CT326 Programming III

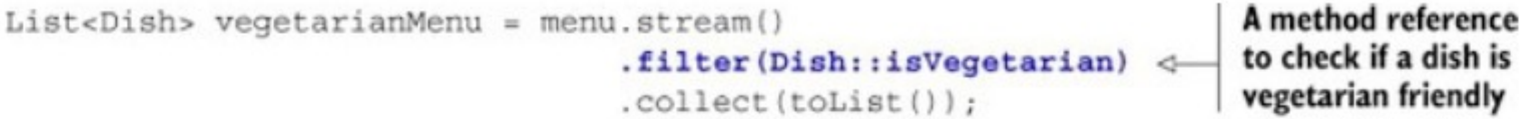
STREAM PROCESSING II

DR ADRIAN CLEAR
SCHOOL OF COMPUTER SCIENCE

Objectives for today

- Become familiar with operations for
  - Filtering, slicing, and matching
  - Finding, matching, and reducing

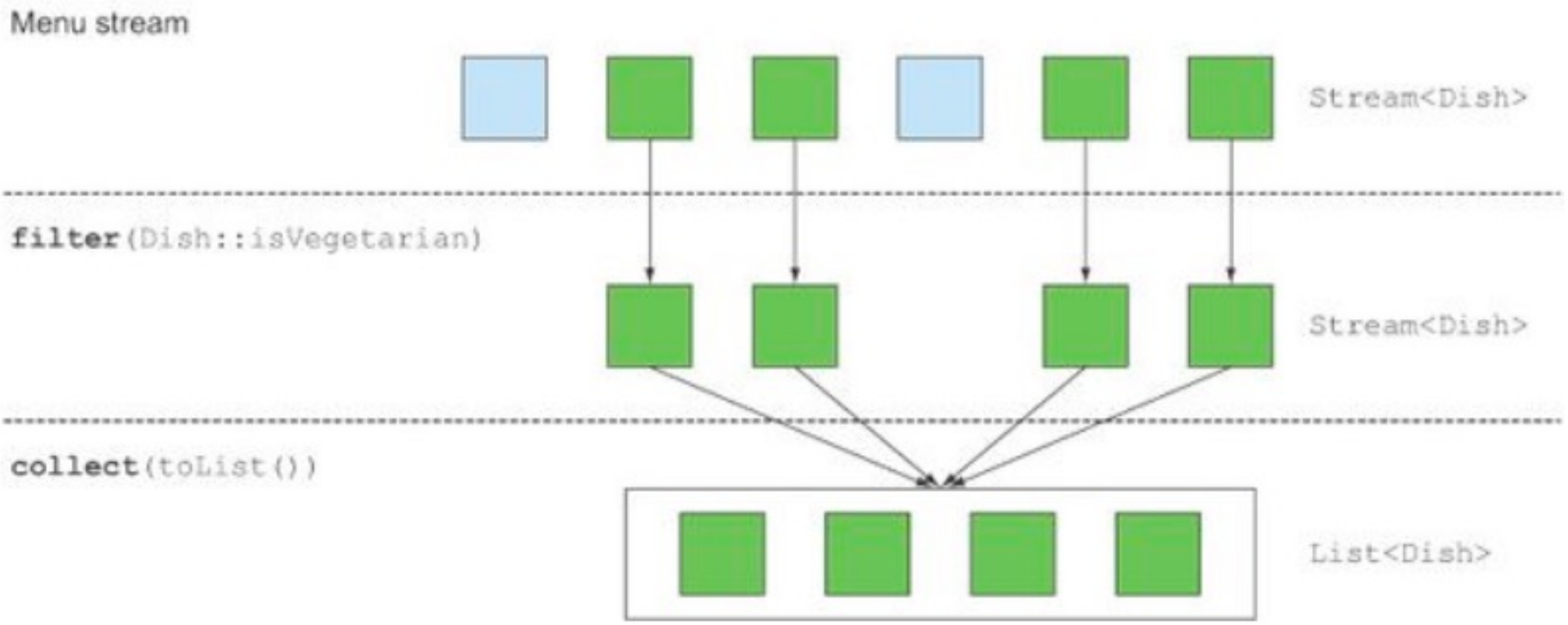
Filtering with predicates



```
List<Dish> vegetarianMenu = menu.stream()
                                .filter(Dish::isVegetarian)
                                .collect(toList());
```

A method reference to check if a dish is vegetarian friendly

Figure 5.1. Filtering a stream with a predicate



Filtering unique elements

- `distinct()`
  - Returns a stream with unique elements
  - Uses implementation of the equals method of objects produced by a stream

- How would you filter all even numbers from a list, making sure there are no duplicates, and print them to the console?

