5 Working with streams

This chapter covers

- Filtering, slicing, and mapping
- Finding, matching, and reducing
- Using numeric streams (primitive stream specializations)
- Creating streams from multiple sources
- Infinite streams

Strems let you move from external iteration to internal iteration (run your code in parallel)

```
List<Dish> vegetarianDishes = new ArrayList<>();
for(Dish d: menu) {
    if(d.isVegetarian()){
        vegetarianDishes.add(d);
    }
}
```

```
import static java.util.stream.Collectors.toList;
List<Dish> vegetarianDishes =
    menu.stream()
        .filter(Dish::isVegetarian)
        .collect(toList());
```

5.1 Filtering

- The Stream interface supports a filter method taking as argument a predicate (a function mapping a boolean).

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

Use a method reference to check if a dish is vegetarian friendly.

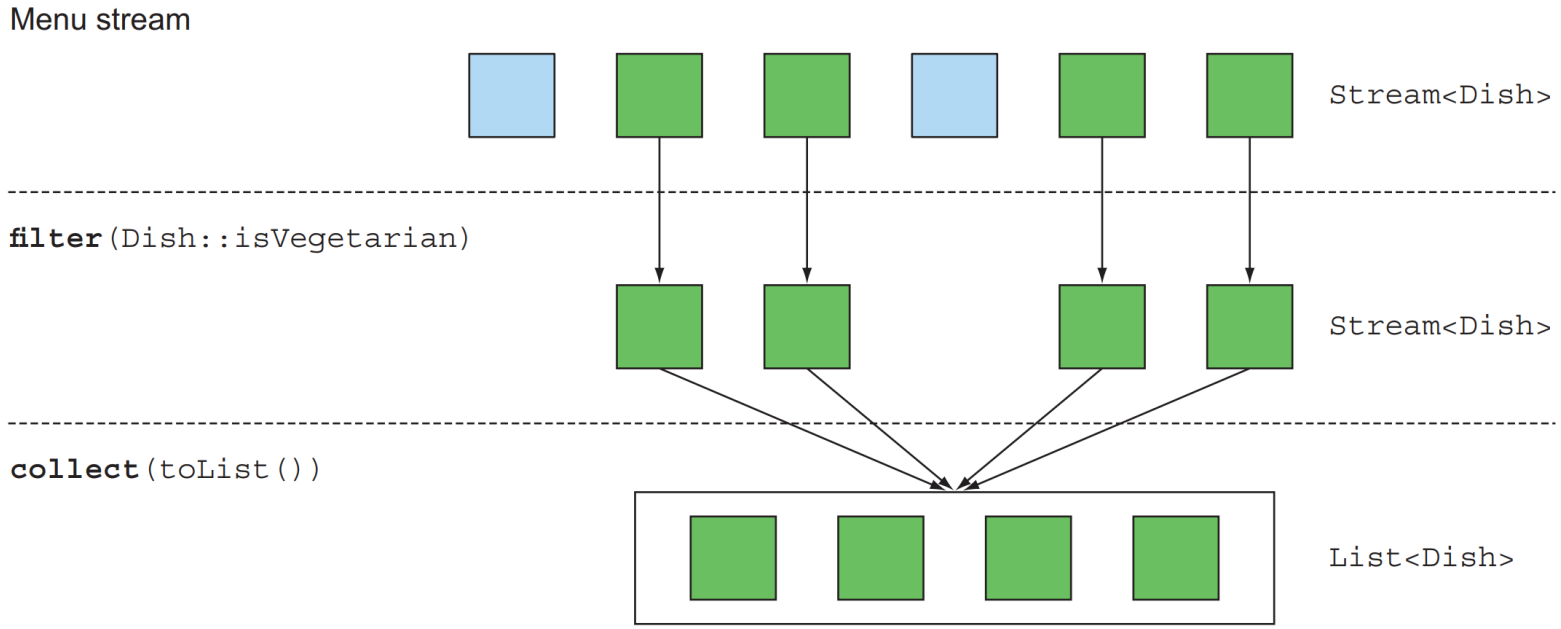


Figure 5.1 Filtering a stream with a predicate

Filtering unique elements

- Streams also support a method called distinct that returns a stream with unique elements (according to the implementation of the hashcode and equals methods of the objects produced by the stream).

```
List<Integer> numbers = Arrays.asList(1, 2, 1, 3, 3, 2, 4);
numbers.stream()
       .filter(i -> i % 2 == 0)
       .distinct()
       .forEach(System.out::println);
```

