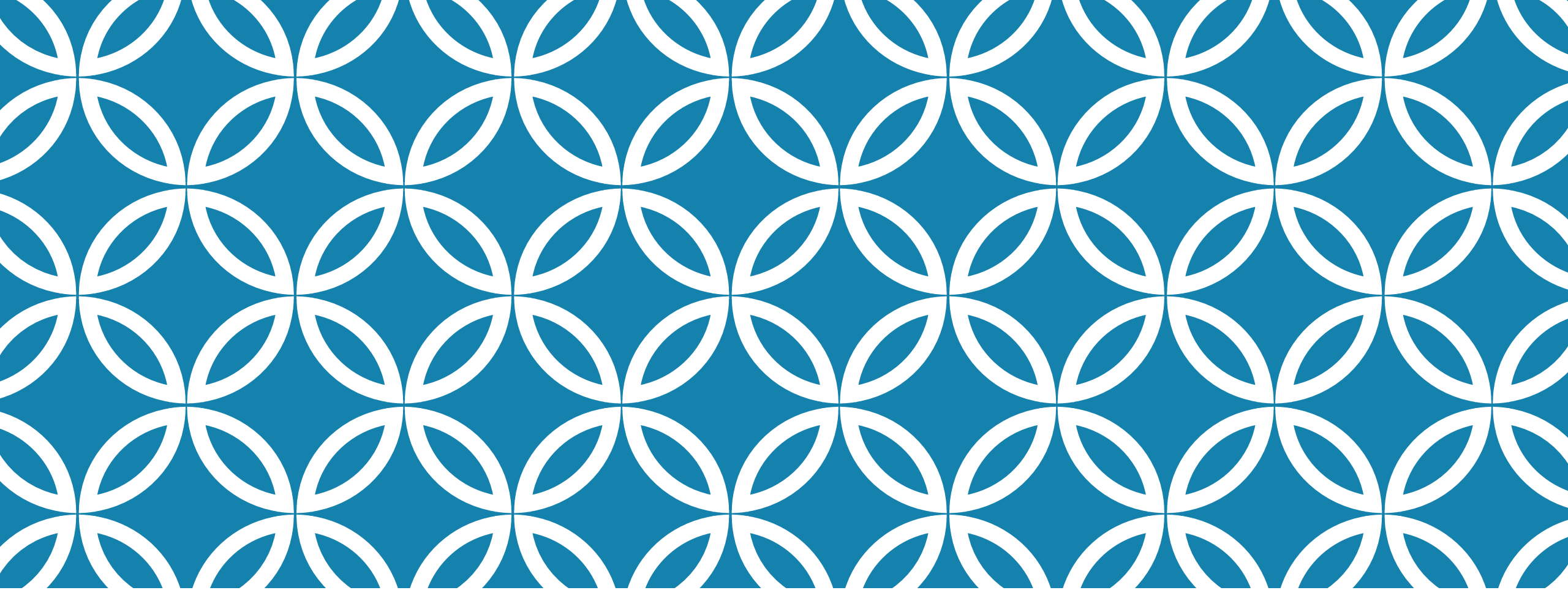
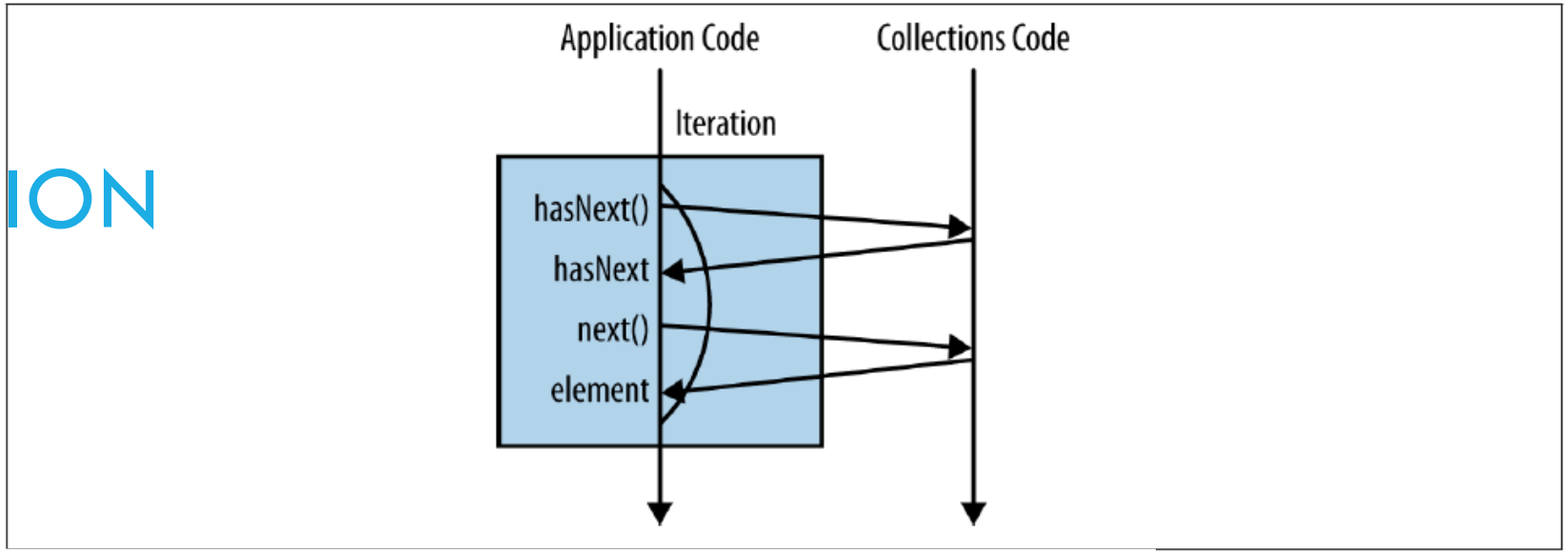


