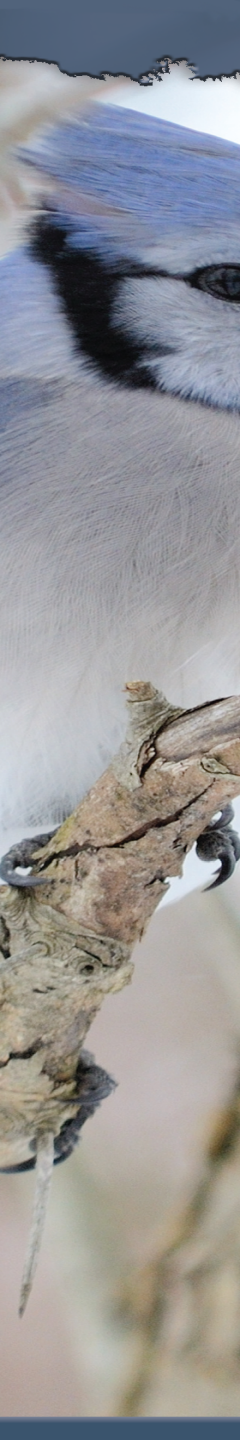
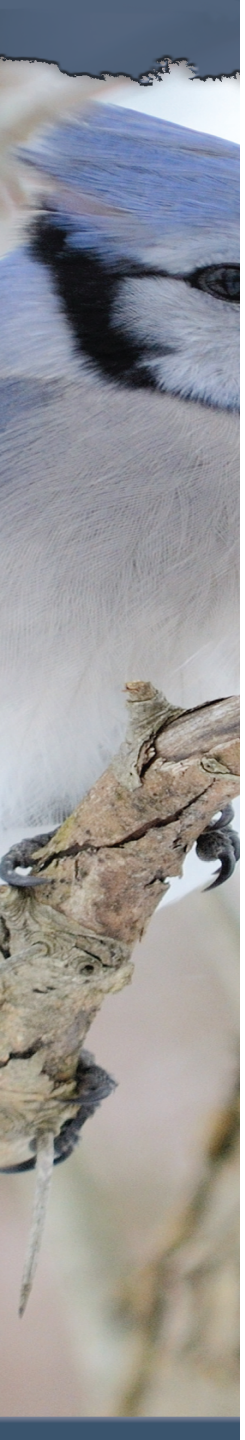
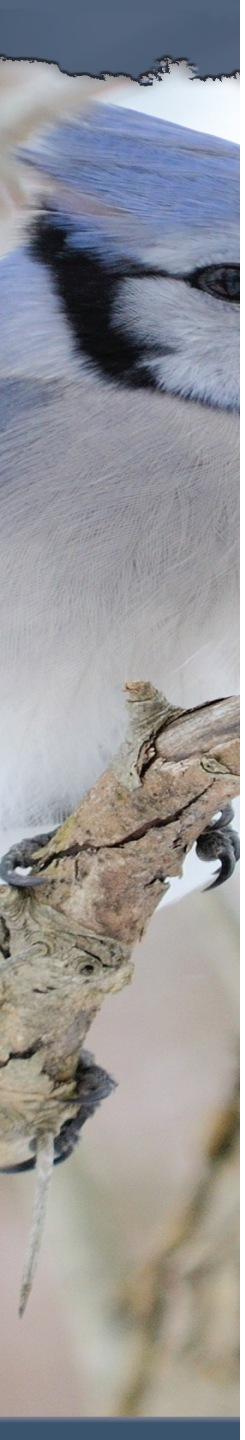


Functional Processing of Collections



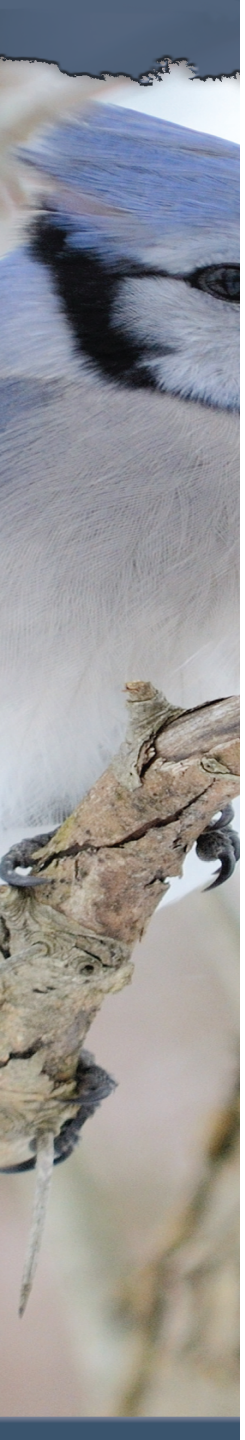
Overview

- An alternative look at collections and iteration.
- A *functional* style of programming.
- Complements the imperative style used so far.
- Streams.
- Lambda notation.



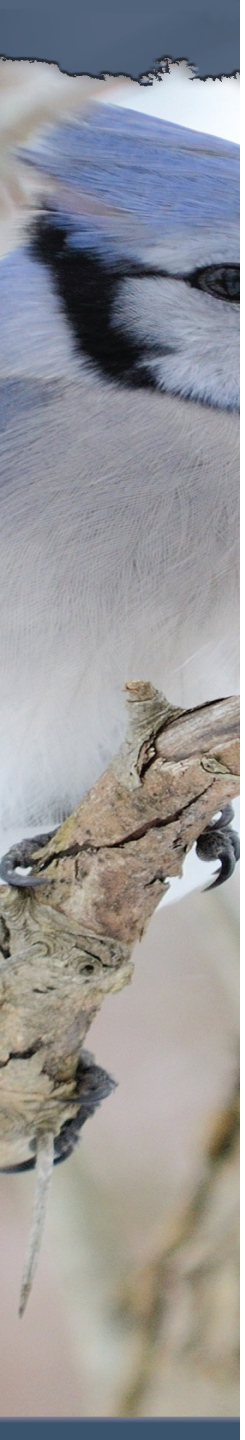
First introduced in Java 8

- Lambdas borrow well-established techniques from the world of functional languages, such as Lisp, Haskell, Erlang, etc.
- Lambdas require additional syntax in the language.
- Stream operations provide an alternative means of implementing tasks associated with iteration over collections.
- Some existing library classes have been retro-fitted to support streams and lambda.
- Streams often involve multi-stage processing of data in the form of a *pipeline* of operations.



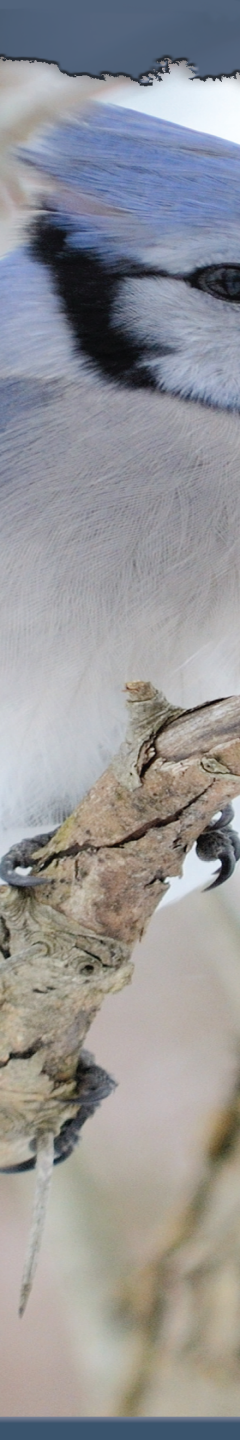
Lambda functions

- Bear a strong similarity to simple methods.
- They have:
  - A return type.
  - Parameters.
  - A body.
- They don’t have a name (anonymous methods).
- They have no associated object.
- They can be passed as parameters:
  - As *code* to be executed by the receiving method.