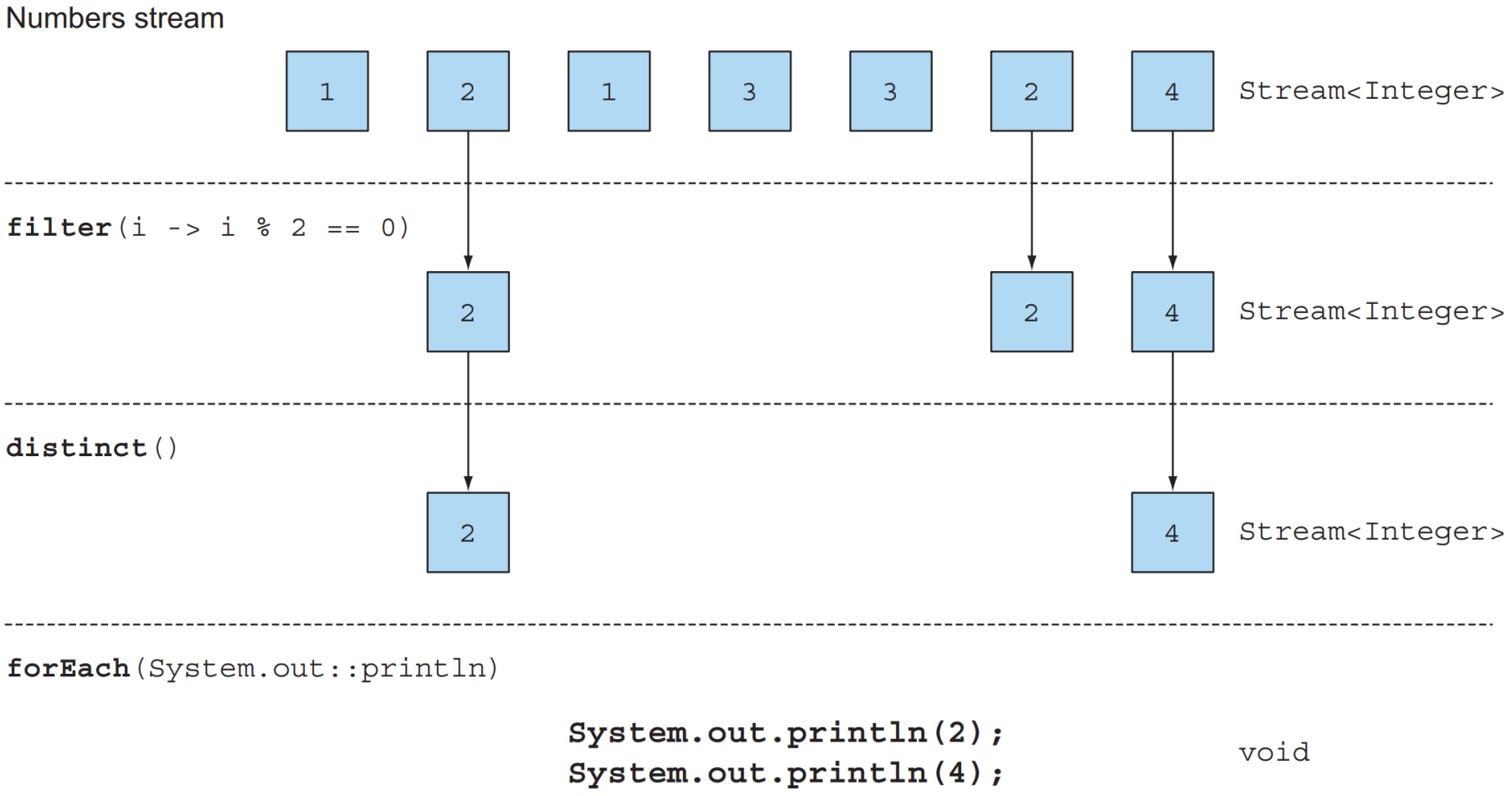


Figure 5.2 Filtering unique elements in a stream

5.2 Slicing a stream

- There are operations:
  - efficiently select or drop elements using a predicate
  - ignore the first few elements of a stream
  - truncate a stream to a given size.
- takeWhile and dropWhile
  - Given the following special list of dishes:

```
List<Dish> specialMenu = Arrays.asList(
    new Dish("seasonal fruit", true, 120, Dish.Type.OTHER),
    new Dish("prawns", false, 300, Dish.Type.FISH),
    new Dish("rice", true, 350, Dish.Type.OTHER),
    new Dish("chicken", false, 400, Dish.Type.MEAT),
    new Dish("french fries", true, 530, Dish.Type.OTHER));
```

Select the dishes that have fewer than 320 calories.

```
List<Dish> filteredMenu
    = specialMenu.stream()
                 .filter(dish -> dish.getCalories() < 320)
                 .collect(toList());
```

Lists seasonal fruit, prawns

- Support you have the initial list which was already sorted on the number of calories!
  - Using takeWhile

```
List<Dish> slicedMenu1
    = specialMenu.stream()
                 .takeWhile(dish -> dish.getCalories() < 320)
                 .collect(toList());
```

Lists seasonal fruit, prawns

  - Using dropWhile (the complement of takeWhile)

```
List<Dish> slicedMenu2
    = specialMenu.stream()
                 .dropWhile(dish -> dish.getCalories() < 320)
                 .collect(toList());
```

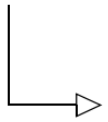
Lists rice, chicken, french fries

Truncating a stream

- limit(n): returns another stream that's no longer than a given size.

```
List<Dish> dishes = specialMenu
                        .stream()
                        .filter(dish -> dish.getCalories() > 300)
                        .limit(3)
                        .collect(toList());
```