STREAMS AND FUNCTIONAL PROGRAMMING

Chandreyee Chowdhury

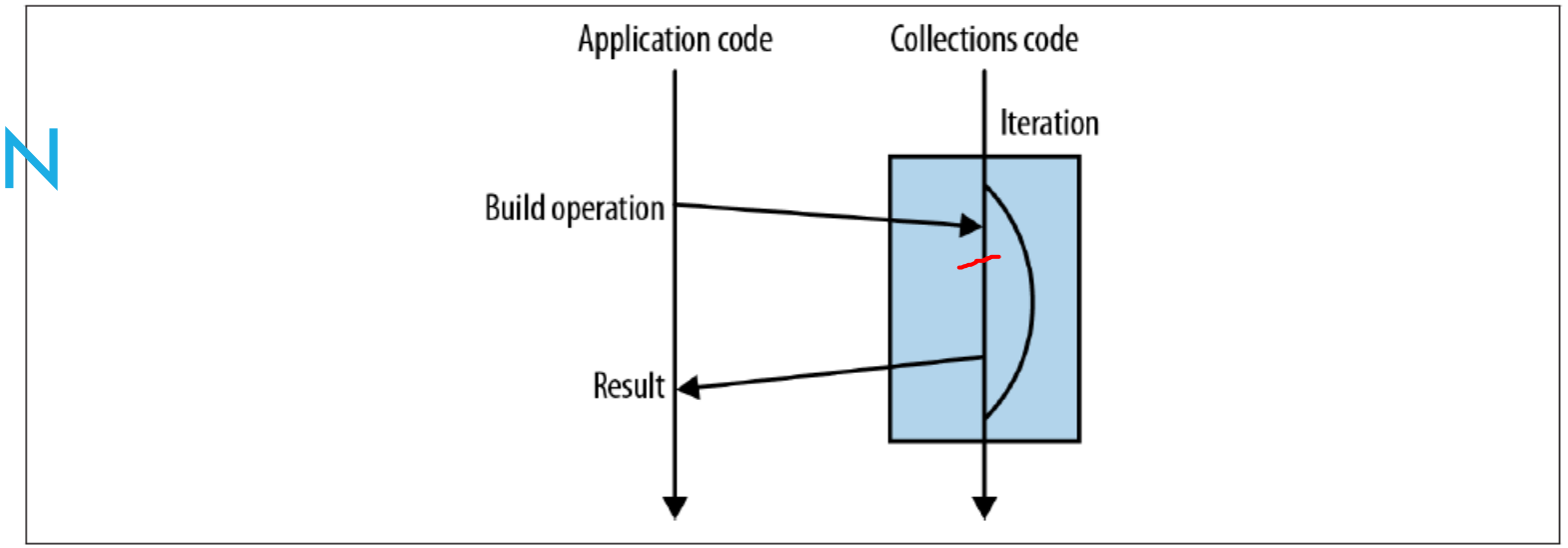
EXTERNAL ITERATION




```
int count = 0;
Iterator<Artist> iterator = allArtists.iterator();
while(iterator.hasNext()) {
    Artist artist = iterator.next();
    if (artist.isFrom("London")) {
        count++;
    }
}
```

- **Inherently serial in nature**
- **Hard to parallelize**

INTERNAL ITERATION




```
long count = allArtists.stream()
                       .filter(artist -> artist.isFrom("London"))
                       .count();
```

- Instead of returning an Iterator to control the iteration, it returns the equivalent interface in the internal iteration world: Stream.
- A Stream is a tool for building up complex operations on collections using a functional approach
- **The functions performed are**
  - **Finding all the artists from London**
  - **Counting a list of artists**

JAVA STREAMS

- Streams allow to write collection processing code from a higher level of abstraction
- It allows programmers to write codes that are
  - **Declative- more concise and readable**
  - **Composable- greater flexibility**
  - **Parallelizable- greater performance**
    - **Maximize the performance for multicore architecture transparently**
    - **Don’t need to specify how many threads to use**

STREAMS

- Streams can be defined as a sequence of elements from a source that supports data processing operations
- Collections are data structures focusing on storing and accessing of elements
- Streams are about expressing computations
- Unlike collection, stream provides an interface to a sequence of specific type of elements

STREAMS

Streams can be defined as a sequence of elements from a source that supports data processing operations

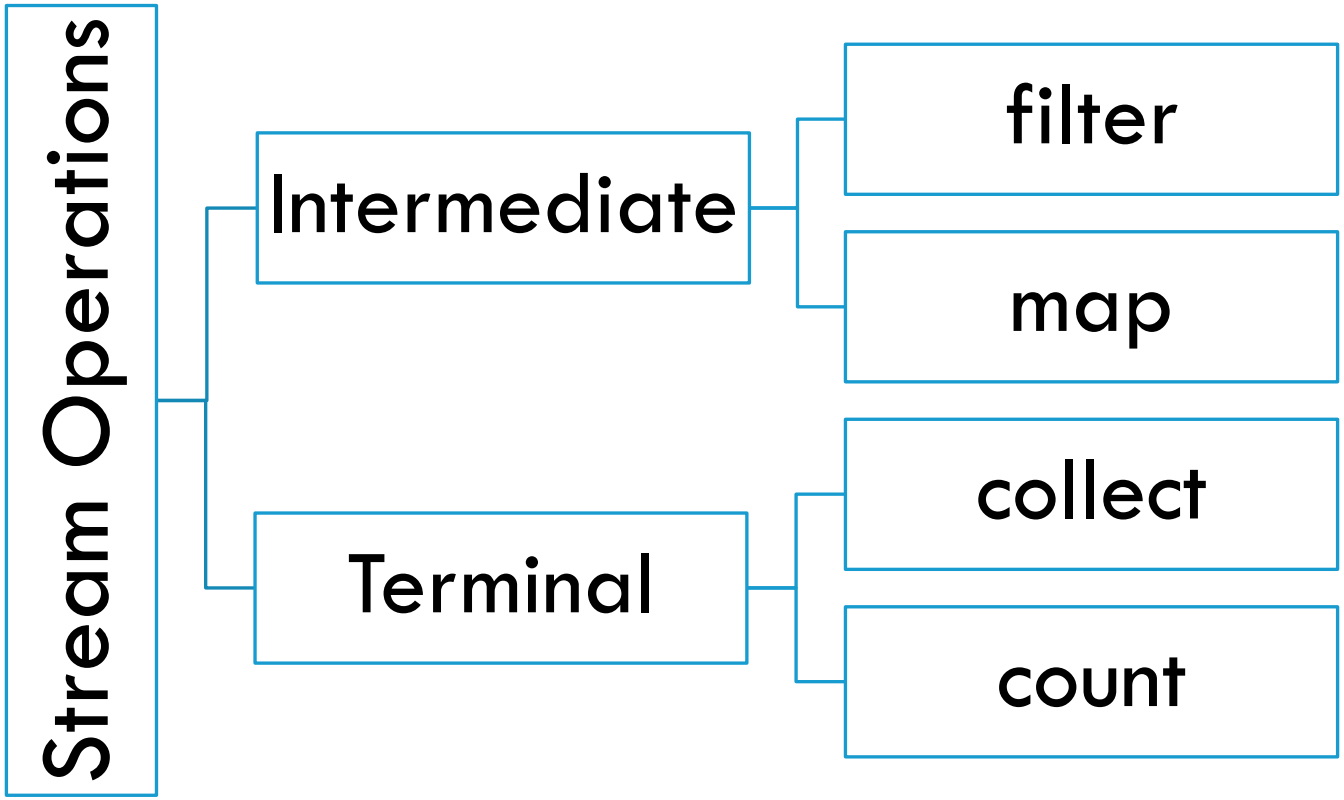
- **Source**
  - **Streams consume data from a data providing source such as, collections, arrays, or I/O resources**
  - **Streams from an ordered collection preserves the ordering**
- **Data processing operations**
  - **supports both database like operations and functional programming operations to manipulate data**
  - **operations can be executed in sequence or in parallel**

```
menu.stream().filter(d->d.getCalories()>350)
             .map(d1->d1.getName())
          .collect(toList());
```

