

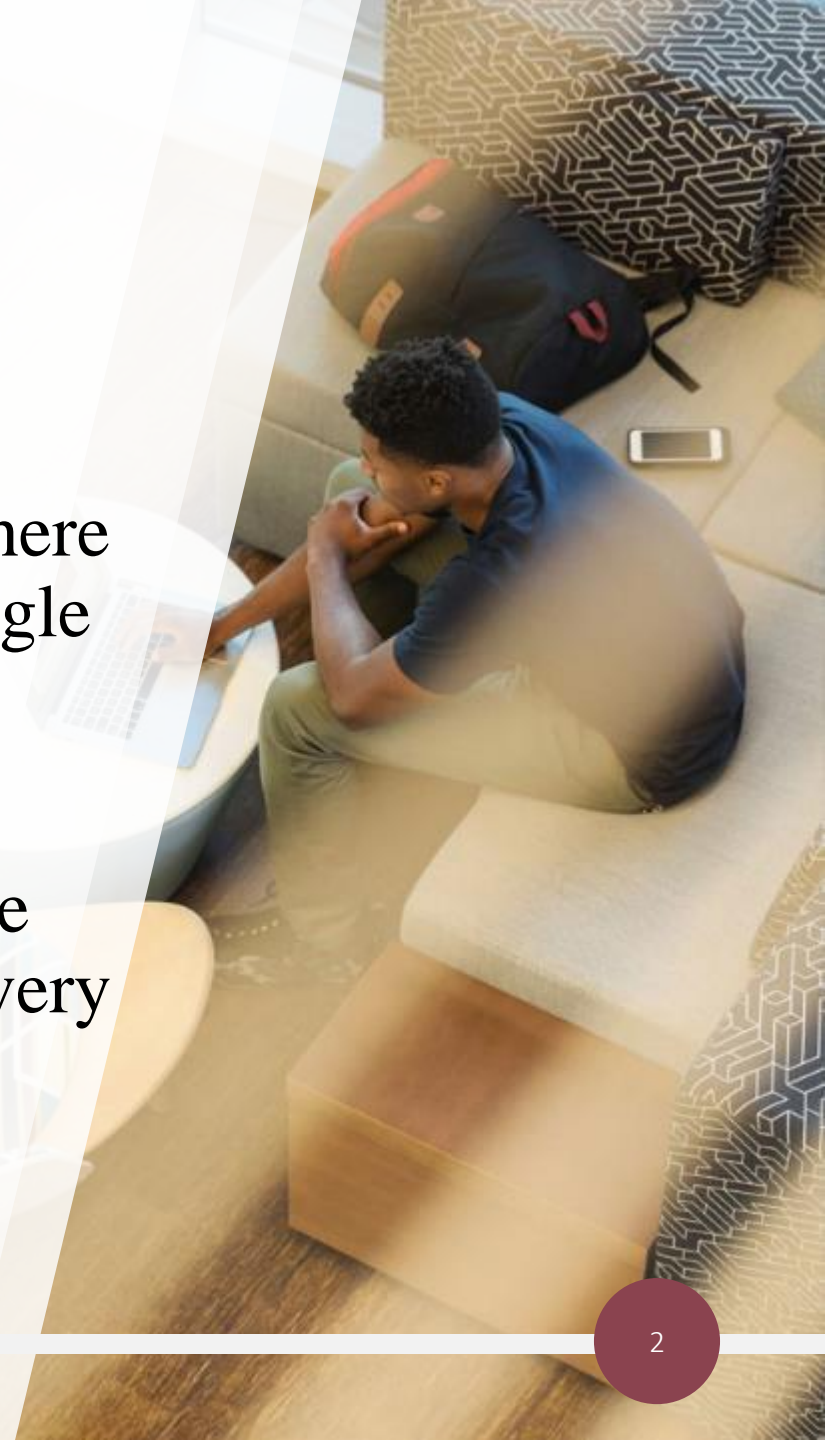
A photograph of four students in a library setting. A young man in a grey t-shirt is smiling and looking at a laptop. A young woman with glasses is looking at the laptop. Another young woman is looking at a book. A young man is looking at the laptop. The background is filled with bookshelves. There are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