Lists rice, chicken, french fries



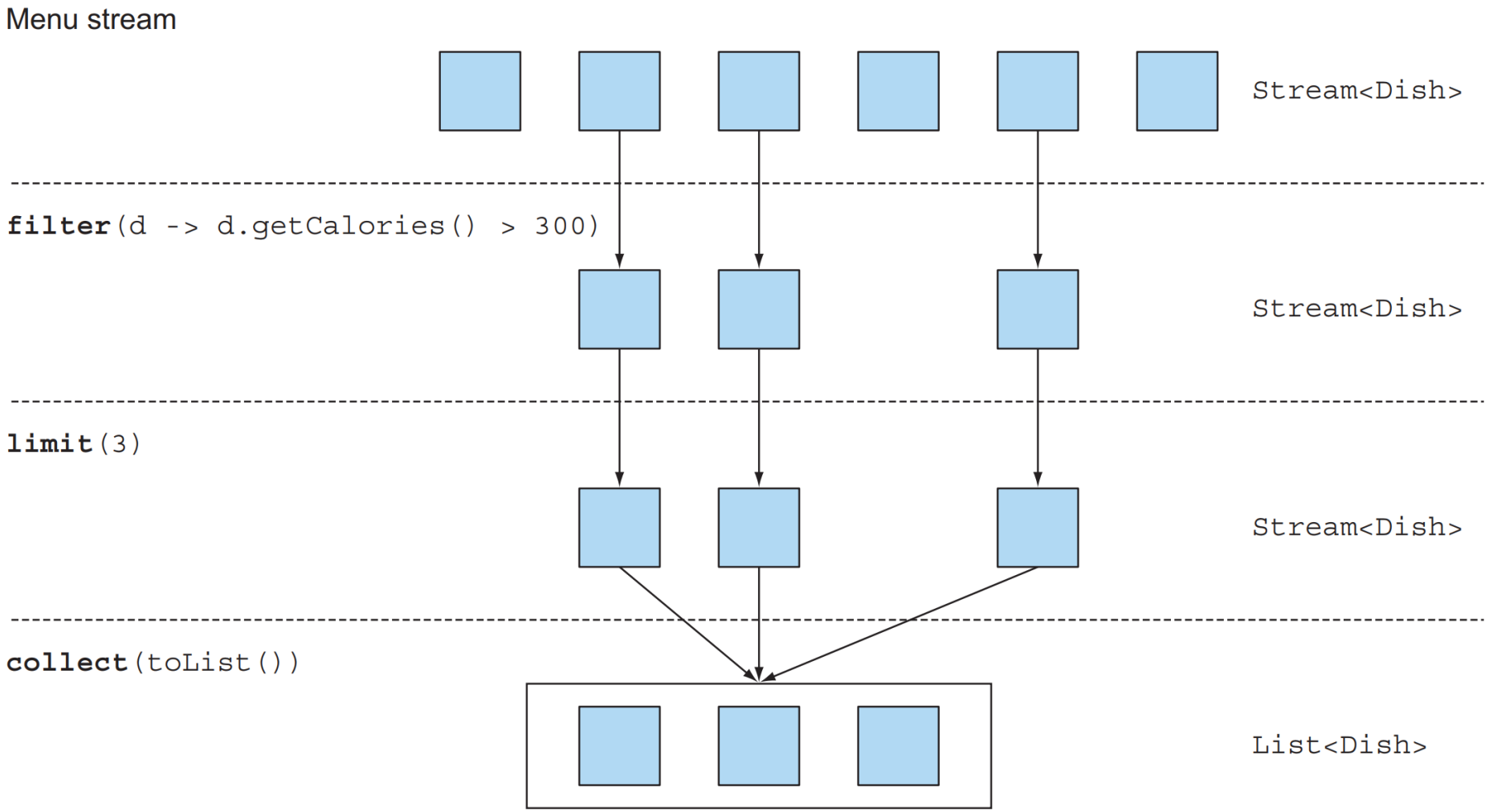


Figure 5.3 Truncating a stream

Skipping elements

- skip(n): returns a stream that discards the first n elements.

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

- limit(n) and skip(n) are complementary!