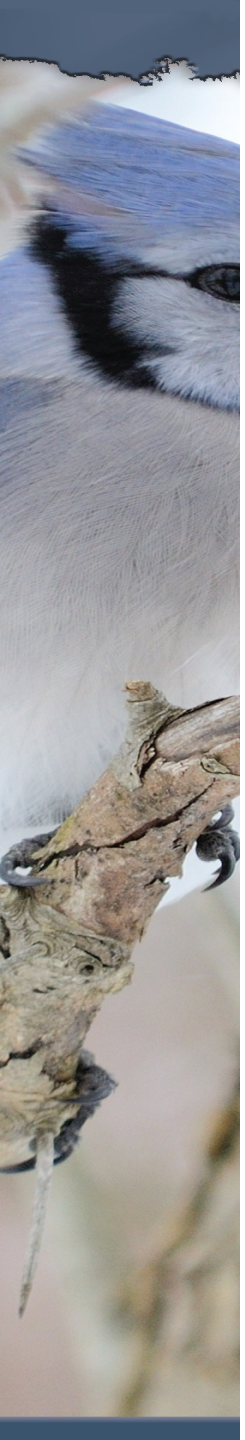
Example scenario

- Animal monitoring in a national park (*animal-monitoring* project).
- Spotters send back reports of animals they have seen (**`Sighting`** objects).
- Base collates sighting reports to check on population levels.
- Review version 1 of the project, which is implemented in a familiar (imperative) style:
  - The **`AnimalMonitoring`** class has methods to:
    - List all sighting records;
    - List sightings of a particular animal;
    - Identify animals that could be endangered;
    - Calculate sighting totals;
    - Etc.



Method and *lambda function* equivalent

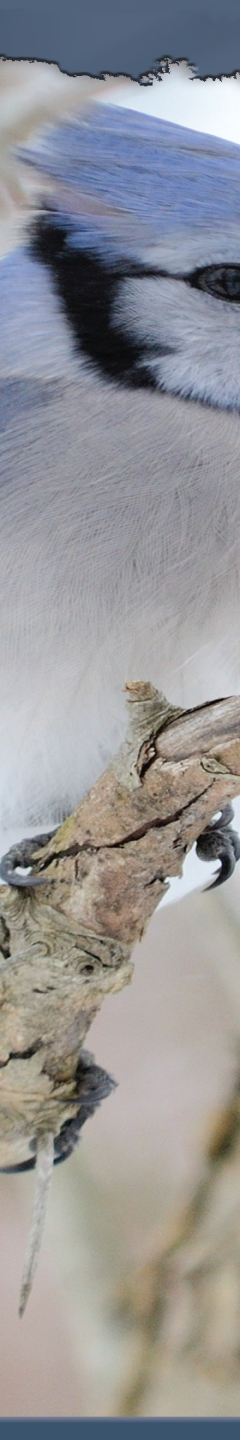
```
void printSighting(Sighting record) {
    System.out.println(record.getDetails());
}
```



Method and *lambda function* equivalent

```
void printSighting(Sighting record) {
    System.out.println(record.getDetails());
}
```

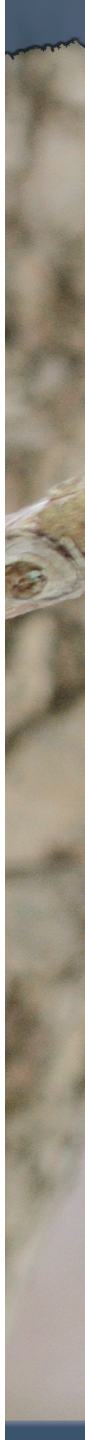
```
(Sighting record) → {
    System.out.println(record.getDetails());
}
```

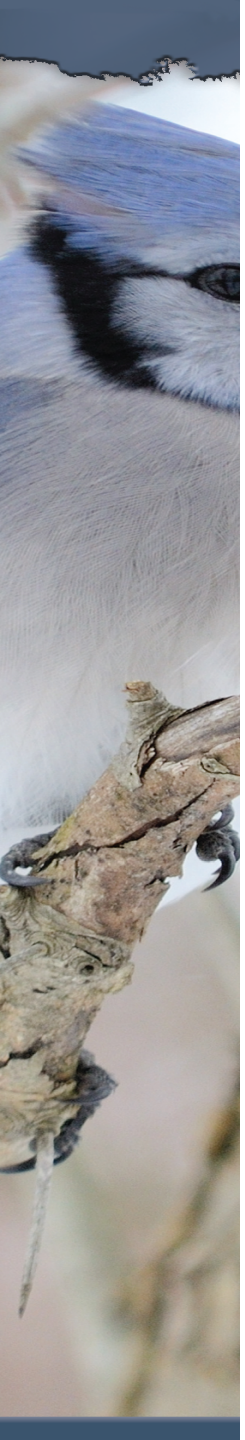


Processing a collection – the usual approach

```
loop (for each element in the collection):
    get one element;
    do something with the element;
end loop
```

```
for (Sighting record : sightings) {
    printSighting(record);
}
```

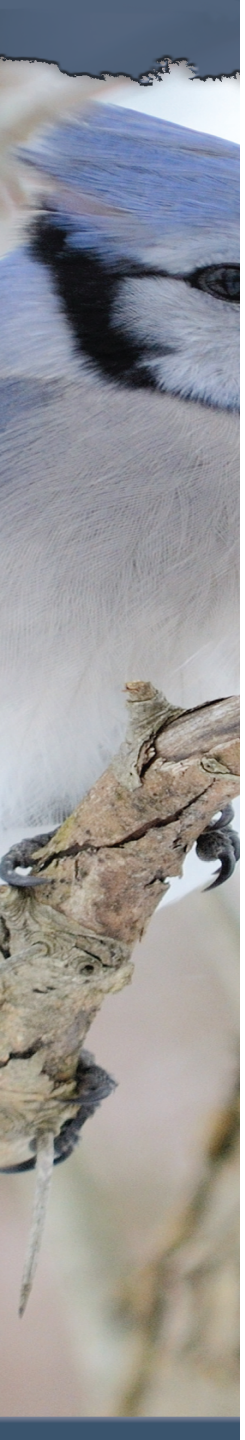





Processing a collection – a functional approach

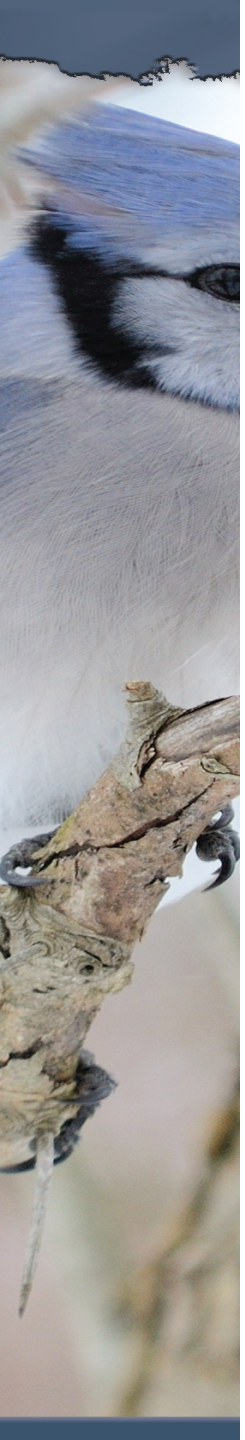
```
collection.doThisForEachElement(some code);
```

```
sightings.forEach((Sighting record) → {
    System.out.println(record.getDetails());
});
```