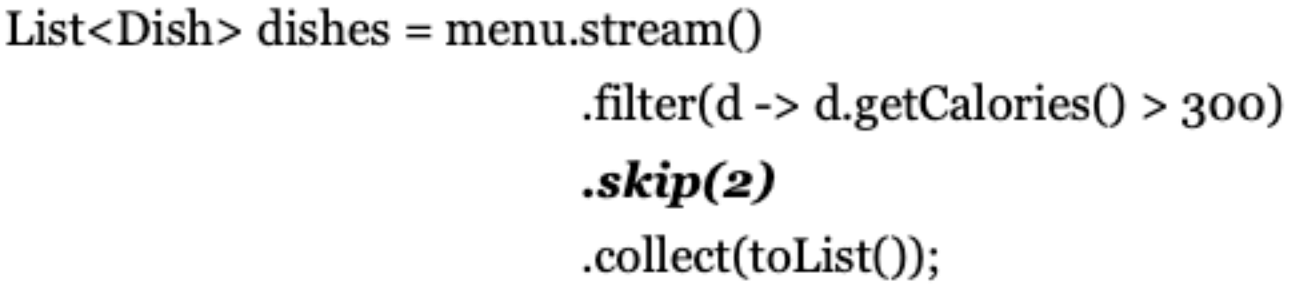
```
List<Integer> numbers = Arrays.asList(1, 2, 1, 3, 3, 2, 4);
...
```

Truncating a stream

- `limit(n)`
  - Returns a stream no longer than n
- Adheres to order for ordered streams
- Also works on unordered streams (e.g., a stream of a `Set`) but order cannot be assumed

Skipping elements

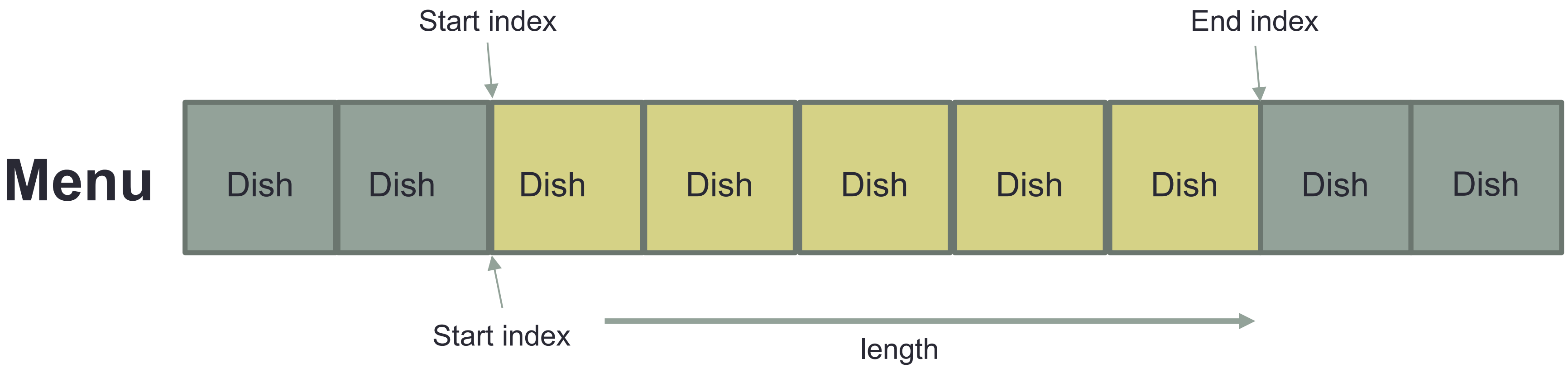
- `skip(n)`
  - Returns a stream that discards the first n elements
  - Or an empty stream



```
List<Dish> dishes = menu.stream()
                        .filter(d -> d.getCalories() > 300)
                        .skip(2)
                        .collect(toList());
```

- How would you use streams to filter the first two meat dishes?

- How would you print the names of the middle five dishes on the menu?