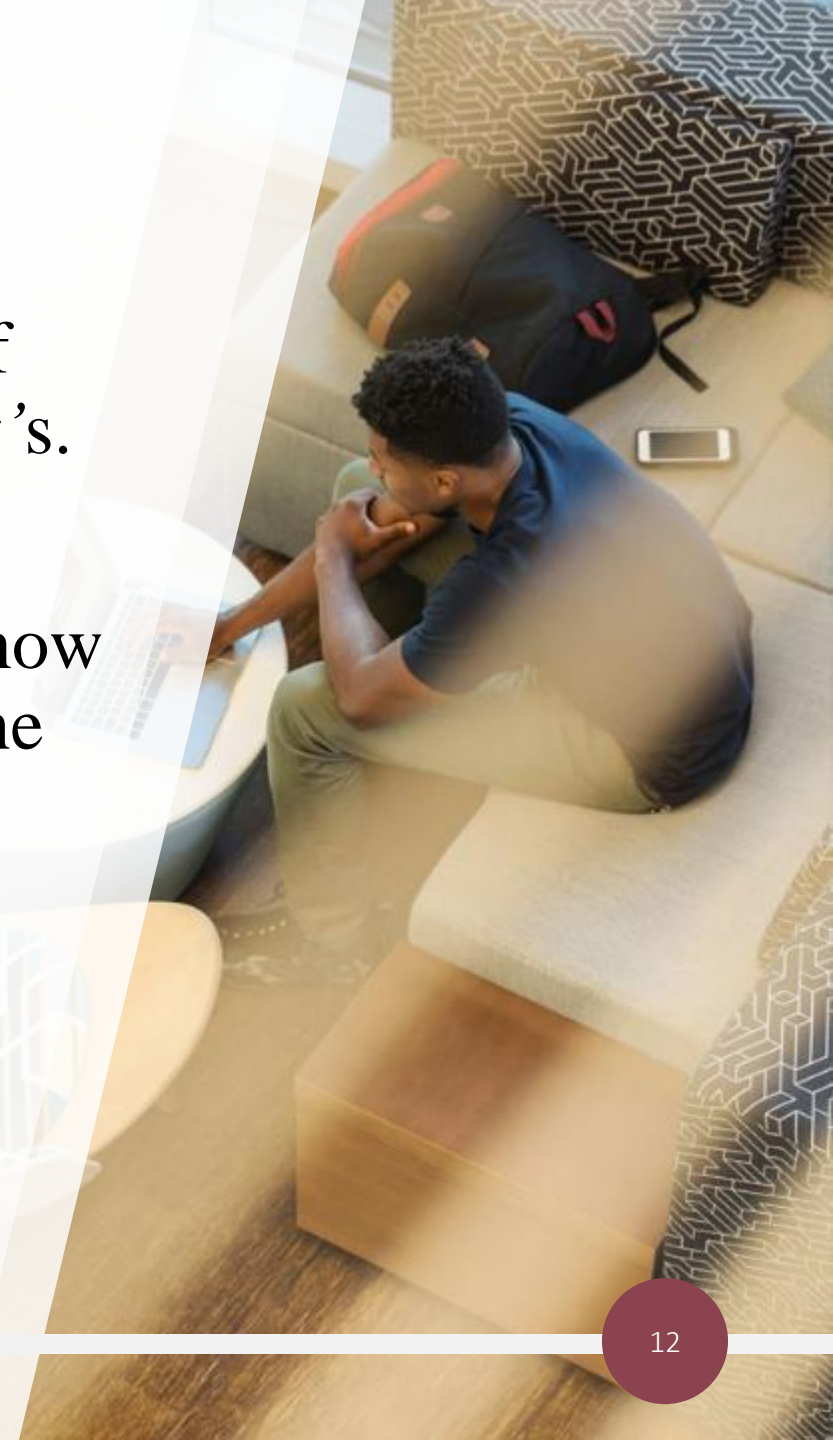
- This is a special type of reduction called a *mutable reduction* because we use the same mutable object while accumulating. This makes it more efficient than regular reductions.
- Common mutable objects are *StringBuilder* and *ArrayList*.



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
```