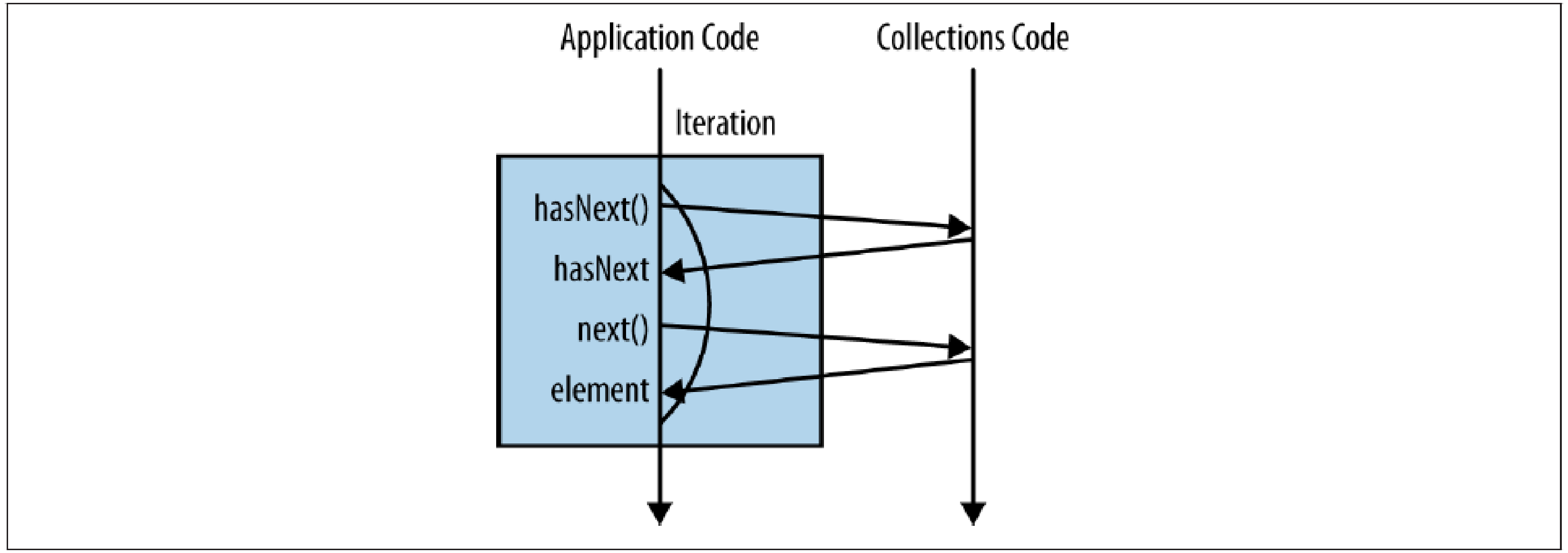
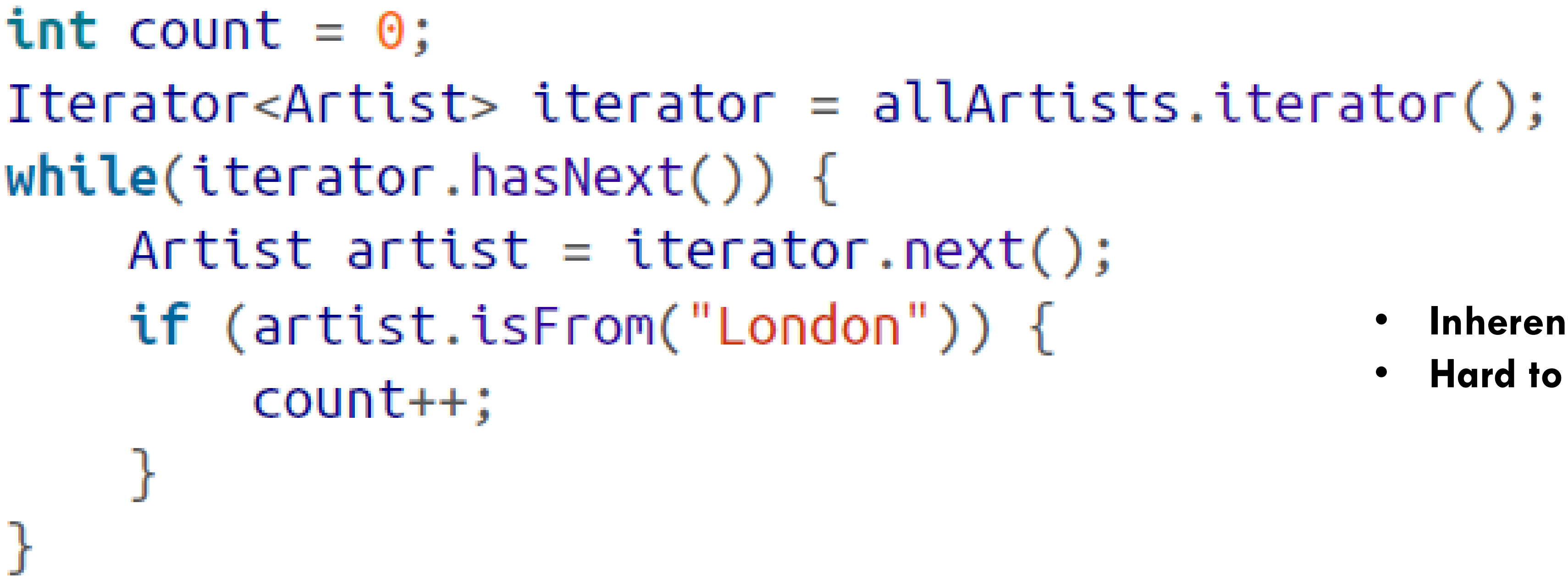
STREAMS AND FUNCTIONAL PROGRAMMING

