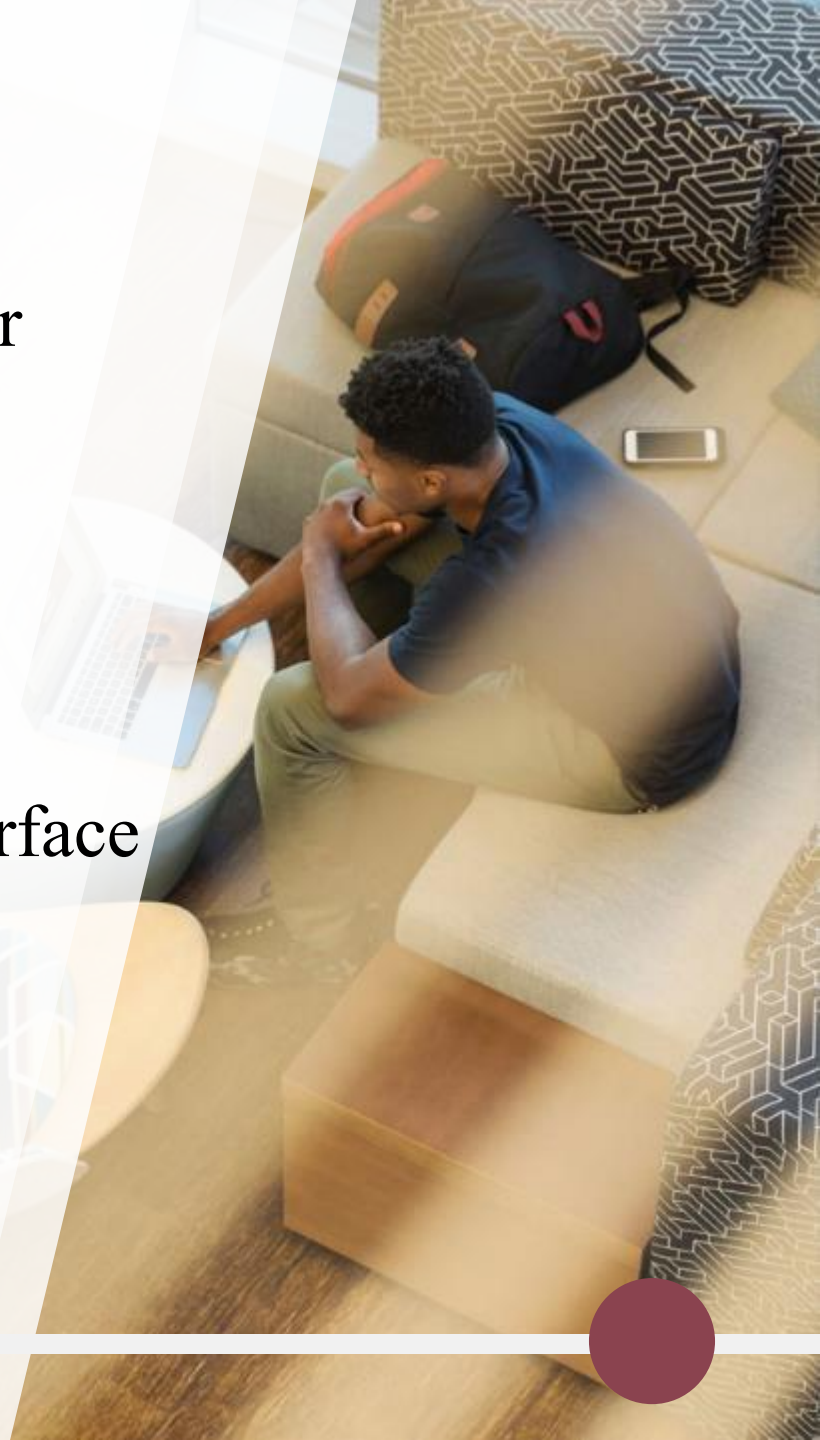
Parallel Streams

- All of the streams thus far have been sequential streams i.e. the streams have processed the data one element at a time.
- Parallel streams can process elements in a stream concurrently i.e. at the same time.
- Java achieves this by splitting the stream up into sub-streams and then the pipeline operations are performed on the sub-streams concurrently (each sub-stream has its own thread).



Parallel Streams

- To make a stream parallel, we can use the *parallel()* or *parallelStream()* methods.
- *parallel()* is available in *Stream<T>*.
- *parallelStream()* is defined in the *Collection<E>* interface





```
Stream<String> animalsStream = List.of("sheep", "pigs", "horses")  
                                   .parallelStream();
```

Collection<E>

```
Stream<String> animalsStream = Stream.of("sheep", "pigs", "horses")  
                                   .parallel();
```

Stream<T>

Parallel Streams

- Firstly, let's look at a sequential stream that sums up a stream of numbers.

```
// Sequential stream
int sum = Stream.of(10, 20, 30, 40, 50, 60)
    // IntStream has the sum() method so we use
    // the mapToInt() method to map from Stream<Integer>
    // to an IntStream (i.e. a stream of int primitives).
    // IntStream mapToInt(ToIntFunction)
    //     ToIntFunction is a functional interface:
    //         int applyAsInt(T value)
    //         .mapToInt(n -> n.intValue())
    //         .mapToInt(Integer::intValue)
    //         .mapToInt(n -> n)
    .sum();

System.out.println("Sum == "+sum); // 210
```

Sequential stream



```
int sum = Stream.of(10, 20, 30, 40, 50, 60)
                .parallel()
                .mapToInt(Integer::intValue)
                .sum();
System.out.println("Sum == "+sum); // 210
```

Parallel stream

Parallel Streams

- What is happening in the background?

Sequential stream

10	20	30	40	50	60
	30	30	40	50	60
		60	40	50	60
			100	50	60
				150	60
					210

Parallel stream

Thread 1	Thread 2
10 20 30	40 50 60
30 30	90 60
60	150
210	



Parallel Streams

- Be careful if order is important, as the order of thread completion will determine the final result (not the order in the original collection).

```
public static void sequentialStream() {  
    Arrays.asList("a", "b", "c") // create List  
        .stream() // stream the List  
        .forEach(System.out::print); // abc  
}  
  
public static void parallelStream() {  
    Arrays.asList("a", "b", "c") // create List  
        .stream() // stream the List  
        .parallel()  
        .forEach(System.out::print); // bca  
}
```