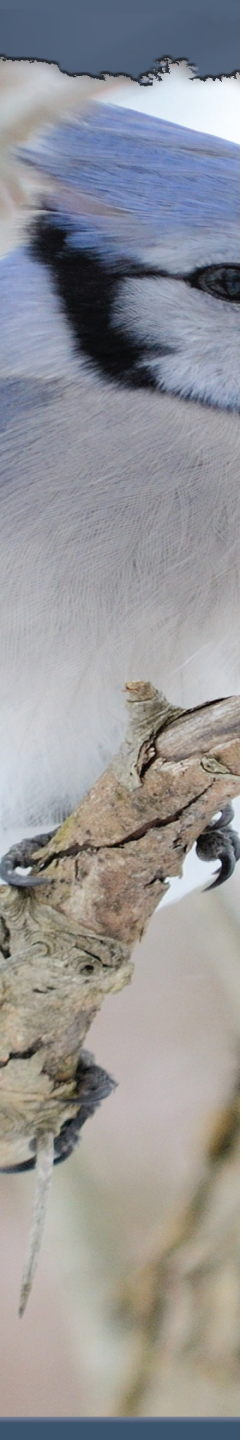
Reduced lambda syntax: infer type

```
sightings.forEach((record) → {
    System.out.println(record.getDetails());
});
```



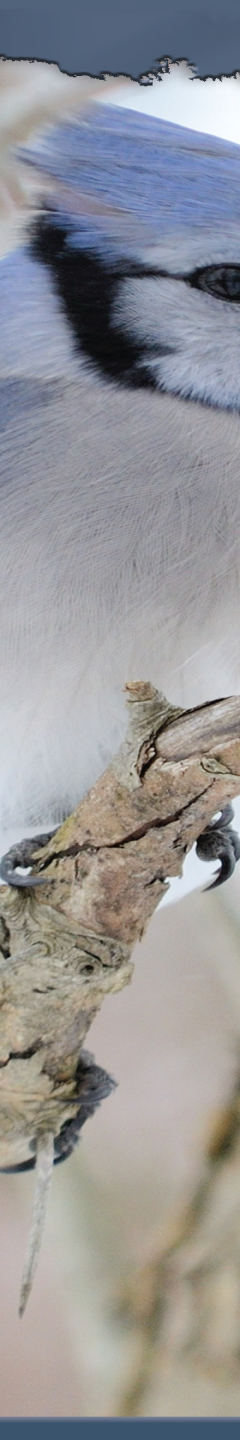
Reduced lambda syntax: single parameter

```
sightings.forEach(record → {
    System.out.println(record.getDetails());
});
```



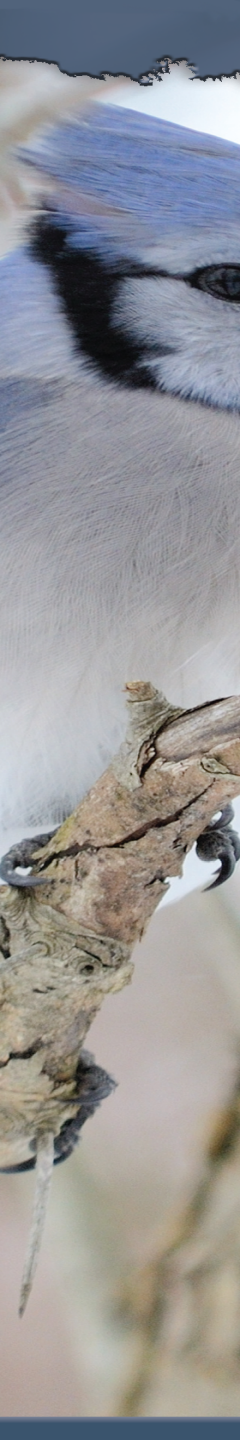
Reduced lambda syntax: single statement

```
sightings.forEach(
    record -> System.out.println(record.getDetails())
);
```



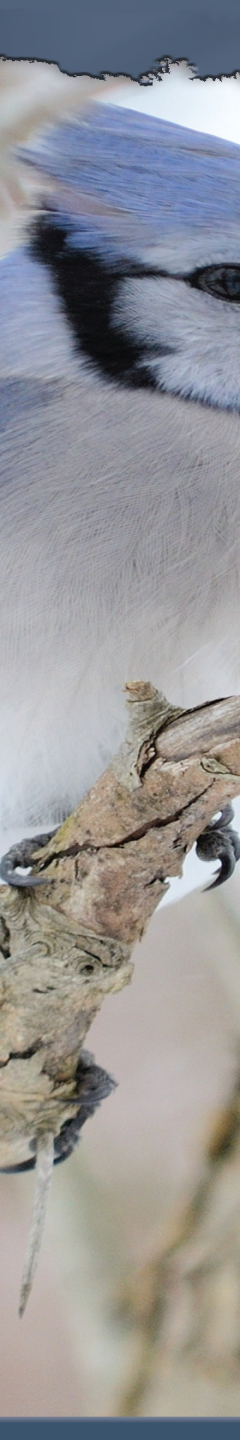
Loop vs lambda syntax: ask vs tell

- Loop is asking: I'm asking you to give me each element of your collection in turn and I'll do something with it.
- Lambda is telling: I'm giving you a function and telling you apply it to each element of your collection.
  - I don't care how you apply it – that's your responsibilty.
- Telling is more object-oriented.



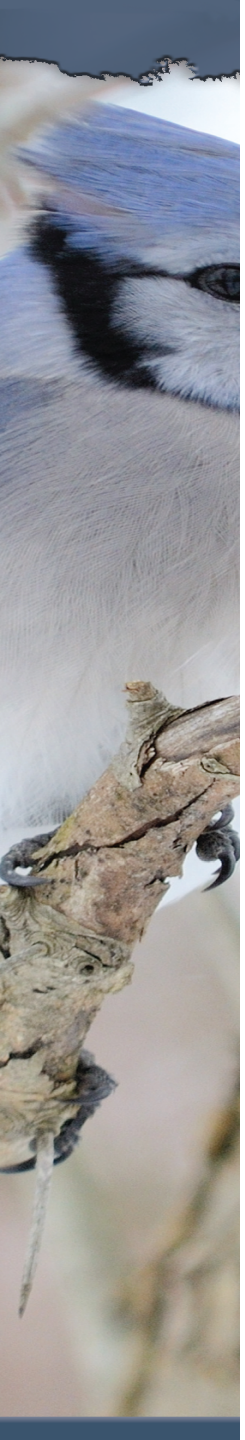
Streams

- Streams are often created from the contents of a collection.
- Elements in a stream can be processed in parallel.
- Interesting and useful.
- ...but unfortunately no time this year.