Dish

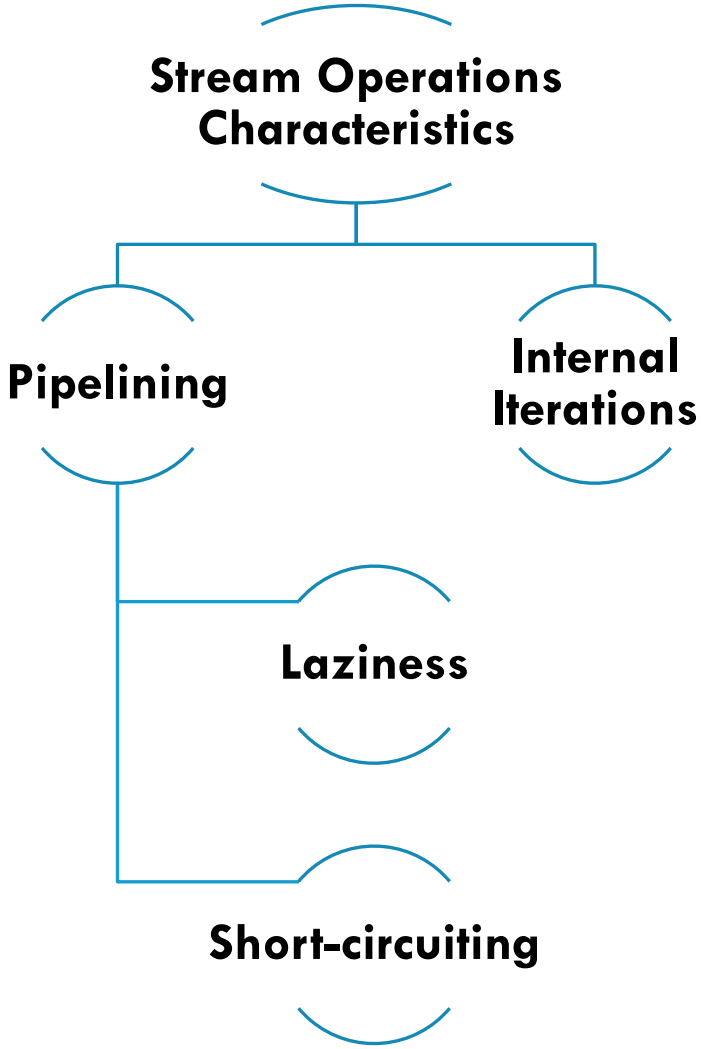
```
private final String name;
private final boolean vegetarian;
private final int calories;
private final Type type;

public Dish(String name, boolean vegetarian, int calories, Type type);
public String getName();
public boolean isVegetarian();
public int getCalories();
public Type getType();
public String toString();
public enum Type { MEAT, FISH, OTHER }
```

STREAM OPERATIONS



```
menu.stream().filter(d->d.getCalories()>350)
                .map(d1->d1.getName())
                .collect(toList());
```

- Loop fusion- filter and map are two separate operations that are merged into one pass
- short circuiting- despite the fact that there are many high calorie dishes, the only 3 are selected



STREAM VS COLLECTION

Stream

- fixed data structure whose elements are computed on demand
- lazily constructed collection
- Consumer driven
- Traversable exactly once
- Stream is a set of values spread out in time
- Internal iteration

Collection

- every element is computed before it is added to a collection
- eagerly constructed collection
- Supplier driven
- No such restriction
- A set of values spread out in space
- External iteration

EXTERNAL VS INTERNAL ITERATION

Internal Iteration

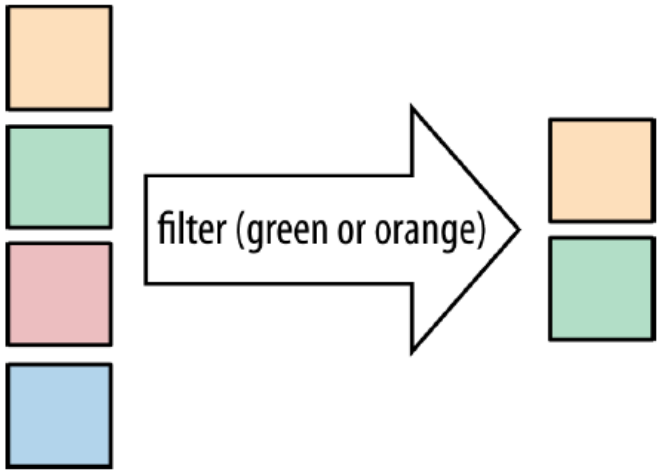
- processing of elements can be done in parallel or in a different order that is more optimized
- stream library can automatically chose a data representation and implementation of parallelism to match the machine hardware

External Iteration

- programmer needs to implement parallelism and define the order in which the elements of a collection can be processed
- committed to a single threaded step-by-step sequential iteration