Start index
End index
Menu
Dish
Dish
Dish
Dish
Dish
Dish
Dish
Dish
Dish
Start index
length

Mapping

- Selecting, or *extracting*, information from certain objects
  - Like selecting a column from a table in SQL
- Takes a function as an argument
  - Often a method reference is used
- The type of stream returned by the map method is determined by the return type of the argument function
  - E.g., `map(Dish::getName)` returns a stream of type Stream<String>

- Suppose you have a list of words as follows:

```
List<String> words = Arrays.asList(”Richard", ”Of", ”York", ”Gave”, “Battle”, “In”, “Vain”);
```

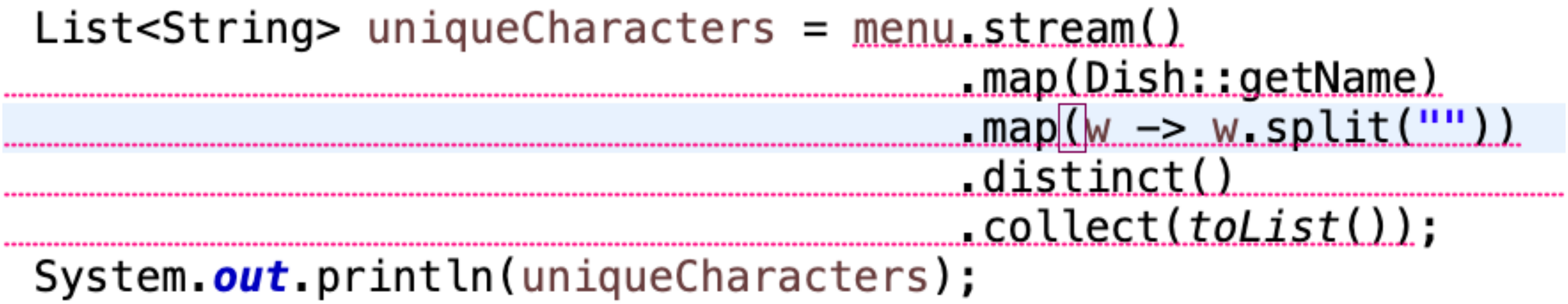
- How might you use stream processing to return a list of the number of characters in each word?

flatMap

- How would you find the unique letters of dishes on the menu?

flatMap

- How would you find the unique letters of dishes on the menu?

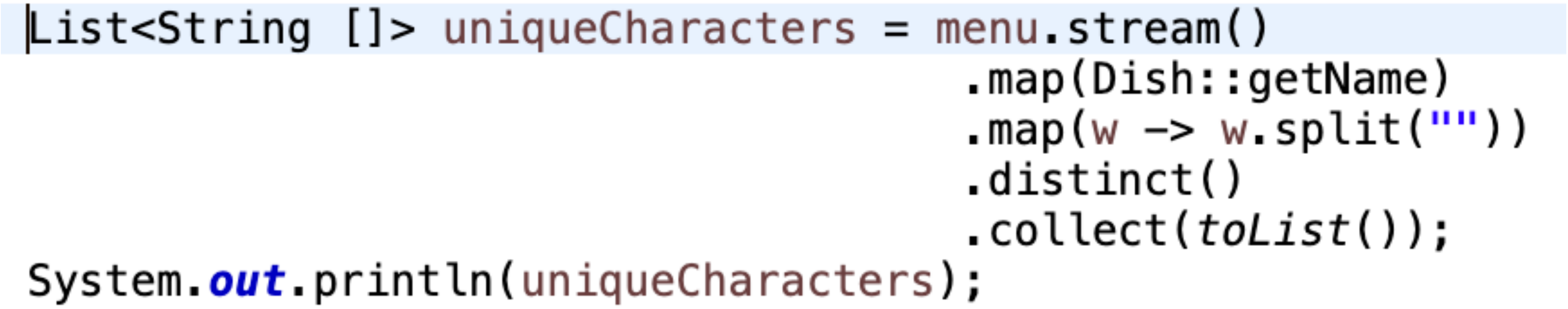


```
List<String> uniqueCharacters = menu.stream()
                                    .map(Dish::getName)
                                    .map(w -> w.split(""))
                                    .distinct()
                                    .collect(toList());
System.out.println(uniqueCharacters);
```

- `map(w -> w.split(""))`
  - Returns type `Stream<String [] >`

flatMap

- How would you find the unique letters of dishes on the menu?