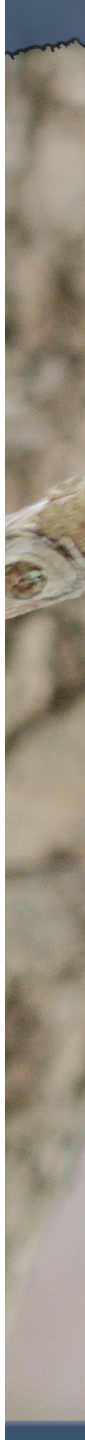
Streams

- Streams are often created from the contents of a collection.
- An **`ArrayList`** is not a stream, but its **`stream`** method creates a stream of its contents.
- Elements in a stream are not accessed via an index, but usually sequentially.
- The contents and ordering of the stream cannot be changed – changes require the creation of a new stream.
- A stream could potentially be infinite!
- Elements in a stream can be processed in parallel.



Streams example

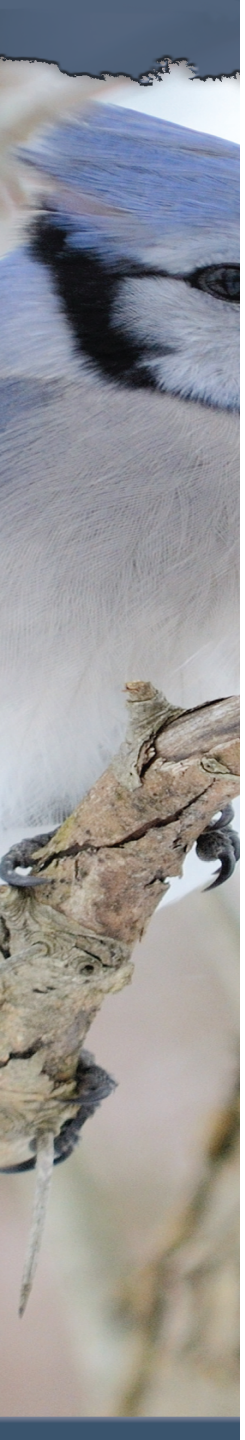
```
List<String> l = Arrays.asList(new String[]{"A", "B", "C", "D"});
```



Streams example

```
List<String> l = Arrays.asList(new String[]{"A", "B", "C", "D"});
```

```
// ask for an element and then do something with it
for (int i = 0; i < l.size(); i++) {
    System.out.print(l.get(i) + " ");
}
==> A B C D
```

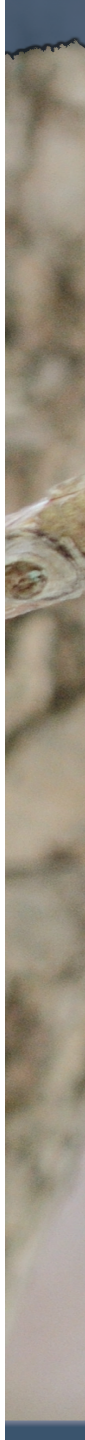


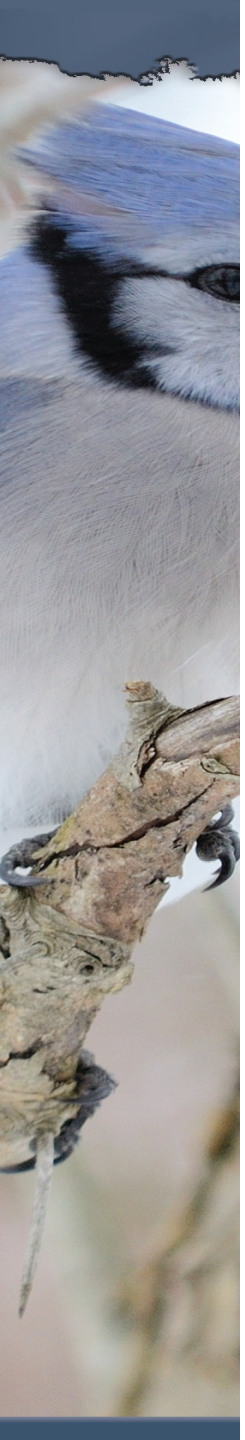
Streams example

```
List<String> l = Arrays.asList(new String[]{"A", "B", "C", "D"});
```

```
// ask for an element and then do something with it
for (int i = 0; i < l.size(); i++) {
    System.out.print(l.get(i) + " ");
}
==> A B C D
```

```
// tell the collection what to do with each element
l.forEach(c -> System.out.print(c + " "));
==> A B C D
```



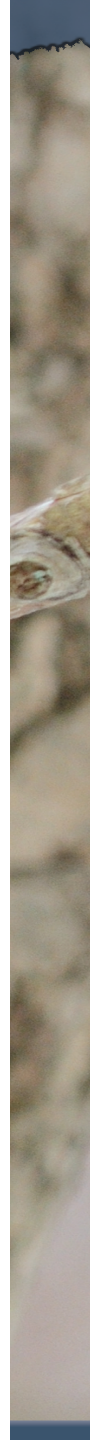
Streams example

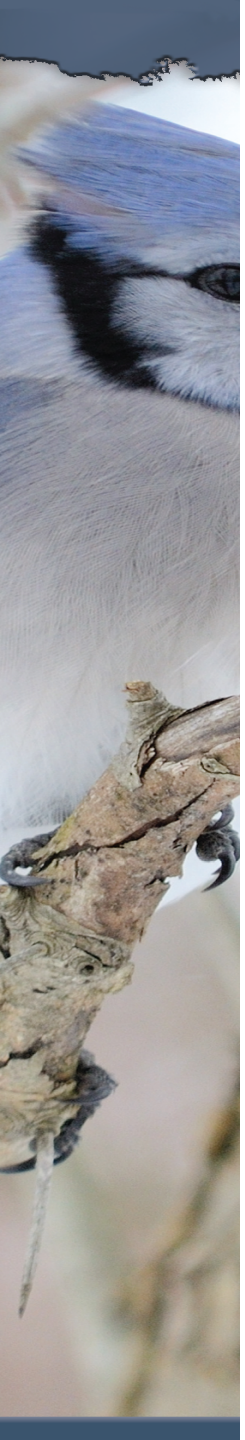
```
List<String> l = Arrays.asList(new String[]{"A", "B", "C", "D"});
```

```
// ask for an element and then do something with it
for (int i = 0; i < l.size(); i++) {
    System.out.print(l.get(i) + " ");
}
==> A B C D
```

```
// tell the collection what to do with each element
l.forEach(c -> System.out.print(c + " "));
==> A B C D
```

```
// stream the collection and filter its elements
l.stream().filter(c -> !c.equals("B"))
    .forEach(c -> System.out.print(c + " "));
==> A C D
```