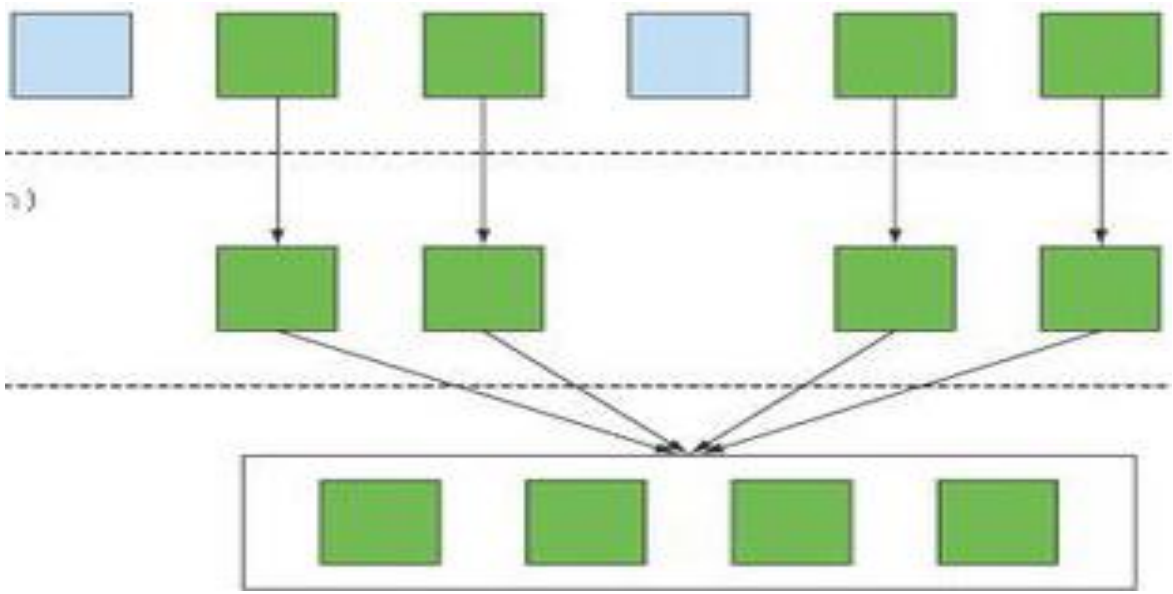
FILTERING

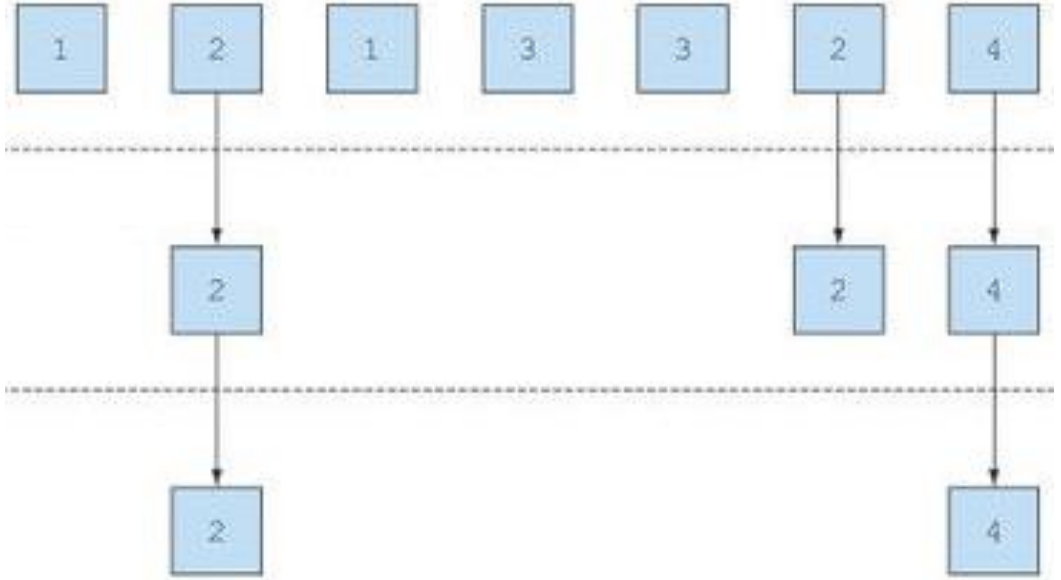
- Where clause of a select statement
- Takes a Predicate object as an argument
- Returns a stream including all elements that match with the predicate
- If you're refactoring legacy code, the presence of an if statement in the middle of a for loop is a pretty strong indicator that you really want to use filter



FILTERING

```
List<String> WithNos=Stream.of("a","1ab","1A", "2A")
                                    .filter(d2->Character.isDigit(d2.charAt(0)))
    .collect(toList());
```





SMALL PROBLEMS

TRUNCATING A STREAM

- Limit
- Streams support the limit(n) method, which returns another stream that's no longer than a given size
- The requested size is passed as argument to limit.
- If the stream is ordered, the first elements are returned up to a maximum of n

- Skip
- Streams support the skip(n) method to return a stream that discards the first n elements.
- If the stream has fewer elements than n, then an empty stream is returned.

MAPPING

```
List<String> collected = Stream.of("a", "b", "hello")
.map(string -> string.toUpperCase())
.collect(toList());
```

The function is applied to each element, mapping it into a new element

the word *mapping* is used because it has a meaning similar to *transforming* but with the nuance of "creating a new version of" rather than "modifying"

Converting strings to uppercase equivalents

List<String> words = Arrays.asList("Java8", "Lambdas", "In", "Action");
List<Integer> wordLengths =

MAPPING

how could you return a list of all the *unique characters* for a list of words?

```
List<String> word1=Arrays.asList("Hi", "Hello", "Hi",
"Hi", "Hello", "Hell", "Heaven");

distinctLetters=word1.stream().map(w->w.split(""))
                              .distinct()
                              .collect(toList());
```