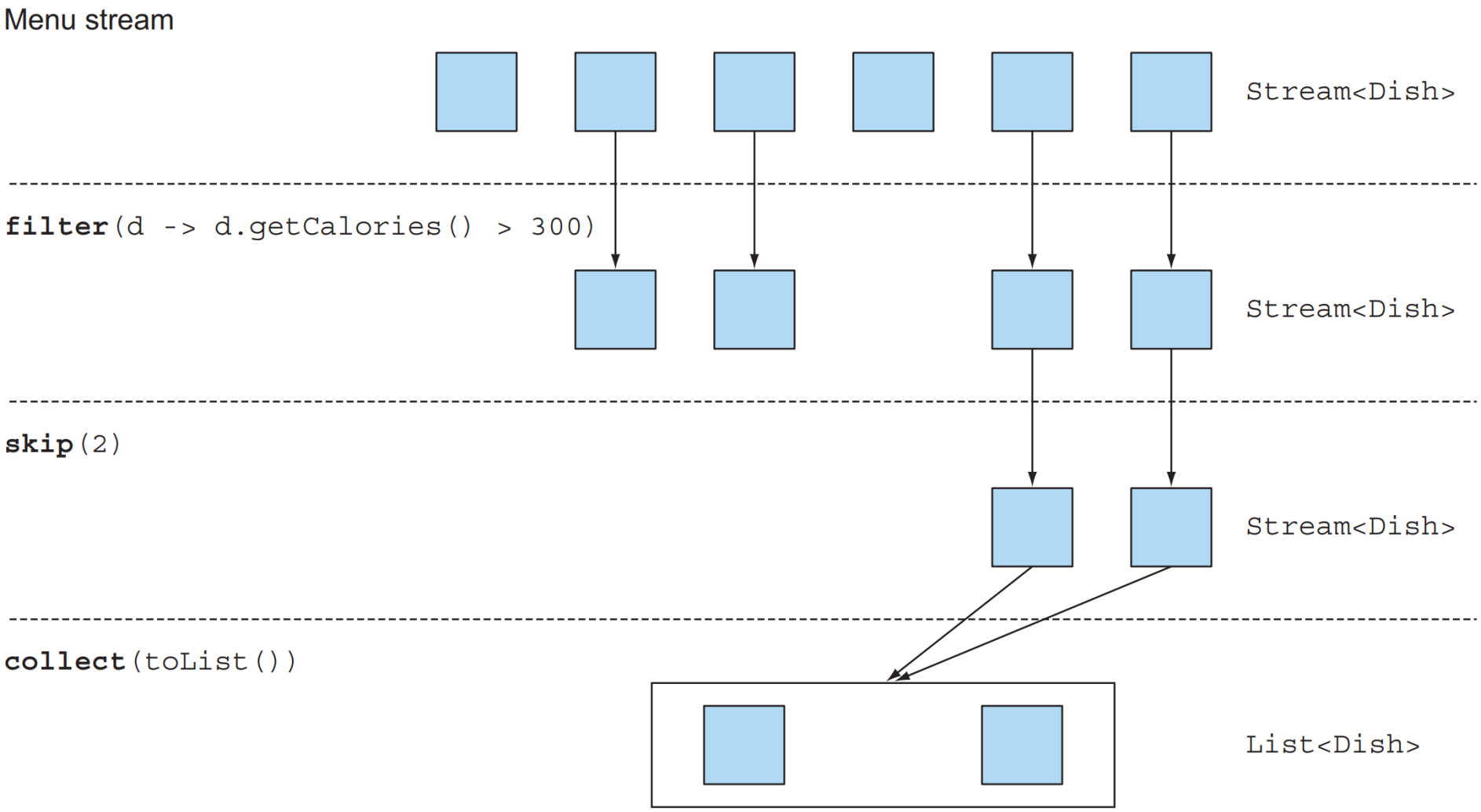


Figure 5.4 Skipping elements in a stream

5.3 Mapping

- map, which takes a function as argument.
  - Pass a method reference Dish::getName to the map method to *extract* the names of the dishes in the stream.

```
List<String> dishNames = menu.stream()
                              .map(Dish::getName)
                              .collect(toList());
```

  - Pass a method reference String::length to map:

```
List<String> words = Arrays.asList("Modern", "Java", "In", "Action");
List<Integer> wordLengths = words.stream()
                                 .map(String::length)
                                 .collect(toList());
```

```
List<Integer> dishNameLengths = menu.stream()
                                    .map(Dish::getName)
                                    .map(String::length)
                                    .collect(toList());
```

Flattening streams

- How could you return a list of all the *unique characters* for a list of words?
  For example, given the list of words ["Hello," "World"] you'd like to return the list ["H," "e," "l," "o," "W," "r," "d"].

```
words.stream()
     .map(word -> word.split(""))
     .distinct()
     .collect(toList());
```

- The stream returned by the map method is of type Stream<String[]>.
- What we want is Stream<String>.