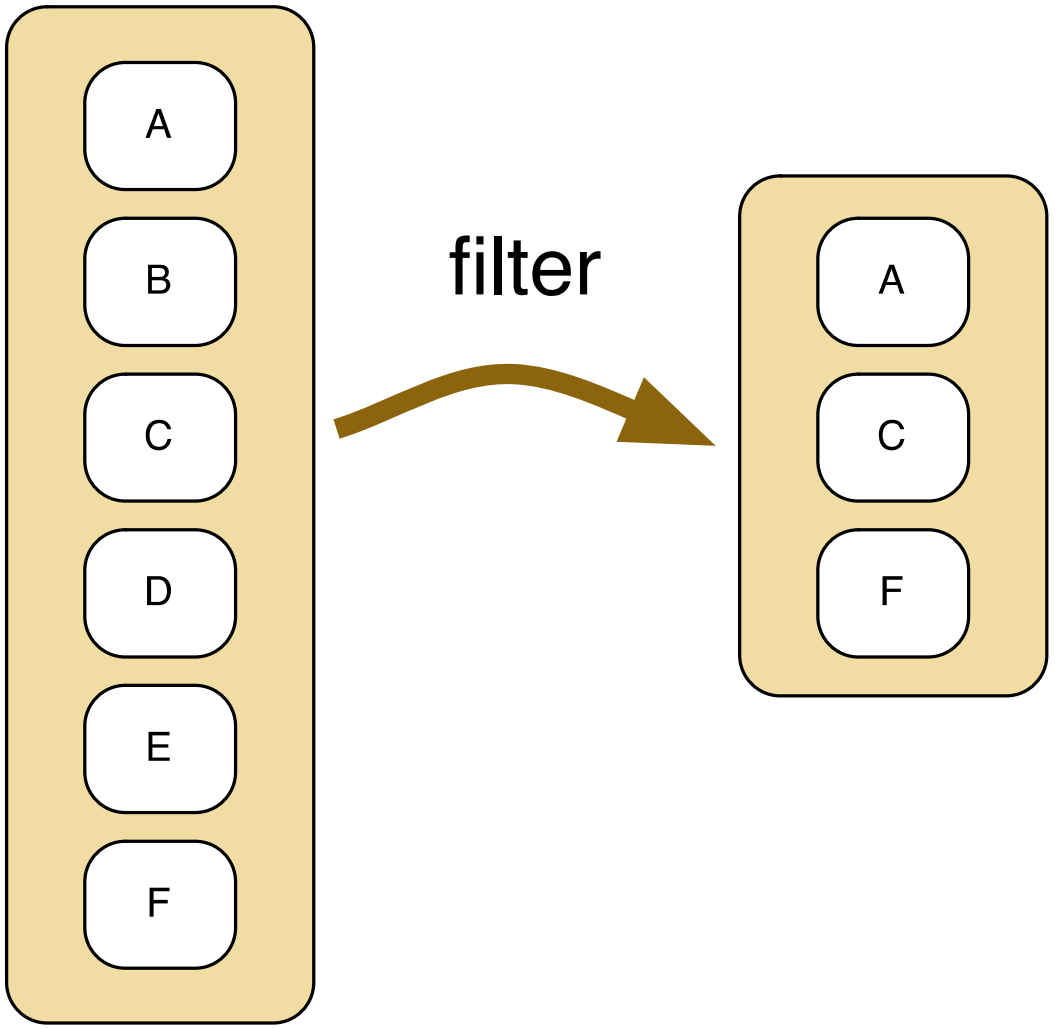



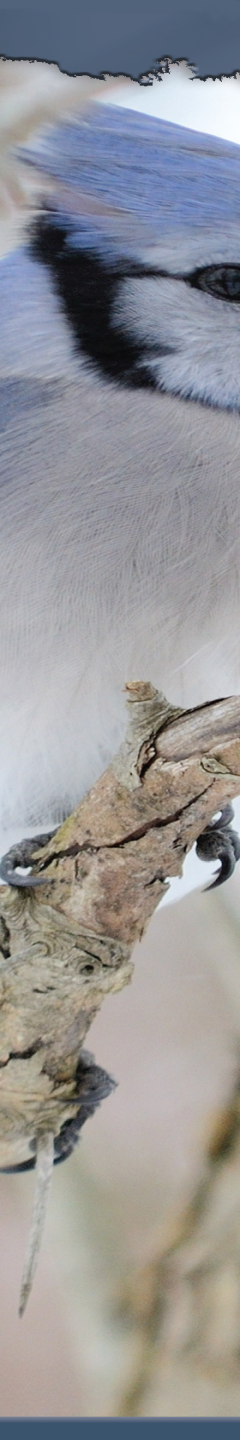


Filters, maps and reductions

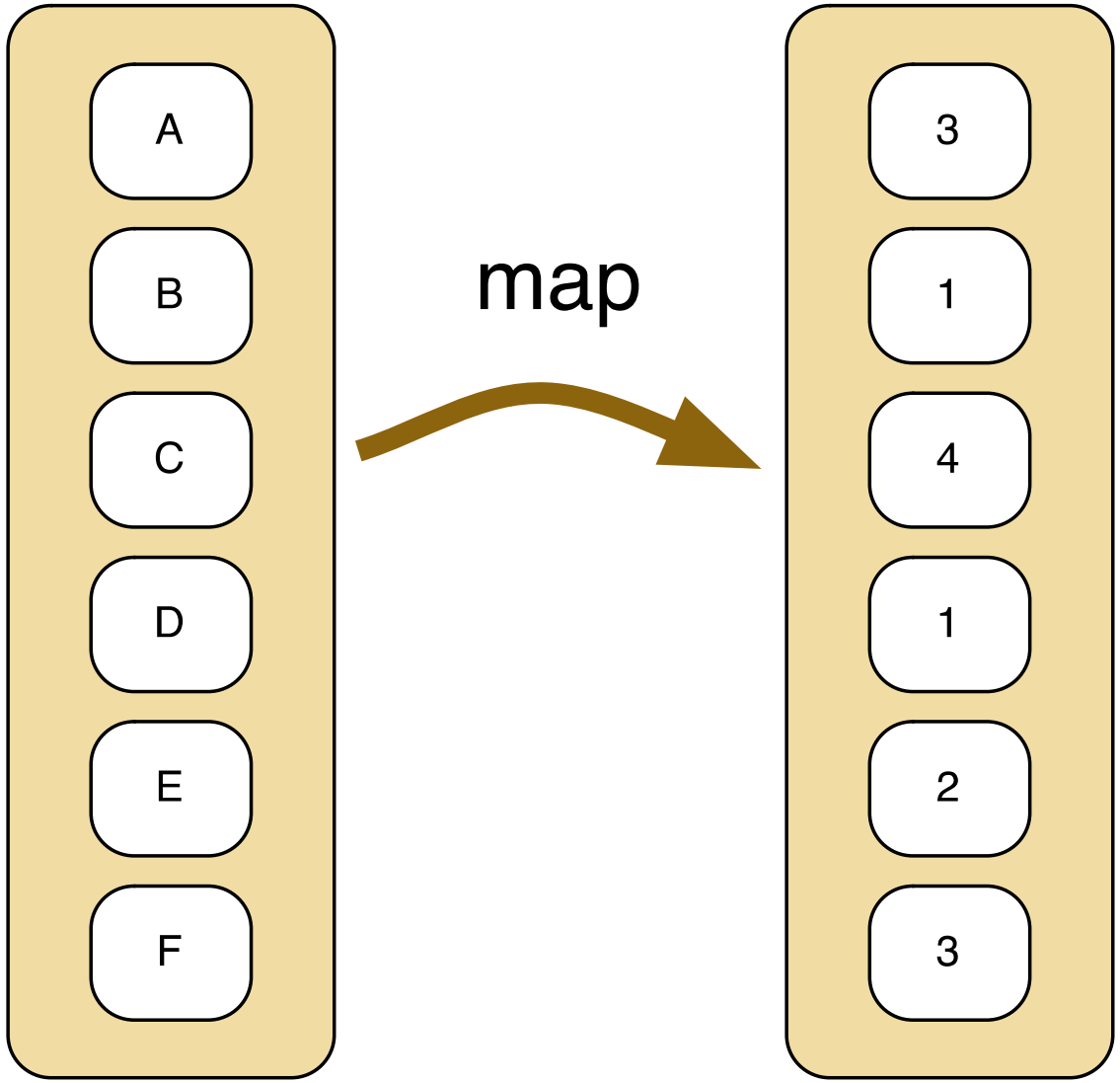
- Streams are immutable, so operations often result in a new stream.
- There are three common types of operation:
  - **Filter**: select items from the input stream to pass on to the output stream.
  - **Map**: replace items from the input stream with different items in the output stream.
  - **Reduce**: collapse the multiple elements of the input stream into a single element.