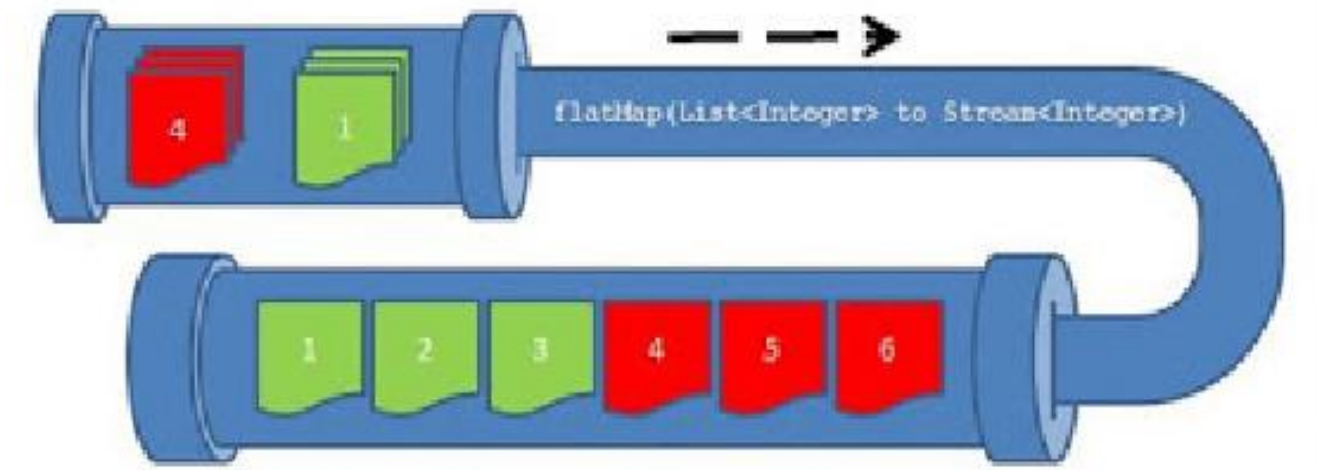
MORE WITH MAPS

FLATMAP

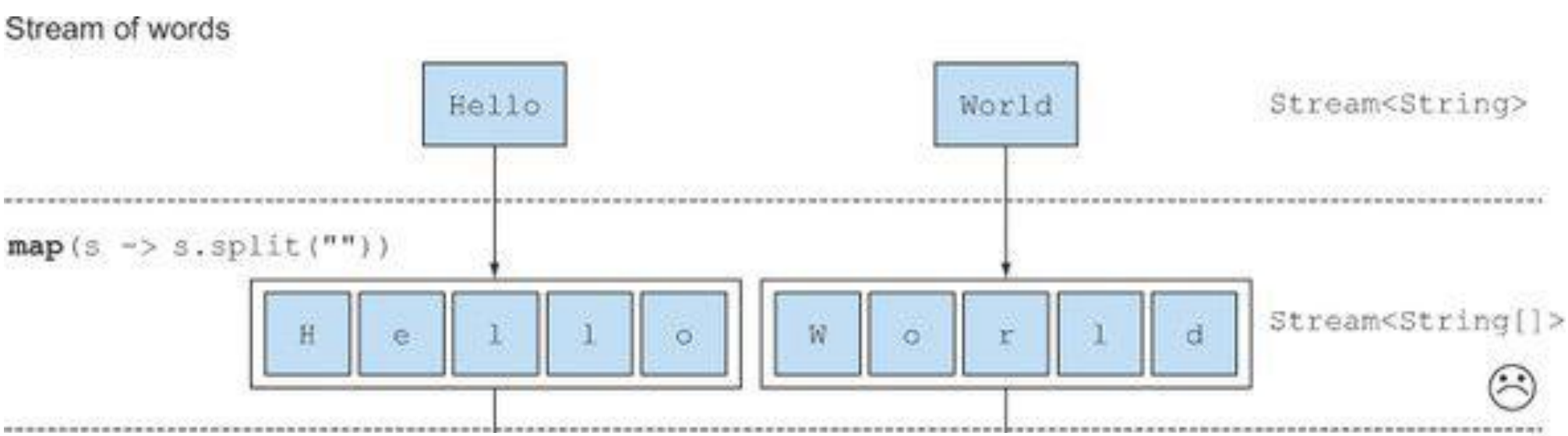
Replaces a value with a stream and concatenates all streams together

List<Integer> together=Stream.of(asList(1,2),asList(4,5)).

flatMap(n->n.stream()).
collect(toList());



FLATMAP


Stream of words
Hello
World
Stream<String>
map(s -> s.split(""))
H
e
l
l
o
W
o
r
l
d
Stream<String[]>

FLATMAP

Form a pair of numbers taking each number from two lists of numbers

```
numbers1.stream()
                .flatMap(i->numbers2.stream()
        .map(k->new int[]{i,k}))
       .collect(toList());
```

FINDING AND MATCHING

SHORT CIRCUITING

- The matching functions do not need to process the entire stream to give the result
- They can turn an infinite stream to constant size
- Limit
- findFirst –more constraining for parallel streams
- findAny
- menu.stream().filter (d->!d.isVegetarian() && d.getCalories()<400)
- .findAny()

- .ifPresent(d->System.out.println(d.getName());

```
if (value != null) value.someMethod();
```

OPTIONAL

- The Optional<T> class (java.util.Optional) is a container class to represent the existence or absence of a value.
- The absence of a value is modeled with an “empty” optional returned by the method Optional.empty
- What the difference is between a null reference and Optional .empty()?
- Semantically, they could be seen as the same thing, but in practice the difference is huge:
- trying to dereference a null will invariably cause a NullPointer-Exception, whereas Optional.empty() is a valid, workable object of type Optional that can be invoked in useful ways.
- It’s important to note that the intention of the Optional class is not to replace every single null reference.
- Instead, its purpose is to help you design more-comprehensible APIs so that by just reading the signature of a method, you can tell whether to expect an optional value
- This forces you to actively unwrap an optional to deal with the absence of a value

OPTIONAL

Optional to avoid bugs related to null checking

isPresent() returns true if Optional contains a value, false otherwise.

T get() returns the value if present; otherwise it throws a NoSuchElement-Exception.

ifPresent(Consumer<T> block) executes the given block if a value is present

T orElse(T other) returns the value if present; otherwise it returns a default value

TERMINAL OPERATIONS

Terminal operation

So far, the terminal operations are found to return

- a boolean
  - (allMatch and so on)
- void
  - (forEach), or
- an Optional object
  - (findAny and so on)

Reducing

- Combines all elements of the stream repeatedly to produce a single value as result
- This is called fold