Chandreyee Chowdhury

EXTERNAL ITERATION

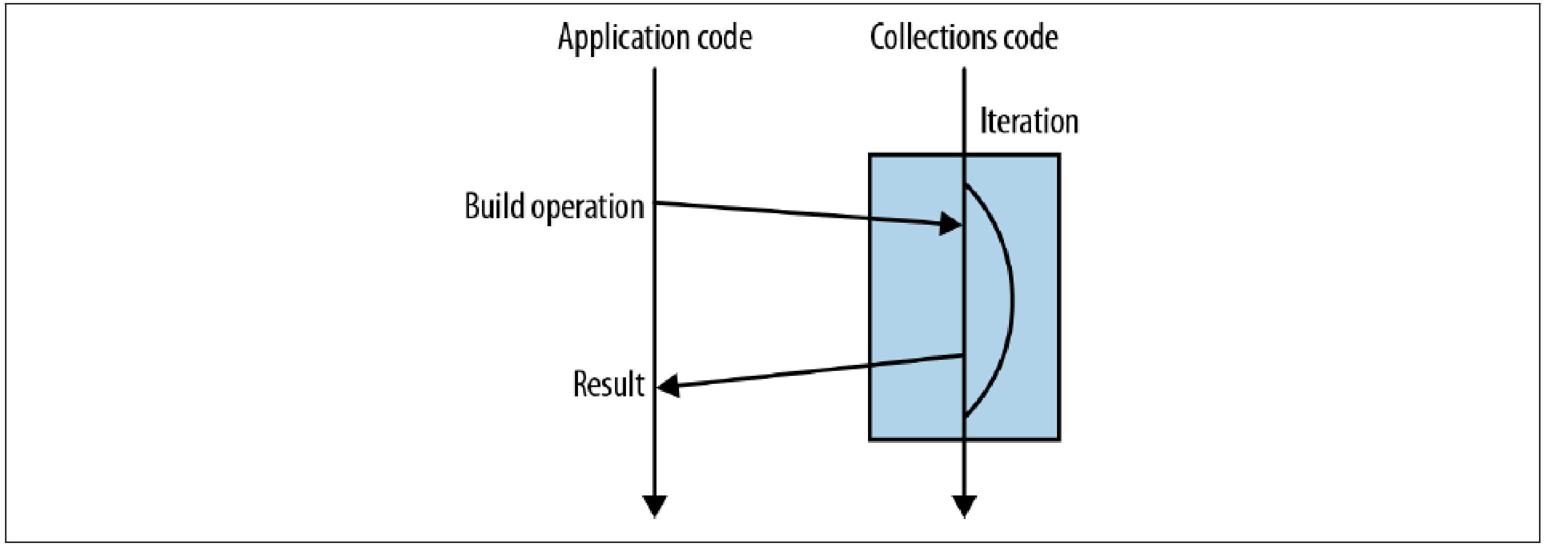


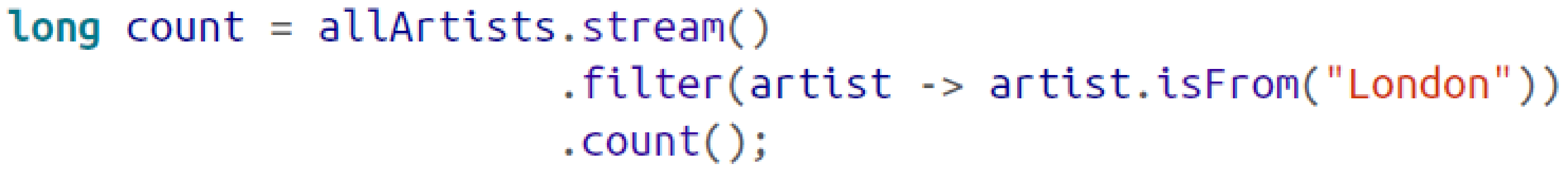


```
int count = 0;
Iterator<Artist> iterator = allArtists.iterator();
while(iterator.hasNext()) {
    Artist artist = iterator.next();
    if (artist.isFrom("London")) {
        count++;
    }
}
```

- **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

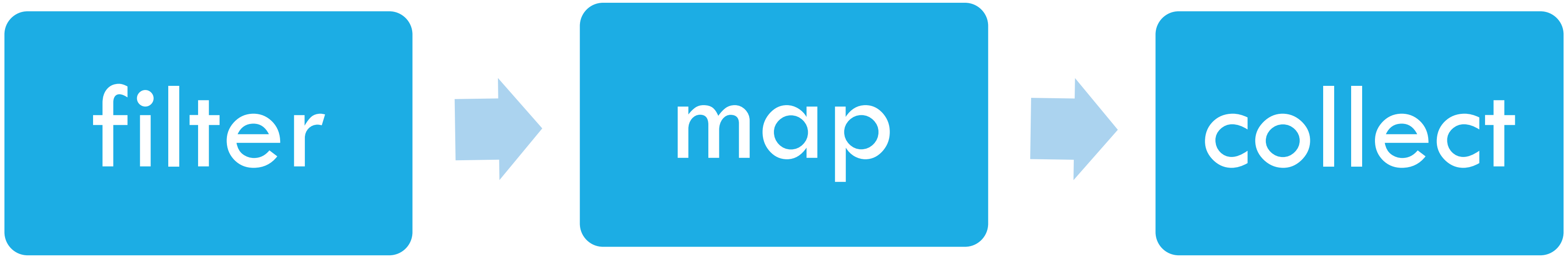
- ❑ Source
  - ❑ Streams consume data from a data providing source such as, collections, arrays, or I/O resources
  - ❑ Streams from an ordered collection preserves the ordering
- ❑ Data processing operations
  - ❑ supports both database like operations and functional programming operations to manipulate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