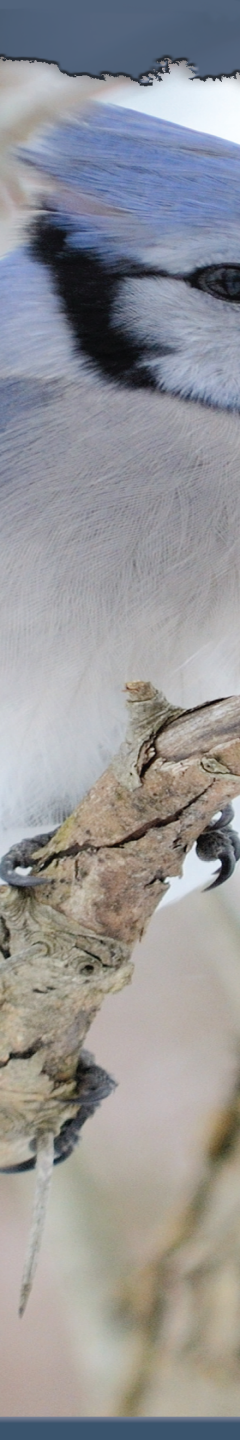
Filter



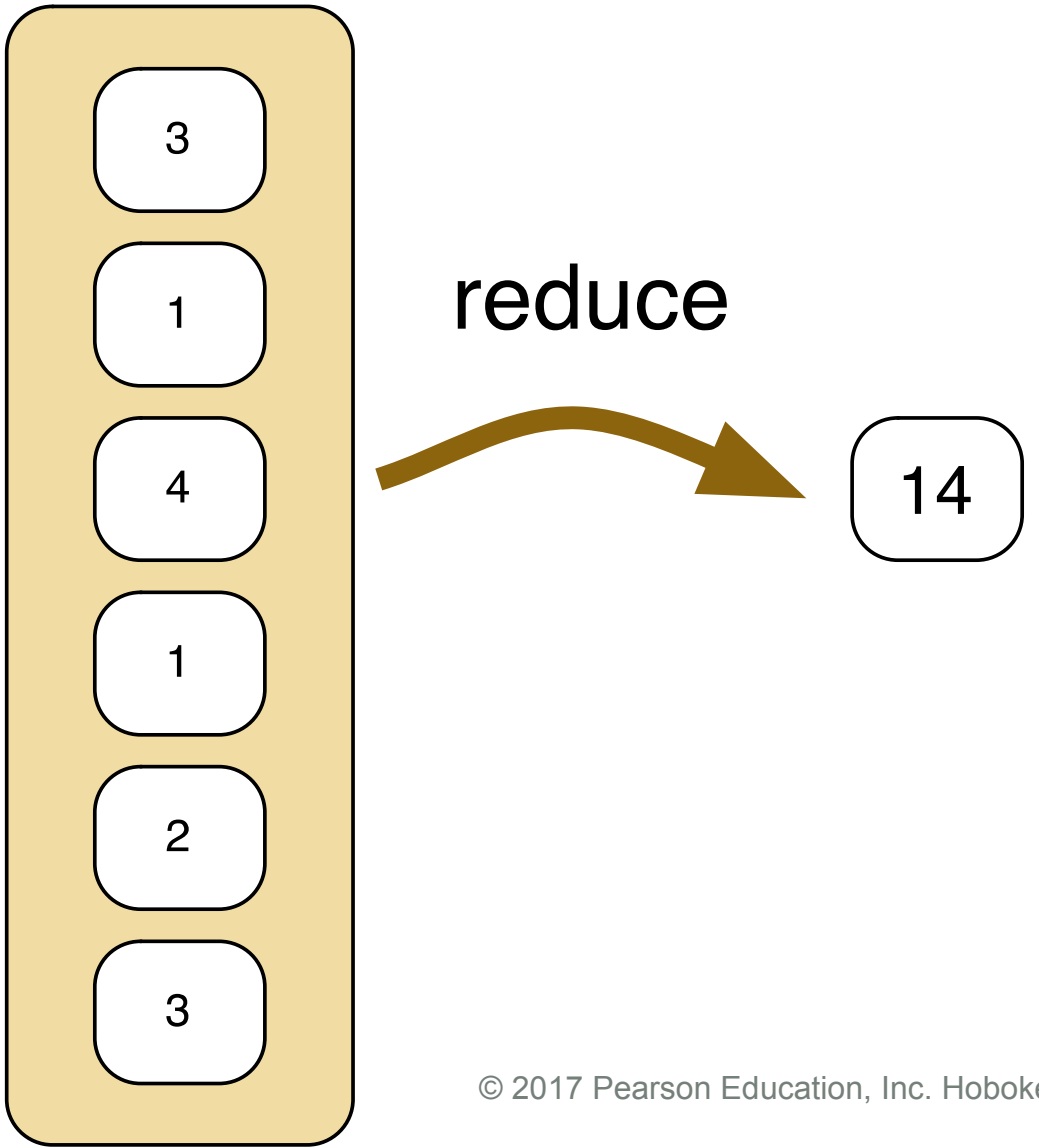


Map

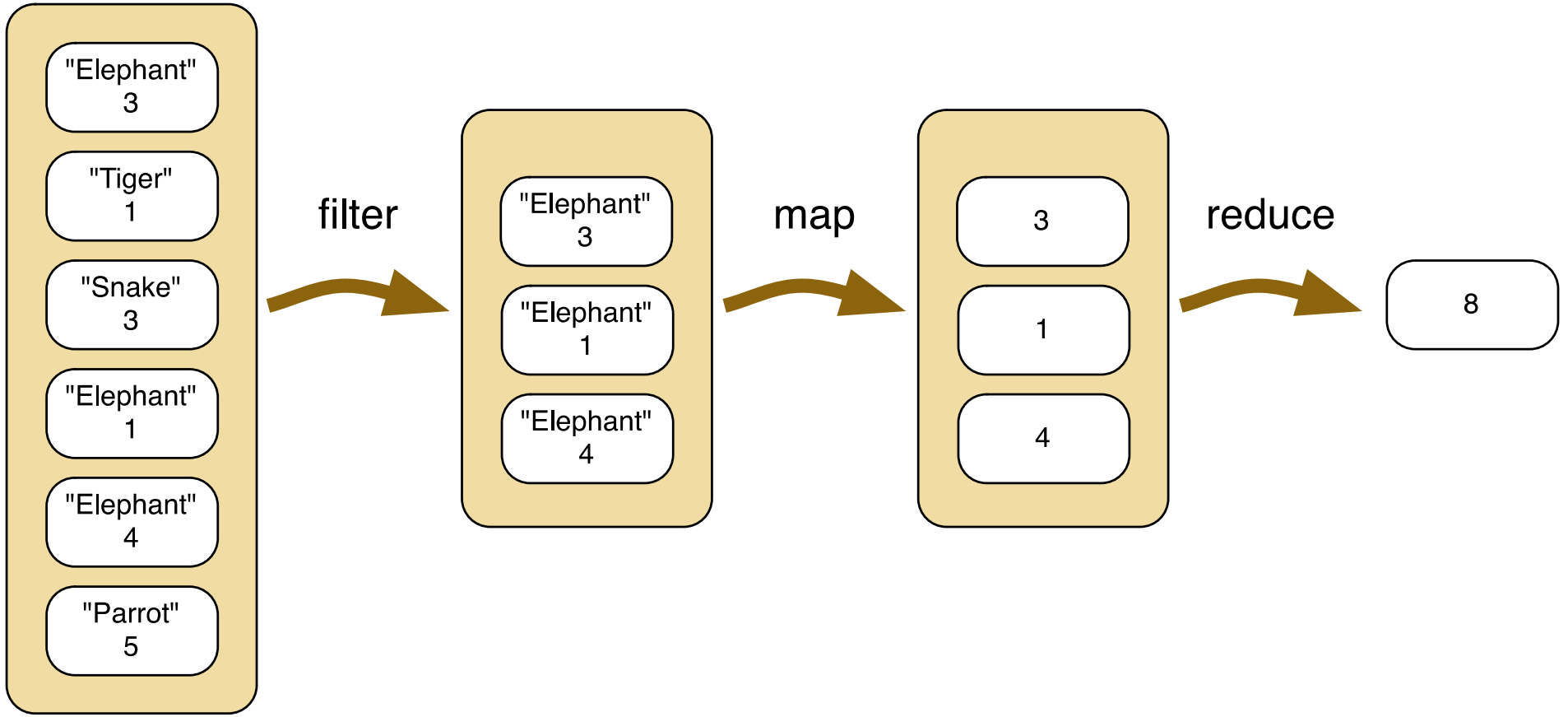




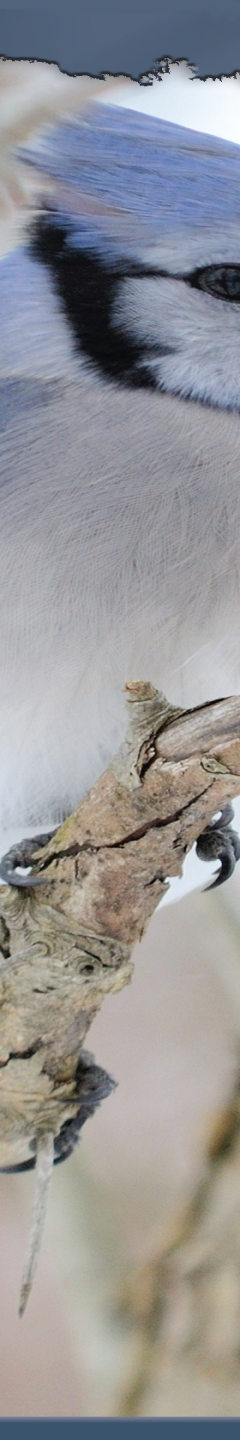
Reduce



A pipeline of operations

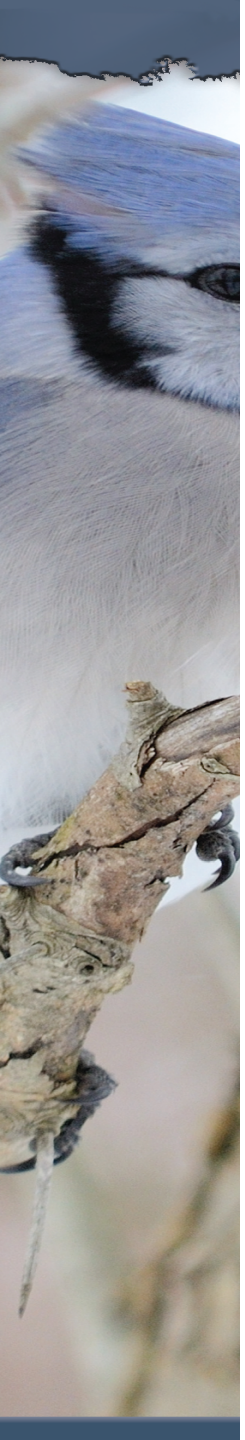



```
filter(name is elephant).map(count).reduce(add up)