REDUCING

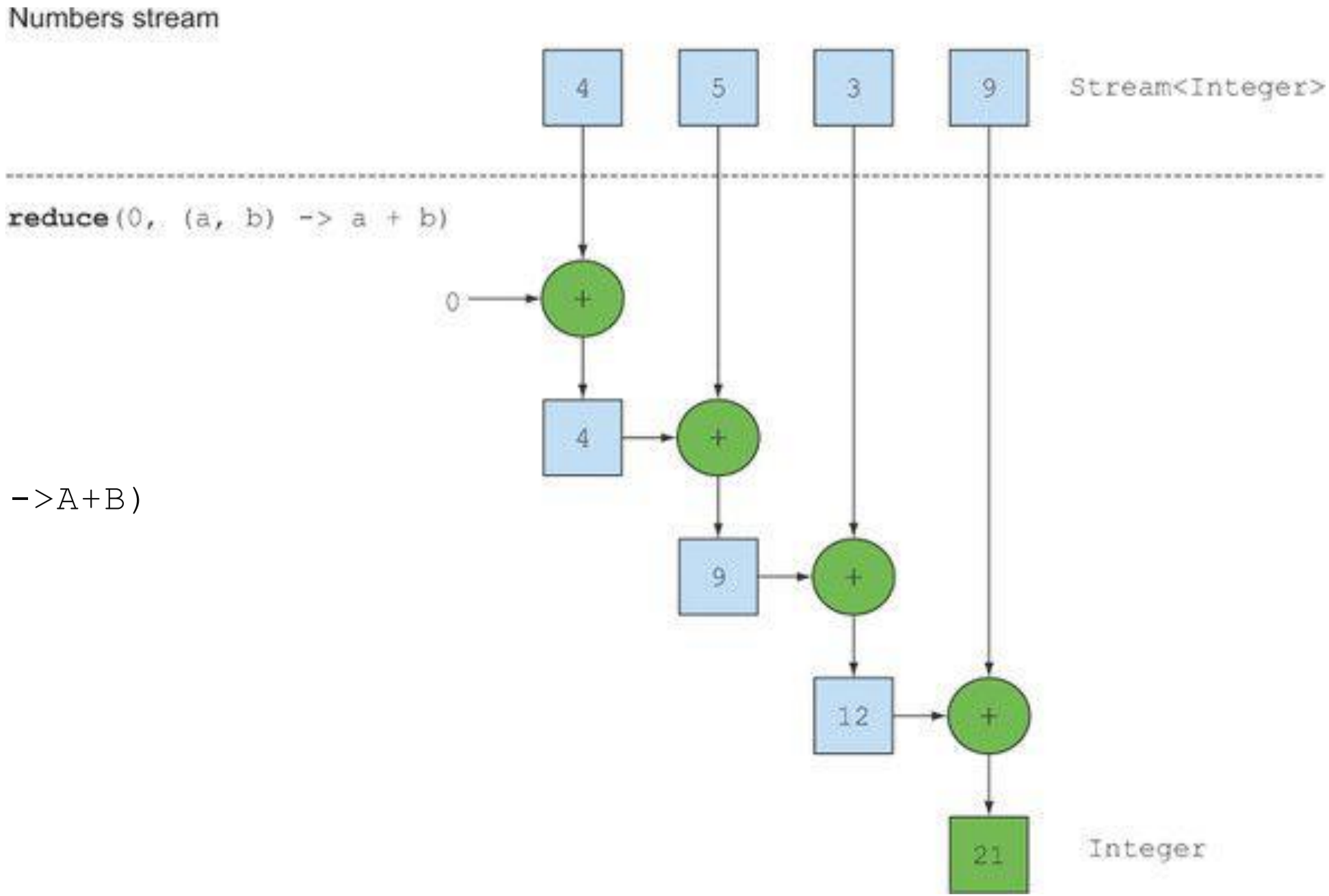
Summing up elements

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

The reduce operation abstracts over this pattern of repeated application

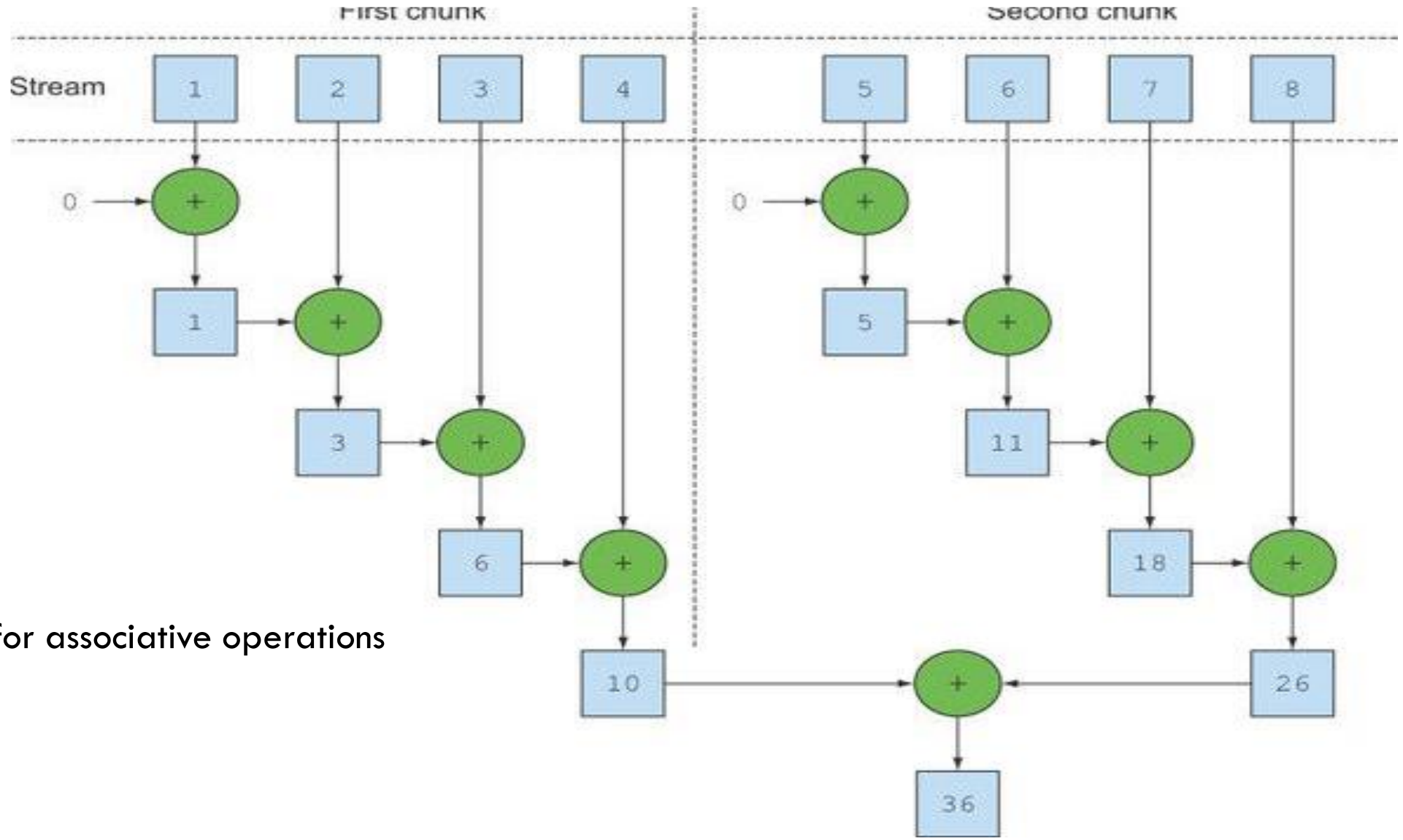
reduce takes two arguments:

- An initial value, here 0.
- A BinaryOperator<T> to combine two elements and produce a new value; here you use the lambda `(accumulator, element) -> accumulator + element`