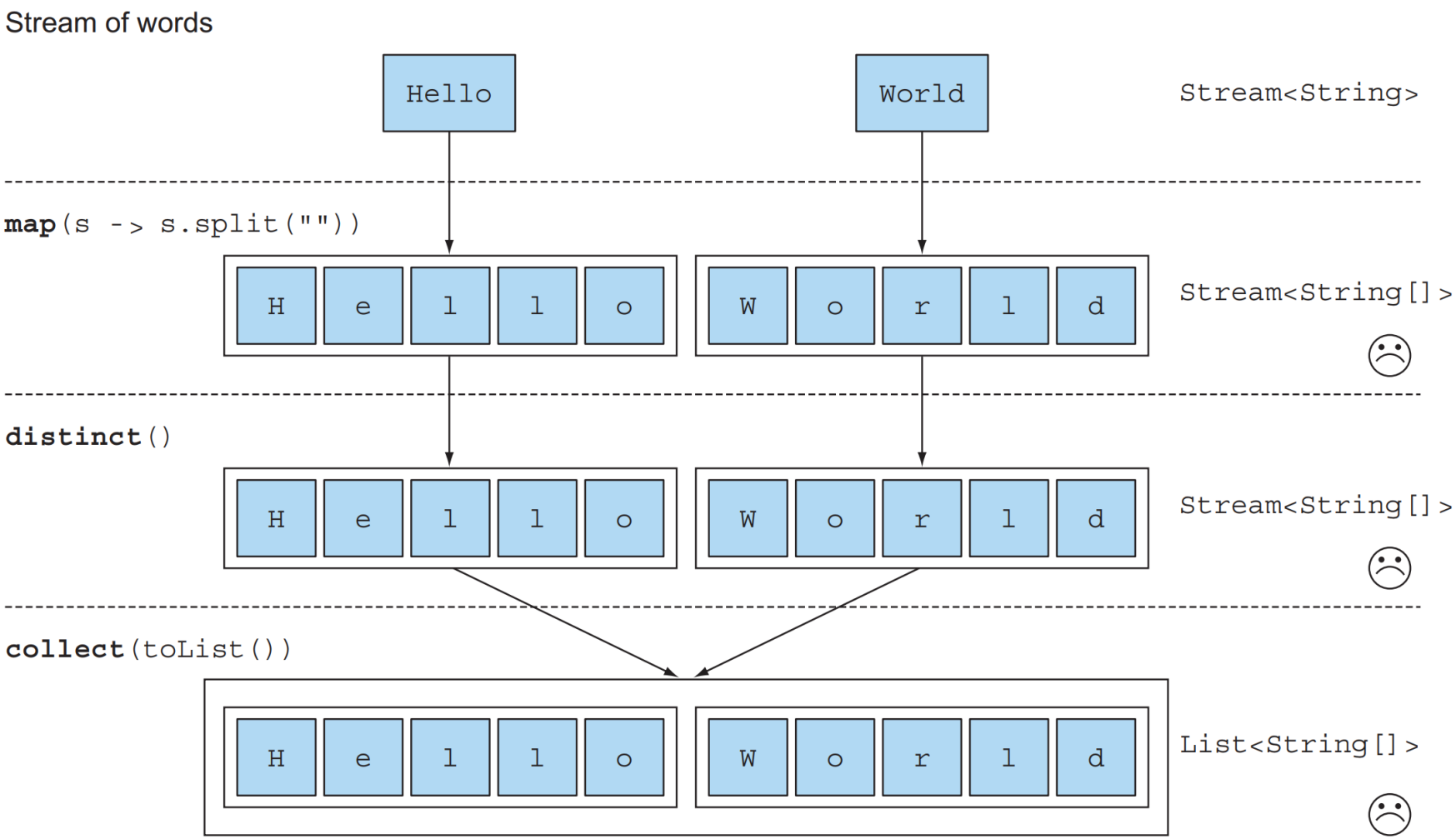


Figure 5.5 Incorrect use of `map` to find unique characters from a list of words

Attempt Using Map and Arrays.Stream

```
String[] arrayOfWords = {"Goodbye", "World"};
Stream<String> streamOfwords = Arrays.stream(arrayOfWords);
```

Use it in the previous pipeline to see what happens:

```
words.stream()
     .map(word -> word.split(""))
     .map(Arrays::stream)
     .distinct()
     .collect(toList());
```

Converts each word into an array of its individual letters

Makes each array into a separate stream

You now end up with a list of streams (more precisely, List<Stream<String>>).