```



Pipelines

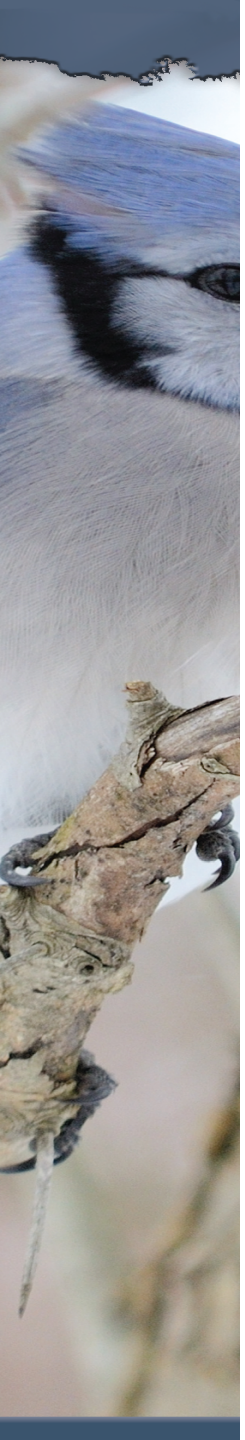
- Pipelines start with a source.
- Operations are either:
  - Intermediate, or
  - Terminal.
- Intermediate operations produce a new stream as output.
- Terminal operations are the final operation in the pipeline.
  - They might have a **`void`** return type.