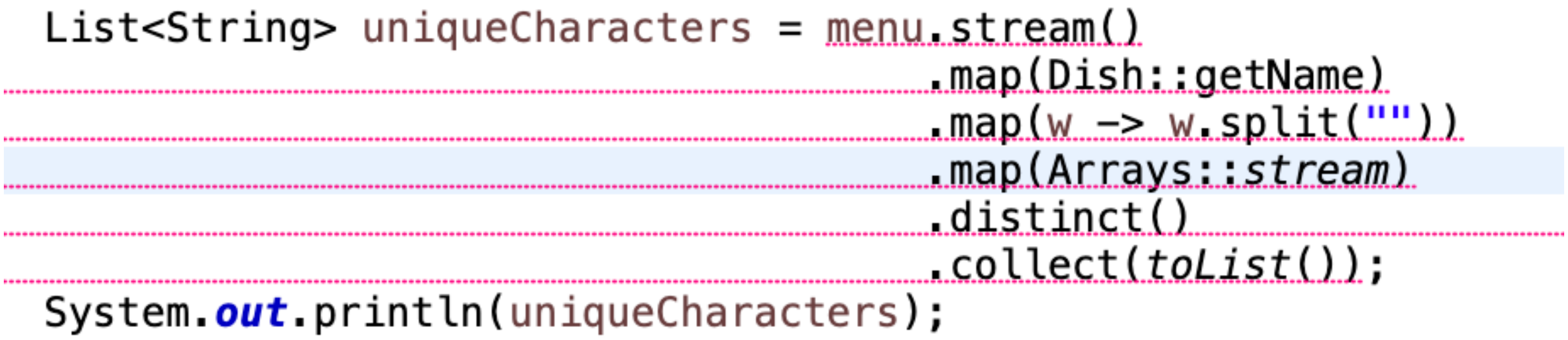


```
List<String []> uniqueCharacters = menu.stream()
                                   .map(Dish::getName)
                                   .map(w -> w.split(""))
                                   .distinct()
                                   .collect(toList());
System.out.println(uniqueCharacters);
```

- Compiles but isn't what we need
  - Ideally, we want `map(w -> w.split(""))` to return something of type `Stream<String>`

flatMap

- `Arrays.stream` takes an array an produces a stream

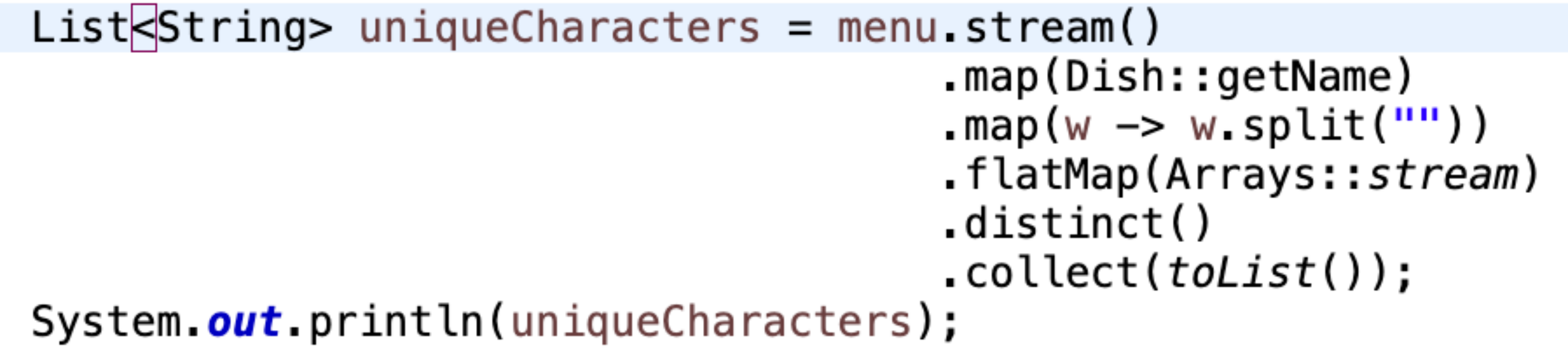


```
List<String> uniqueCharacters = menu.stream()
                                    .map(Dish::getName)
                                    .map(w -> w.split(""))
                                    .map(Arrays::stream)
                                    .distinct()
                                    .collect(toList());
System.out.println(uniqueCharacters);
```