Using flatMap

You can fix this problem by using `flatMap` as follows:

```
List<String> uniqueCharacters =
  words.stream()
       .map(word -> word.split(""))
       .flatMap(Arrays::stream)
       .distinct()
       .collect(toList());
```

Converts each word into an array of its individual letters

Flattens each generated stream into a single stream

Using the flatMap method has the effect of mapping each array not with a stream but *with the contents of that stream*.

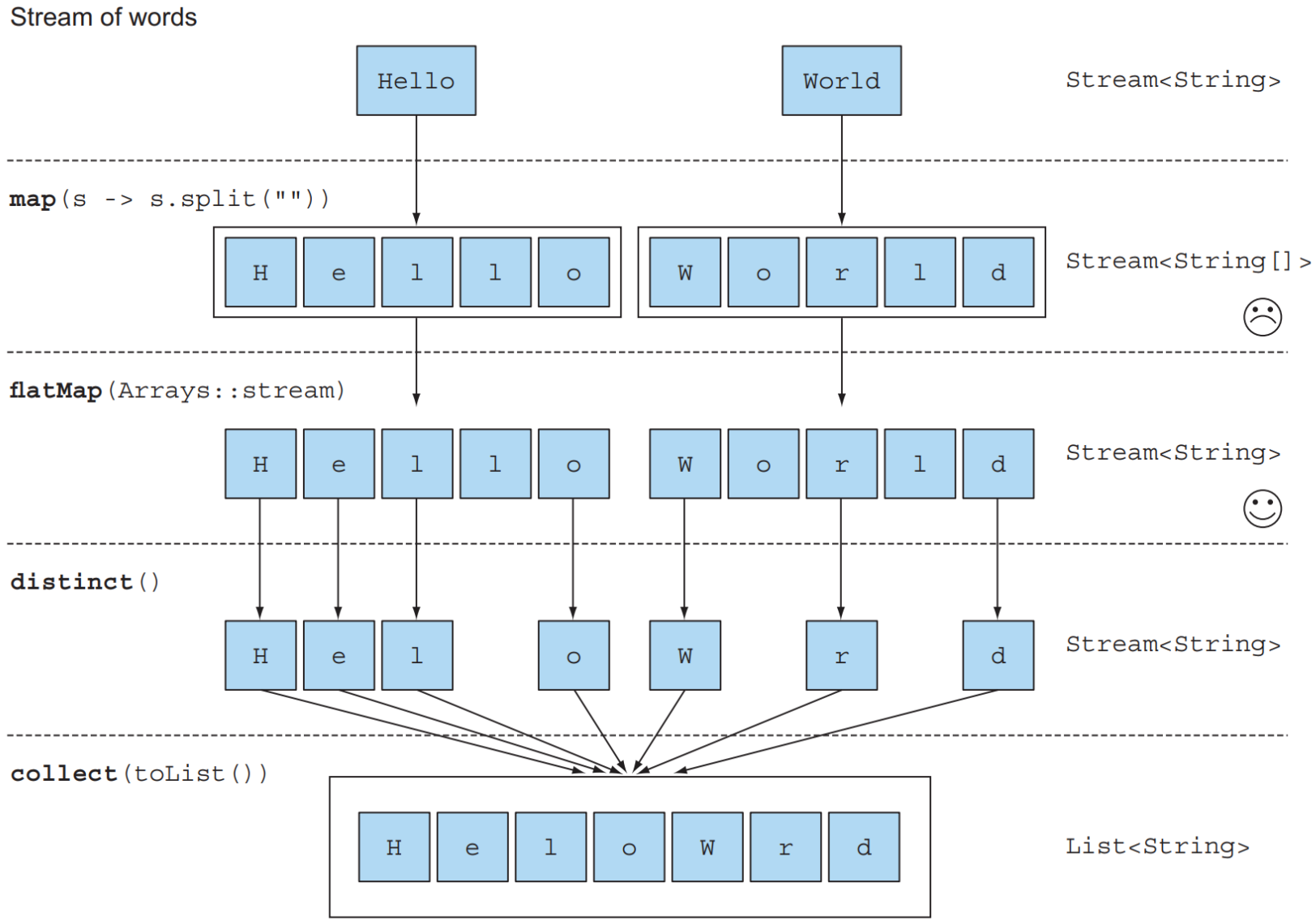


Figure 5.6 Using `flatMap` to find the unique characters from a list of words

5.4 Finding and matching

- The Streams API provides such facilities through the `allMatch`, `anyMatch`, `noneMatch`, `findFirst`, and `findAny` methods of a stream.
- *Checking to see if a predicate matches at least one element*

```
if(menu.stream().anyMatch(Dish::isVegetarian)) {
    System.out.println("The menu is (somewhat) vegetarian friendly!!");
}
```

The `anyMatch` method returns a `boolean` and is therefore a terminal operation.

- *Checking to see if a predicate matches all elements*
  - For example, you can use it to find out whether the menu is healthy (all dishes are below 1000 calories):

```
boolean isHealthy = menu.stream()
                        .allMatch(dish -> dish.getCalories() < 1000);
```

  - noneMatch (The opposite of allMatch)

```
boolean isHealthy = menu.stream()
                        .noneMatch(d -> d.getCalories() >= 1000);
```

  - These three operations—anyMatch, allMatch, and noneMatch—make use of what we call *short-circuiting*, a stream version of the familiar Java short-circuiting && and || operators.

Finding an element

- findAny: returns an arbitrary element of the current stream.
- For example, you may wish to find a dish that's vegetarian. You can combine the `filter` method and `findAny` to express this query:

```
Optional<Dish> dish =
   menu.stream()
       .filter(Dish::isVegetarian)
       .findAny();
```

- OPTIONAL IN A NUTSHELL
  The Optional<T> class (java.util.Optional) is a container class to represent the existence or absence of a value.