Filters

- Filters require a **`Boolean`** lambda as a parameter.
- A **`Boolean`** lambda is called a *predicate*.
- If the predicate returns true for an element of the input stream then that element is passed on to the output stream; otherwise it is not. *(Filters determine which elements to retain.)*
- Some predicates:
  - `s -> s.getAnimal().equals("Elephant”)`
  - `s -> s.getCount() > 0`
  - `(s) -> true // Pass on all elements.`
  - `(s) -> false // Pass on none.`
- Example: print details of only the Elephant sightings.

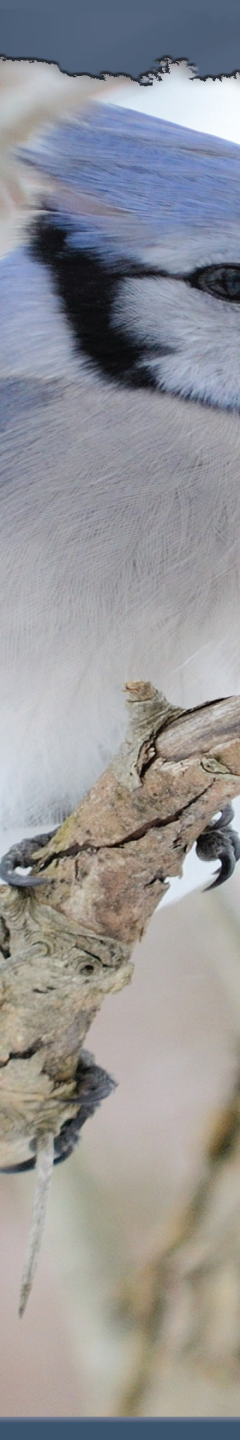
```
sightings.stream()
         .filter(s -> "Elephant".equals(s.getAnimal()))
         .forEach(s -> System.out.println(s.getDetails()));
```



The `map` method

- The type of the objects in the output stream is often (but not necessarily) different from the type in the input stream.
- E.g., extracting just the details **`String`** from a **`Sighting`**:

```
sightings.stream()
         .map(sighting -> sighting.getDetails())
         .forEach(details ->
                  System.out.println(details));
```



The `reduce` method

- More complex than both **`filter`** and **`map`**.
- Its task is to ‘collapse’ a multi-element stream to a single ‘value’.
- It takes two parameters: a value and a lambda:
  `reduce(start, (acc, element) -> acc + element)`
- The first parameter is a starting value for the final result.
- The lambda parameter itself takes two parameters:
  - an accumulating value for the final result, and
  - an element of the stream.
- The lambda determines how to merge an element with the accumulating value.
  - The lambda’s result will be used as the **`acc`** parameter of the lambda for the next element of the stream.
  - The **`start`** value is used as the first **`acc`** parameter that is paired with the first element of the stream.

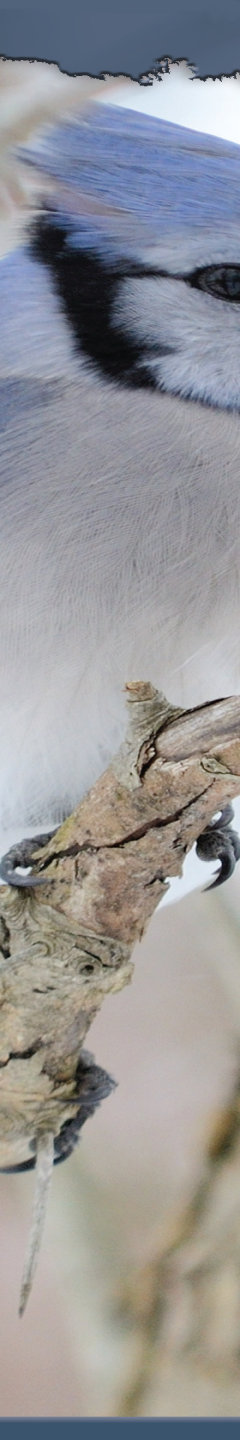
The `reduce` method - a comparative example

```
sightings.stream()
    .filter(sighting -> animal.equals(sighting.getAnimal())
    .map(sighting -> sighting.getCount())
    .reduce(0, (total, count) -> total + count);
```

Initial value

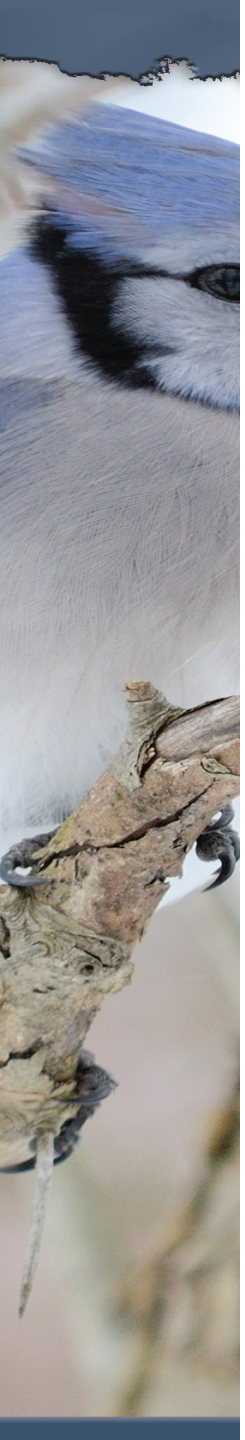
```
int total = 0;
for(Sighting sighting : sightings) {
    if(animal.equals(sighting.getAnimal())) {
        int count = sighting.getCount();
        total = total + count;
    }
}
```

Accumulation



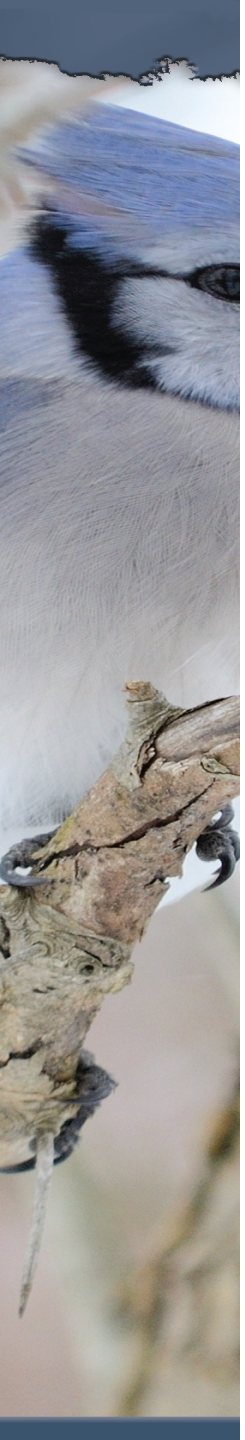
Removal from a collection using a predicate lambda

```
/**
 * Remove from the sightings list all of
 * those records with a count of zero.
 */
public void removeZeroCounts() {
    sightings.removeIf(
          sighting -> sighting.getCount() == 0);
}
```



Summary

- Streams and lambdas are an important and powerful new feature of Java.
- They are likely to increase in importance over the coming years.
- Expect collection processing to move in that direction.
- Lambdas are widely used in other areas, too; e.g. GUI building for event handlers.



Summary

- A collection can be converted to a stream for processing in a pipeline.
- Typical pipeline operations are filter, map and reduce.
- Parallel processing of streams is possible.