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

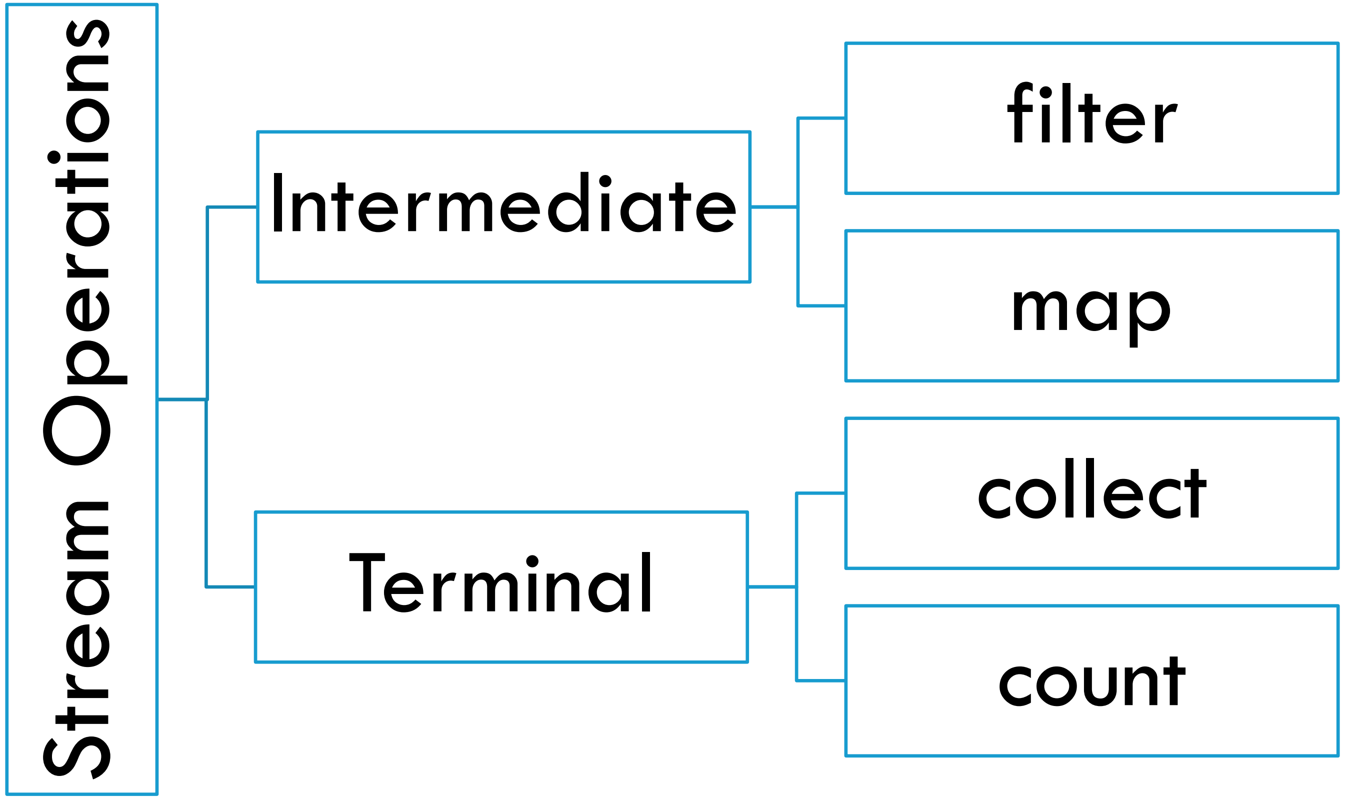
Dish

```
   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