- Now `map(Arrays::`*`stream`*`)` produces a list of streams (`Stream<Stream<String>>`)

flatMap

- `flatMap` allows us to amalgamate all of the separate streams produced from map(Arrays::stream) into a single stream
- Maps each array not with a stream, but with the contents of that stream



```
List<String> uniqueCharacters = menu.stream()
                                    .map(Dish::getName)
                                    .map(w -> w.split(""))
                                    .flatMap(Arrays::stream)
                                    .distinct()
                                    .collect(toList());
System.out.println(uniqueCharacters);
```

Finding and matching

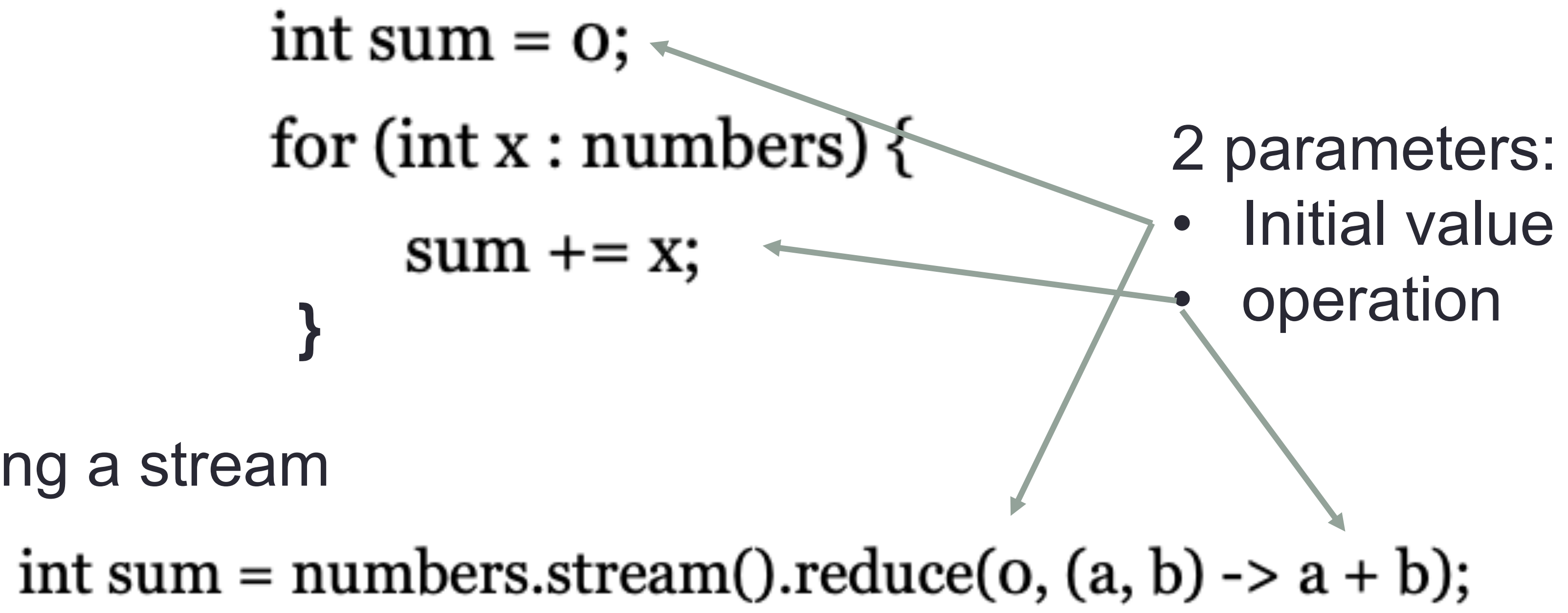
- allMatch
- anyMatch
- noneMatch
- findFirst
- findAny

Reducing

- What if we want to express more complicated queries like
  - “Calculate the sum of all calories in the menu,” or
  - “What is the highest calorie dish in the menu?”
- Combine all elements in the stream repeatedly to produce a single value like an integer
  - i.e., reduce the stream to a single value
  - Known as a *fold* in functional programming