REDUCE(0,(A,B)->A+B)

MUTABLE ACCUMULATOR VS FORK JOIN



Only for associative operations

REDUCING

```
Optional<Integer>
product=numbers.stream().reduce((a,term)→a*term);
```

Maximum and minimum of numbers

```
int max=numbers.stream().reduce(0,(a,b)->(a>b)?a:b);
```

Counting

```
int totalcount= numbers.stream().reduce(0, (a,b)→a+1);
```

SMALL PROBLEMS

VERSION 1

```
List<Artist> musicians = album.getMusicians()
                                .collect(toList());

Track shortestTrack = tracks.stream() .min(Comparator.comparing(track
-> track.getLength())).get();
```

```
Set<String> origins = bands.stream()
                           .map(artist -> artist.getNationality())
                           .collect(toSet());
```

STREAM OPARATIONS

Operations like `map()` and `filter()` are stateless

- take an input stream
- Process each element of the stream
- Produce 0 or 1 result in the output stream

`Sum,max, reduce,limit,skip`

- Need an internal state to accumulate the results
- The state is small and bounded
- It does not depend on the stream being processed

STREAM OPERATIONS

`Sort, distinct`

- take an input stream
- Process each element of the stream
- Produce 1 result in the output stream
- To compute they need the previous history
- Stateful operations
- Unbounded storage space

The stream operations that do not pose an order are easier for parallelization

Stream poses an encounter order in which each element is operated upon

This depends on both

- the source of the data
- The operation performed on the stream

NUMERIC STREAMS

Three numeric interfaces to reduce the cost of boxing

IntStream

DoubleStream

LongStream

```
IntStream exampleStream=menu.stream().mapToInt(d->d.getCalories());
Stream<Integer> example=exampleStream.boxed();
OptionalInt maxCalories=menu.stream().mapToInt(d->d.getCalories()).max();
maxCalories.OrElse(0);
```

INTSTREAM OPTIMIZATIONS

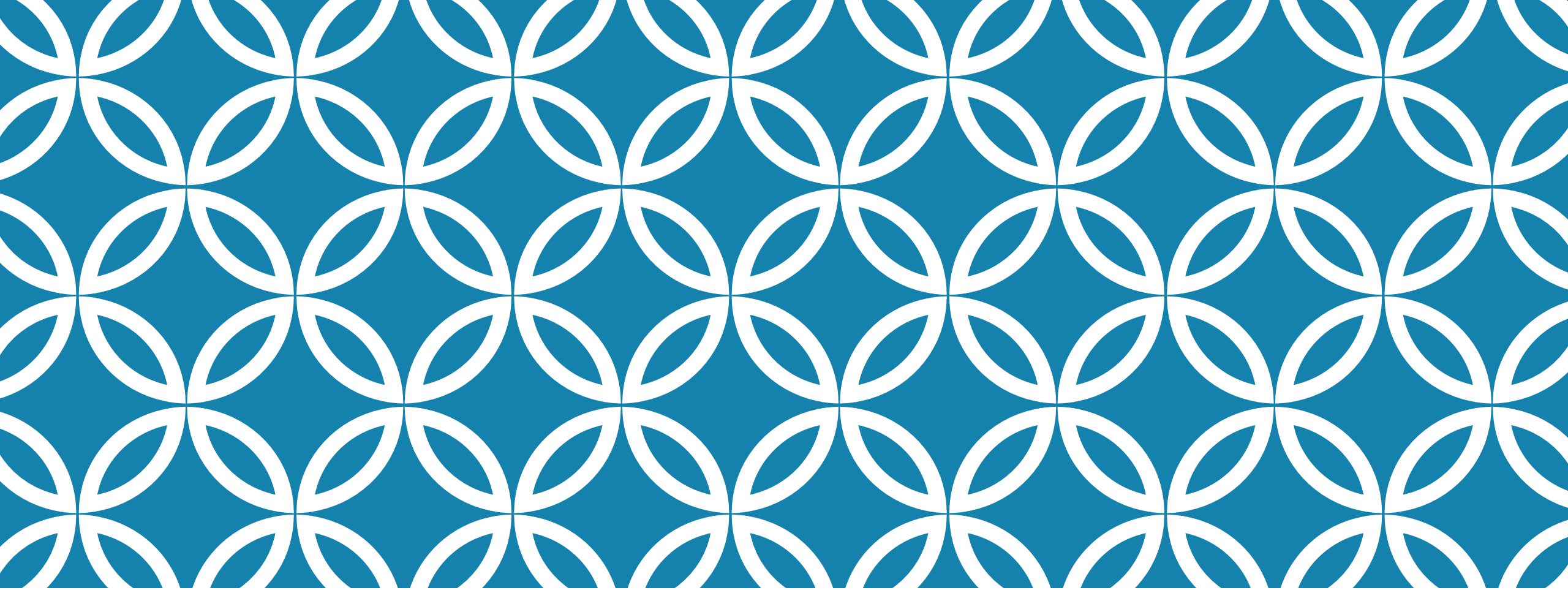
```
int calories = menu.stream().map(Dish::getCalories).sum();

OptionalInt maxCalories = menu.stream()
.mapToInt(Dish::getCalories) .max();
```

INTSTREAM OPTIMIZATIONS

- There are methods sum, average, max, and min that return the sum, average, maximum, and minimum. These methods are not defined for object streams.
- The summaryStatistics method yields an object of type IntSummaryStatistics, LongSummaryStatistics, or DoubleSummaryStatistics that can simultaneously report the sum, average, maximum, and minimum of the stream.

```
IntSummaryStatistics oneToHundred
=IntStream.rangeClosed(1,100).summaryStatistics();
```

```
System.out.println("average=" + oneToHundred.getAverage());
```



STREAM CREATION AND COLLECTING STREAMS

NUMERIC STREAMS

Creating numeric streams

- `IntStream oneToHundred =IntStream.rangeClosed(1,100).filter(i%2==0)`
- `IntStream oneToNinetyNine = IntStream.range(1,100).filter(i%2==0)`