Stream Operations

Intermediate

filter

map

Terminal

collect

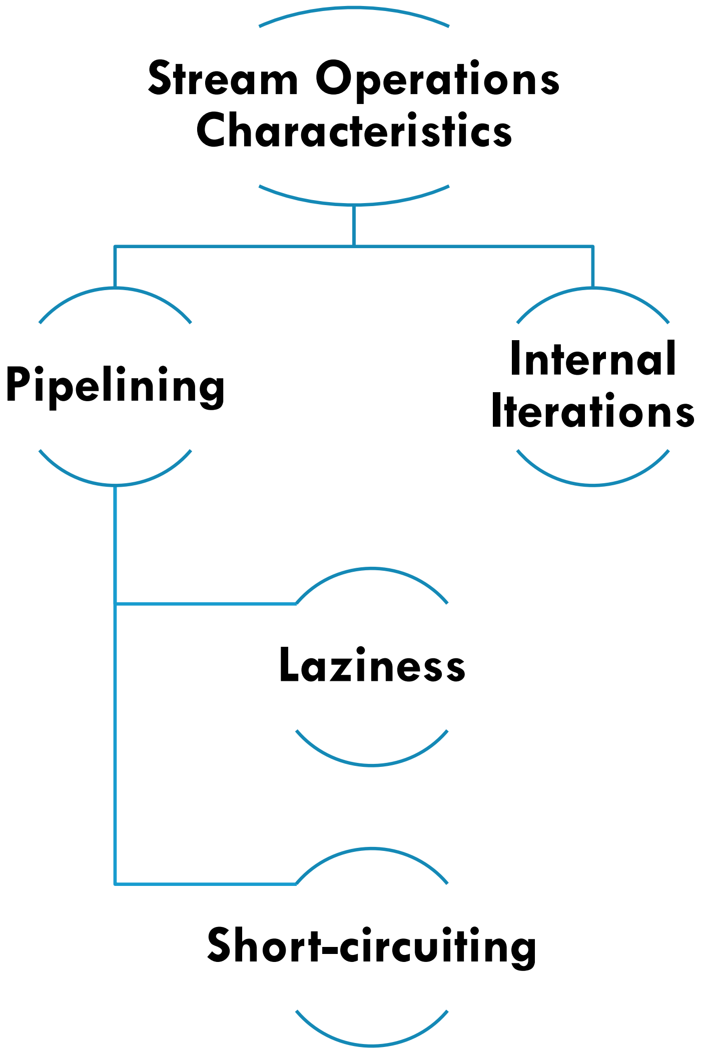
count