Summing numbers

- For-each loop



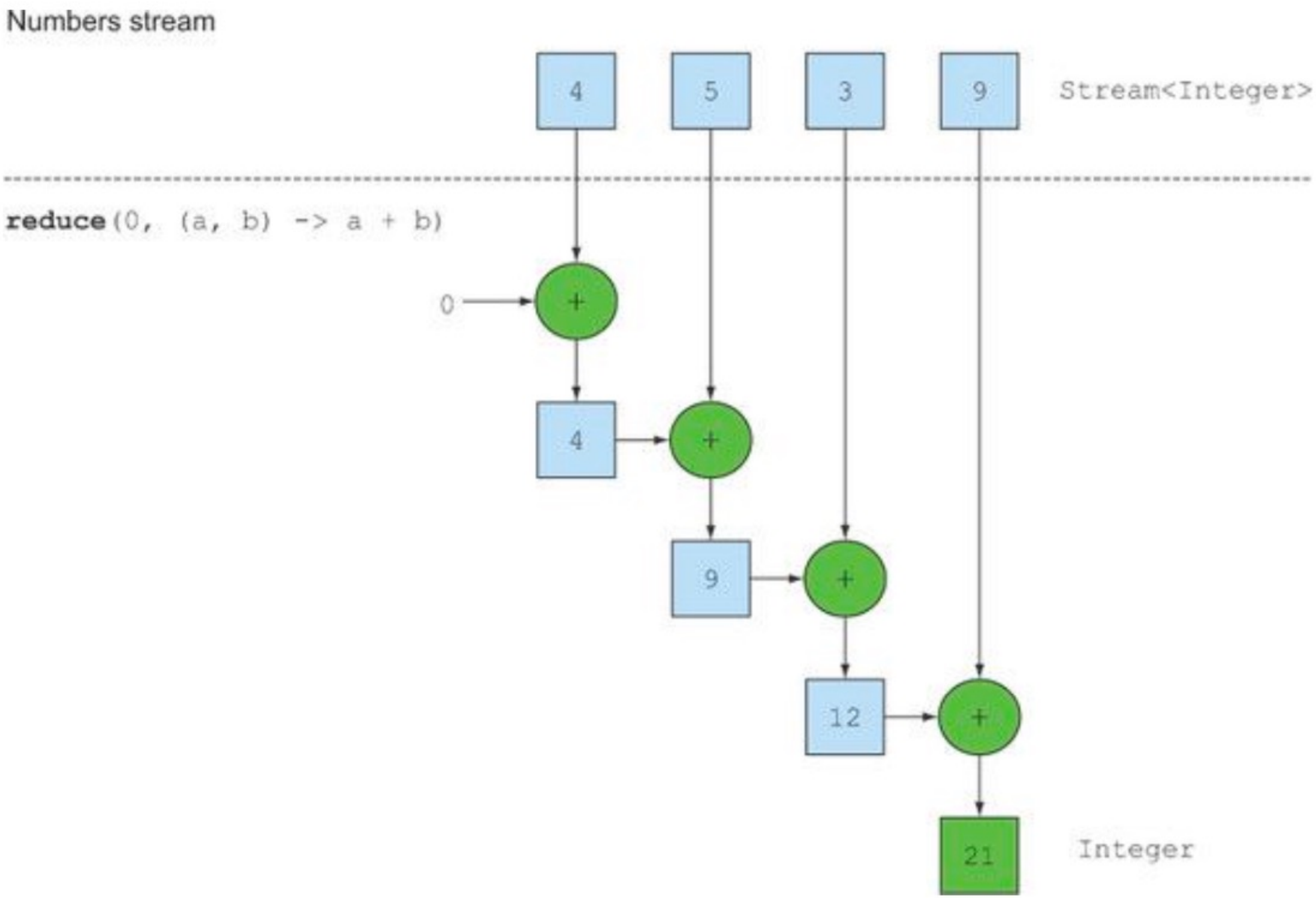
```
int sum = 0;
for (int x : numbers) {
      sum += x;
 }
```

- Using a stream

```
int sum = numbers.stream().reduce(0, (a, b) -> a + b);
```

2 parameters:

- Initial value
- operation


Numbers stream
4
5
3
9
Stream<Integer>
reduce(0, (a, b) -> a + b)
0
+
4
+
9
+
12
+
21
Integer

Stream operations: stateless vs. stateful

- Stateless operations
  - Some operations like map and filter don’t have an internal state
  - They take each element from an input stream and produce zero or one results in the output stream
- Stateful operations
  - Operations like reduce and limit need to have internal state in order to produce their result (e.g. accumulating)
  - This internal state can be *bounded* in size i.e., isn’t affected by the number of elements in the stream
  - Other operations like sorted and distinct are *unbounded* as they require knowing the previous history in order to produce their result
  - Sorted requires all elements to be buffered before a single element can be added to the output stream
    - Can be problematic if the stream is large