BUILDING STREAMS

Static methods

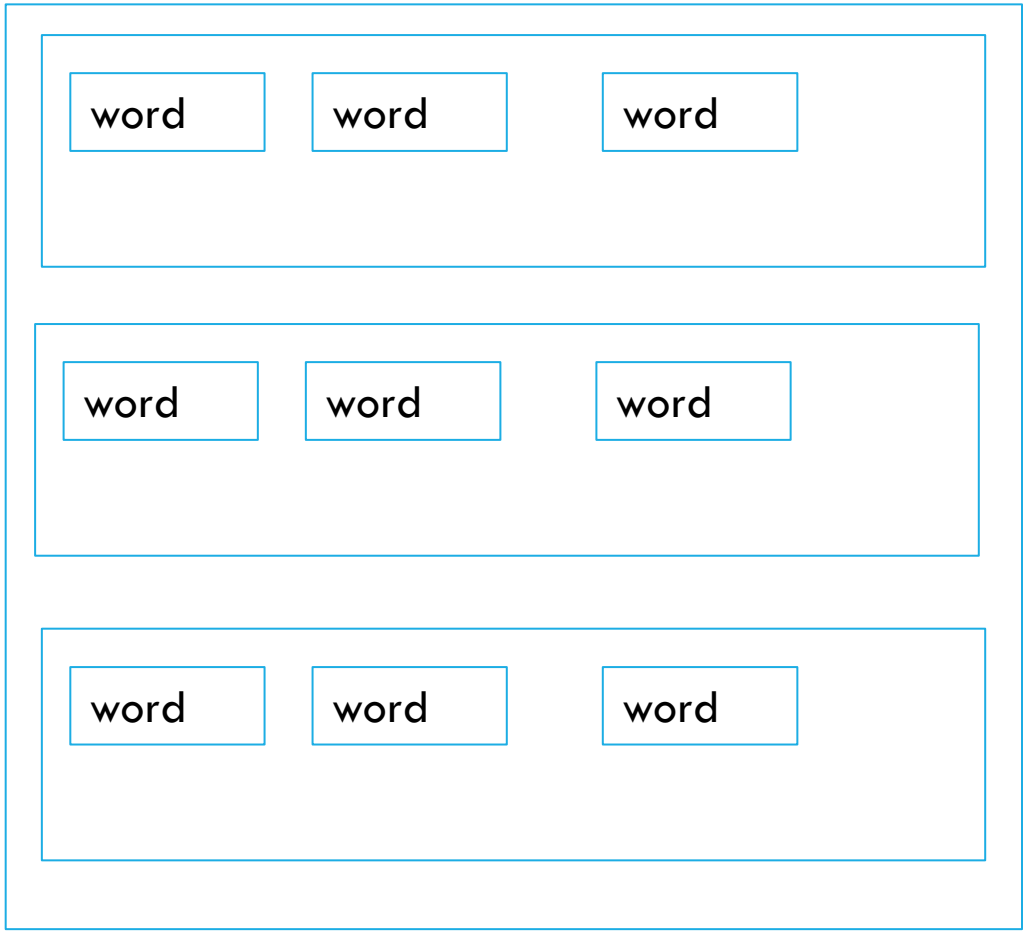
- Stream.of(“Kaushal”, “Bitanu”, “Titir”,”Subhayan”);
- Stream.empty()
- Arrays.stream(1,2,3,4)
- Str.chars()

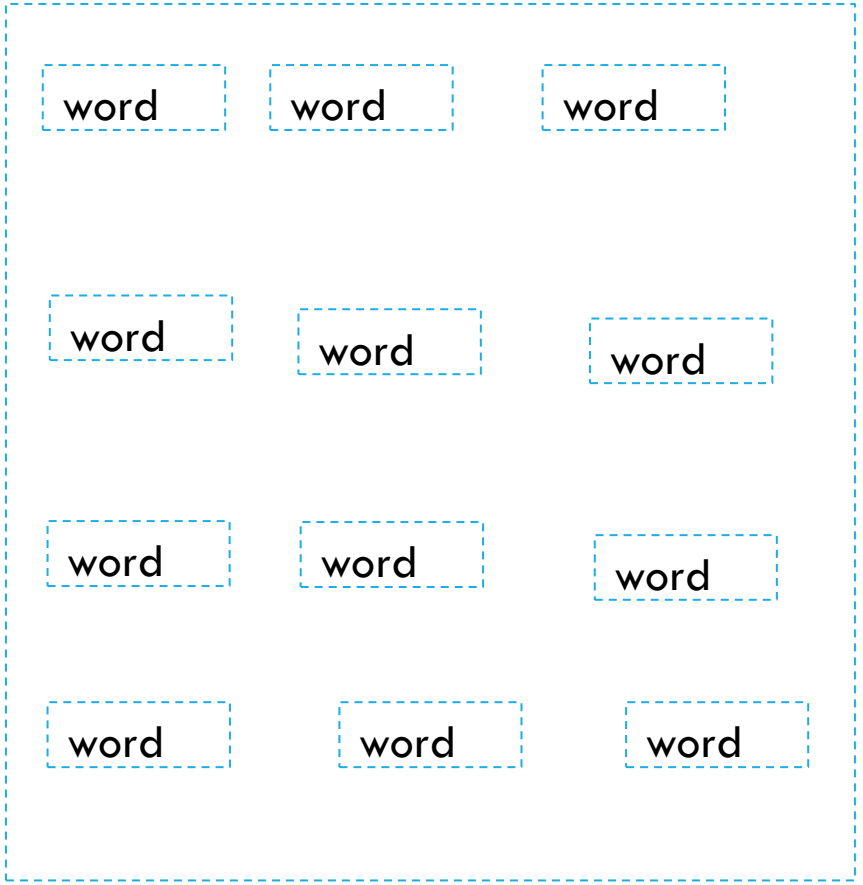
- From files

STREAMS FROM FILES

```
long NoOfUniqueWords =0;
try{
    Stream<String> lines1=
            Files.lines(Paths.get("dataFile.txt"),
            Charset.defaultCharset());
```

```
lines1.map(lines2->

(lines2.split(" ")))
```





```
NoOfUniqueWords =lines1.flatMap(lines2->
                        Arrays.stream(lines2.split(“ ")))
                       .distinct().count();
  System.out.println("1. Unique words are "
                          + NoOfUniqueWords);
            }catch(Exception e){}
```

INFINITE STREAMS

Iterate

- ***Stream.iterate(0, n -> n + 2)***.limit(10).forEach(System.out::println);
- Stream.of(1,2,3,4,5,6,7,8,9,10).?

Fibonacci number

Stream.iterate(new int[]{0, 1}, ???).limit(20)

.forEach(t -> System.out.println("(" + t[0] + "," + t[1] +")"));

t -> new int[]{t[1], t[0]+t[1]}

INFINITE STREAMS

It takes a lambda of type Supplier<T> to provide new values

Stream.generate(Math::random)

.limit(5)

.forEach(System.out::println);

a supplier that’s stateful isn’t safe to use in parallel code