```
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

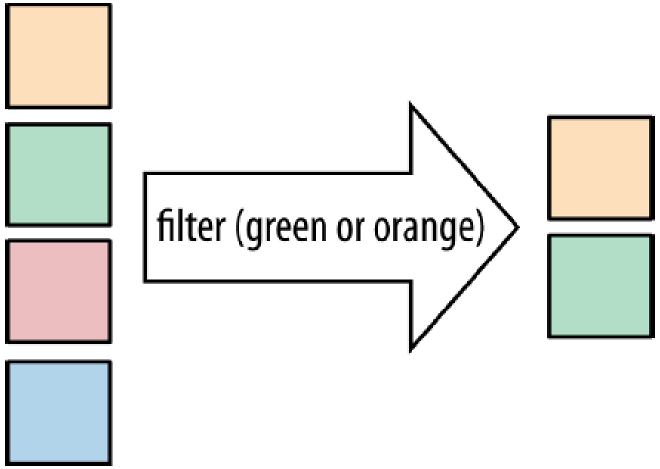
- processing of elements can be done in parallel or in a different order that is more optimized
- stream library can automatically chose a data representation and implementation of parallelism to match the machine hardware

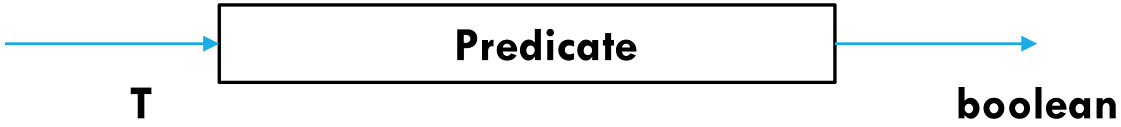
External Iteration

- 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



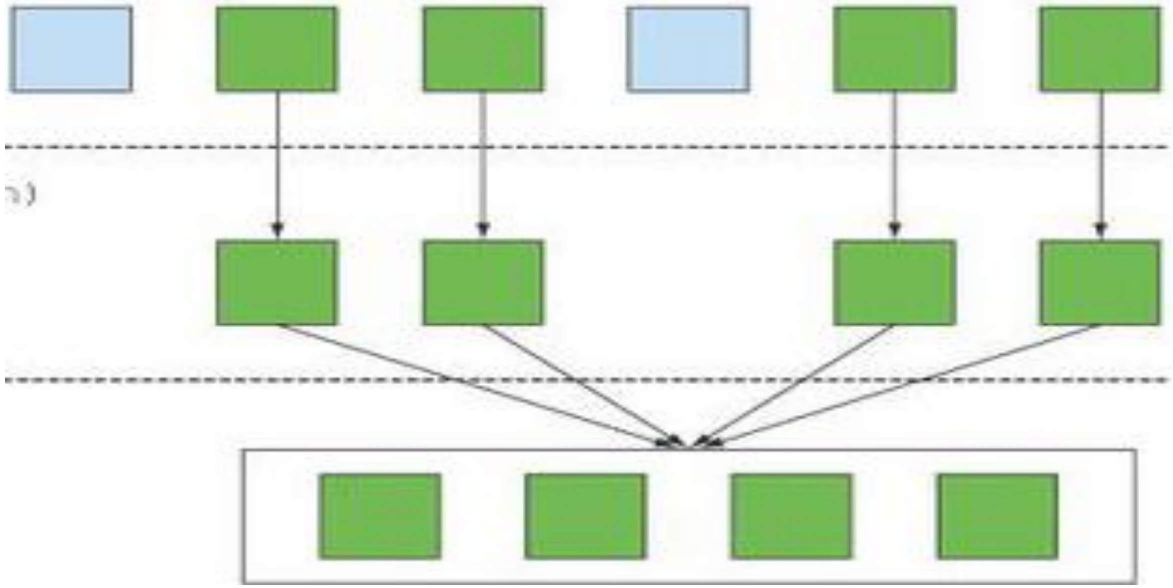

Predicate

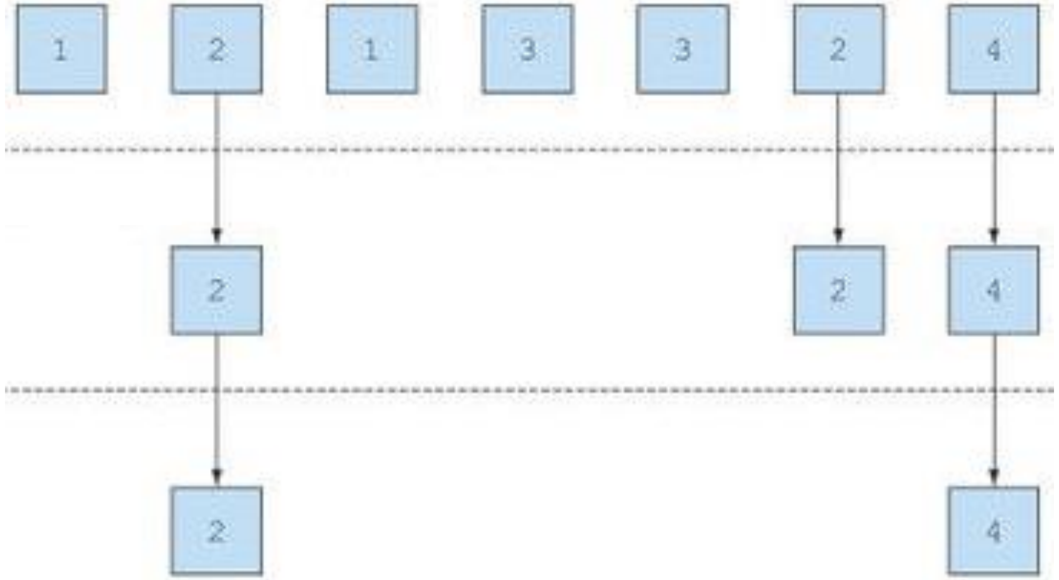
T

boolean

FILTERING

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





SMALL PROBLEMS

Find a list of odd numbers from a list of numbers

Given a list of words

- Extract the list of words that ends with a number
- Extract a list of unique words

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

T
Function
R

```
List<String> collected = new ArrayList<>();
for (String string : asList("a", "b", "hello")) {
    String uppercaseString = string.toUpperCase();
    collected.add(uppercaseString);
}
```

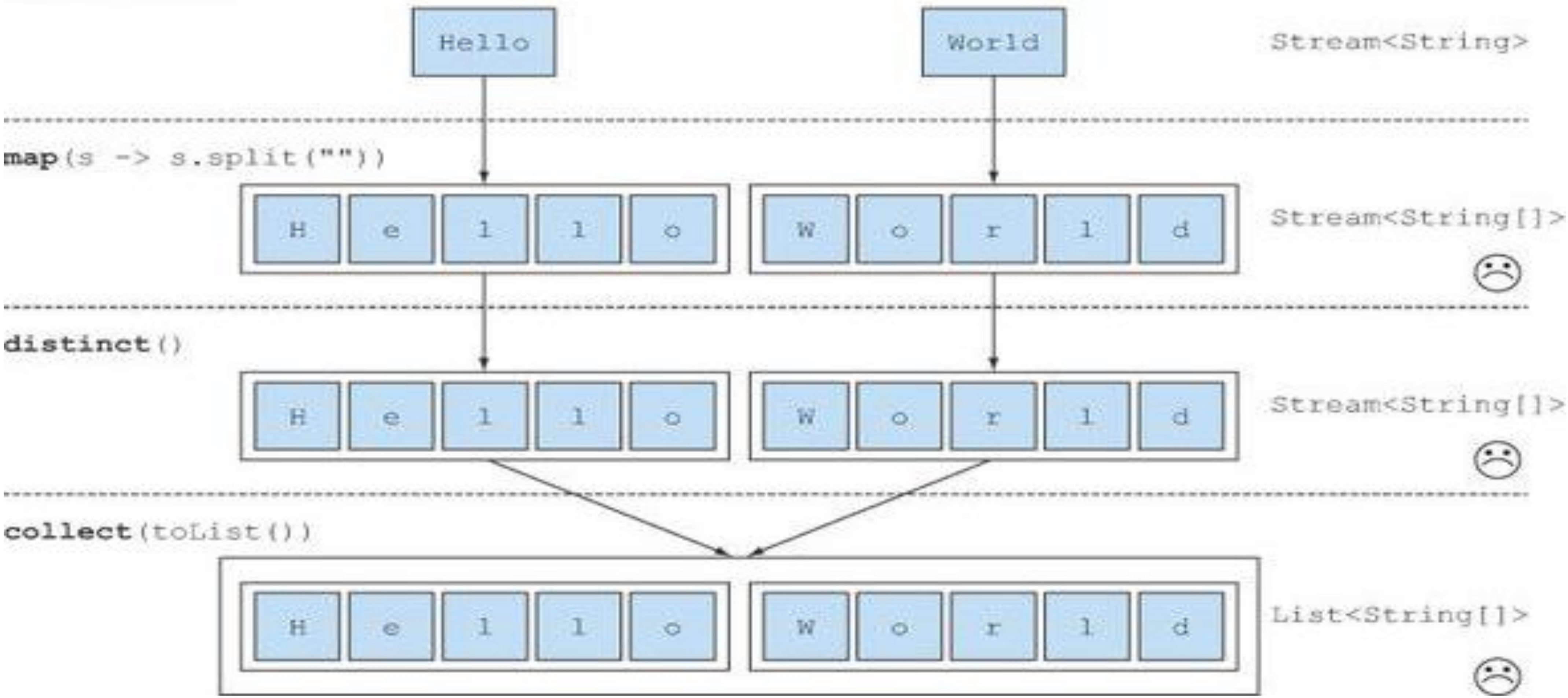
MAPPING

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());
```

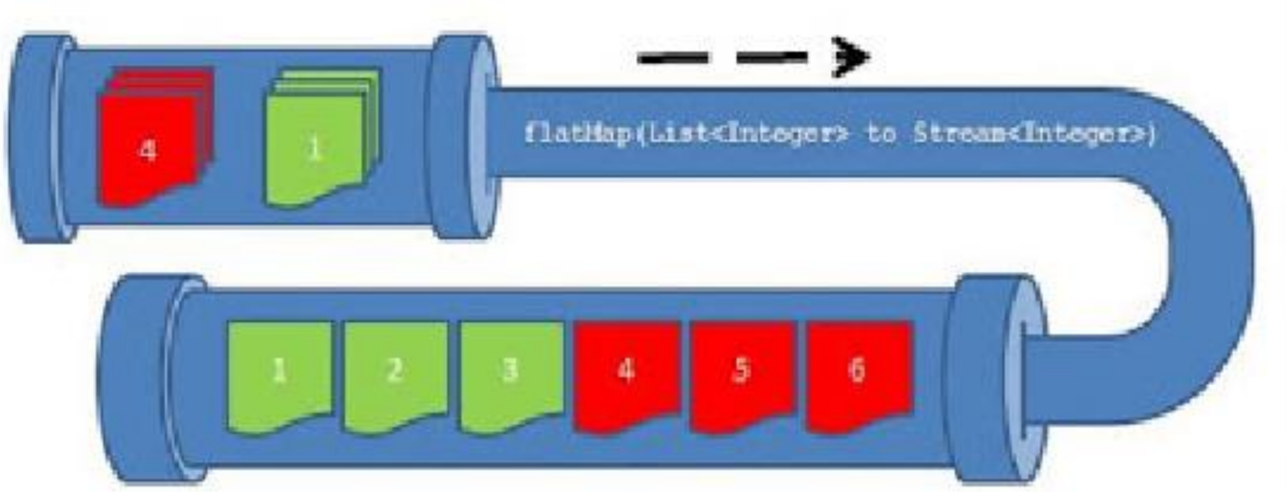
MORE WITH MAPS



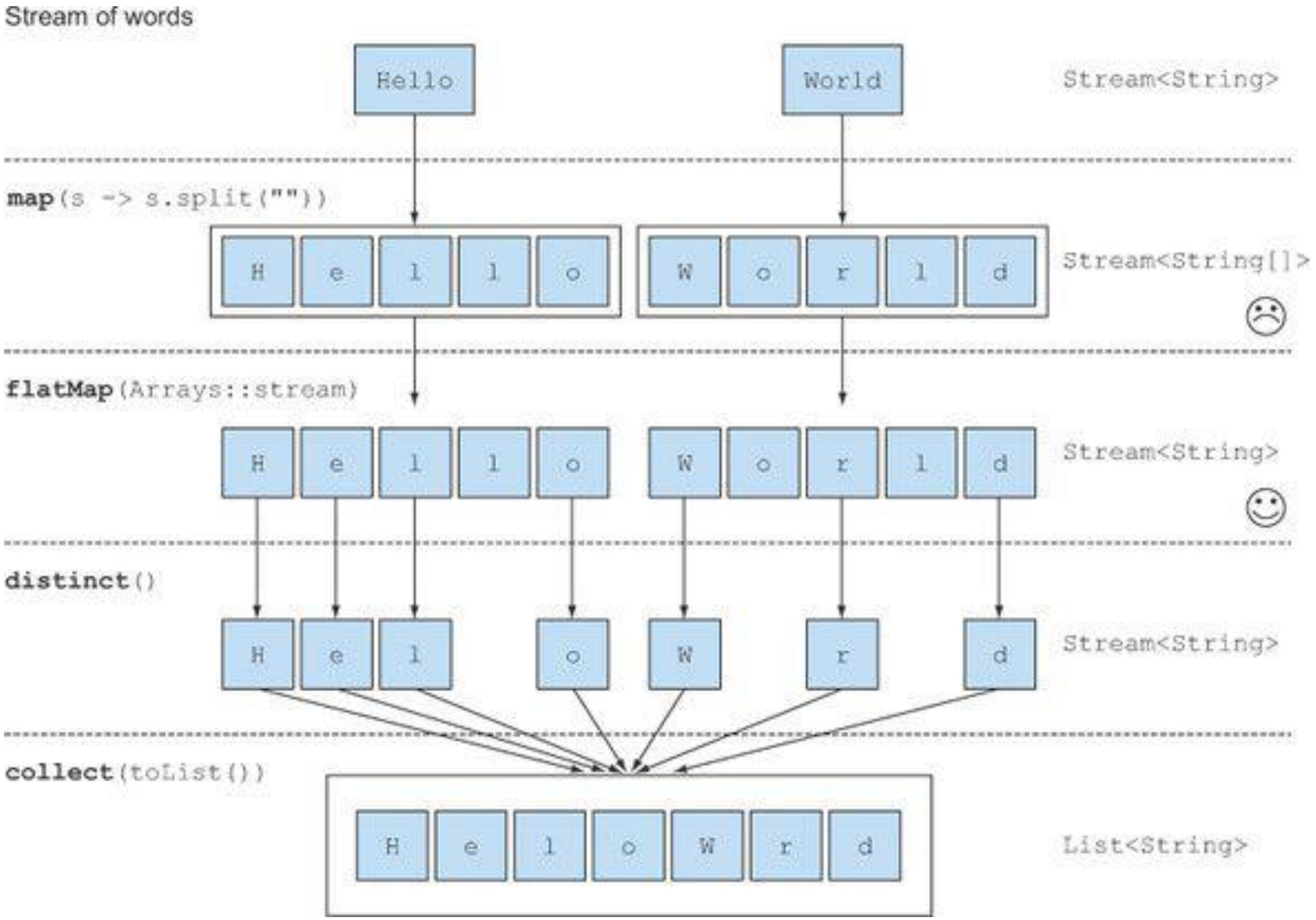
FLATMAP

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



```
map(s -> s.split(""))
flatMap(Arrays::stream)
distinct()
collect(toList())
```

FLATMAP

Form a list of numbers that represents pairwise summations of numbers taking each number from two lists of numbers. Each number should appear exactly once in the list.

```
List<Integer> numbers1=Arrays.asList(1,2,3);
List<Integer> numbers2=Arrays.asList(1,2,3,4);

numbers1.stream().flatMap(i->numbers2.stream().map(j->(i+j)))
                .forEach(k->System.out.println(" " + k));
```

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());
```