- A few methods available in Optional that force you to explicitly check for the presence of a value or deal with the absence of a value:
  - `isPresent()` returns `true` if `Optional` contains a value, `false` otherwise.
  - `ifPresent(Consumer<T> block)` executes the given block if a value is present.
  - `T get()` returns the value if present; otherwise it throws a `NoSuchElementException`.
  - `T orElse(T other)` returns the value if present; otherwise it returns a default value

```
menu.stream()
    .filter(Dish::isVegetarian)
    .findAny()
    .ifPresent(dish -> System.out.println(dish.getName()));
```

Returns an Optional<Dish>.

If a value is contained, it's printed; otherwise nothing happens.

Finding the first element

- for example, the code that follows, given a list of numbers, finds the first square that's divisible by 3):

```
List<Integer> someNumbers = Arrays.asList(1, 2, 3, 4, 5);
Optional<Integer> firstSquareDivisibleByThree =
  someNumbers.stream()
             .map(n -> n * n)
             .filter(n -> n % 3 == 0)
             .findFirst(); // 9
```

5.5 Reducing

- The terminal operations
  - return a `boolean` (`allMatch` and so on)
  - `void` (`forEach`),
  - an `Optional` object (`findAny` and so on).
  - using `collect` to combine all elements in a stream into a List.
- reduction operations
  - Calculate the sum of all calories in the menu," or "What is the highest calorie dish in the menu?" using the `reduce` operation.
  - A stream is reduced to a value.

Summing the elements

- Sum the elements of a list of numbers using a for-each loop:

```
int sum = 0;          // the initial values of the sum variable
for (int x :numbers) {
    sum += x;         // the operation to combine all the elements in the list
}
```

- `int sum = numbers.stream().reduce(0, (a, b) -> a + b);`
  - An initial value, here 0
  - A BinaryOperator<T> to combine two elements and produce a new value; here you use the lambda `(a, b) -> a + b`
- `int product = numbers.stream().reduce(1, (a, b) -> a * b);`
- `int sum = numbers.stream().reduce(0, Integer::sum);  // method reference`
- `Optional<Integer> sum = numbers.stream().reduce((a, b) -> a + b);`

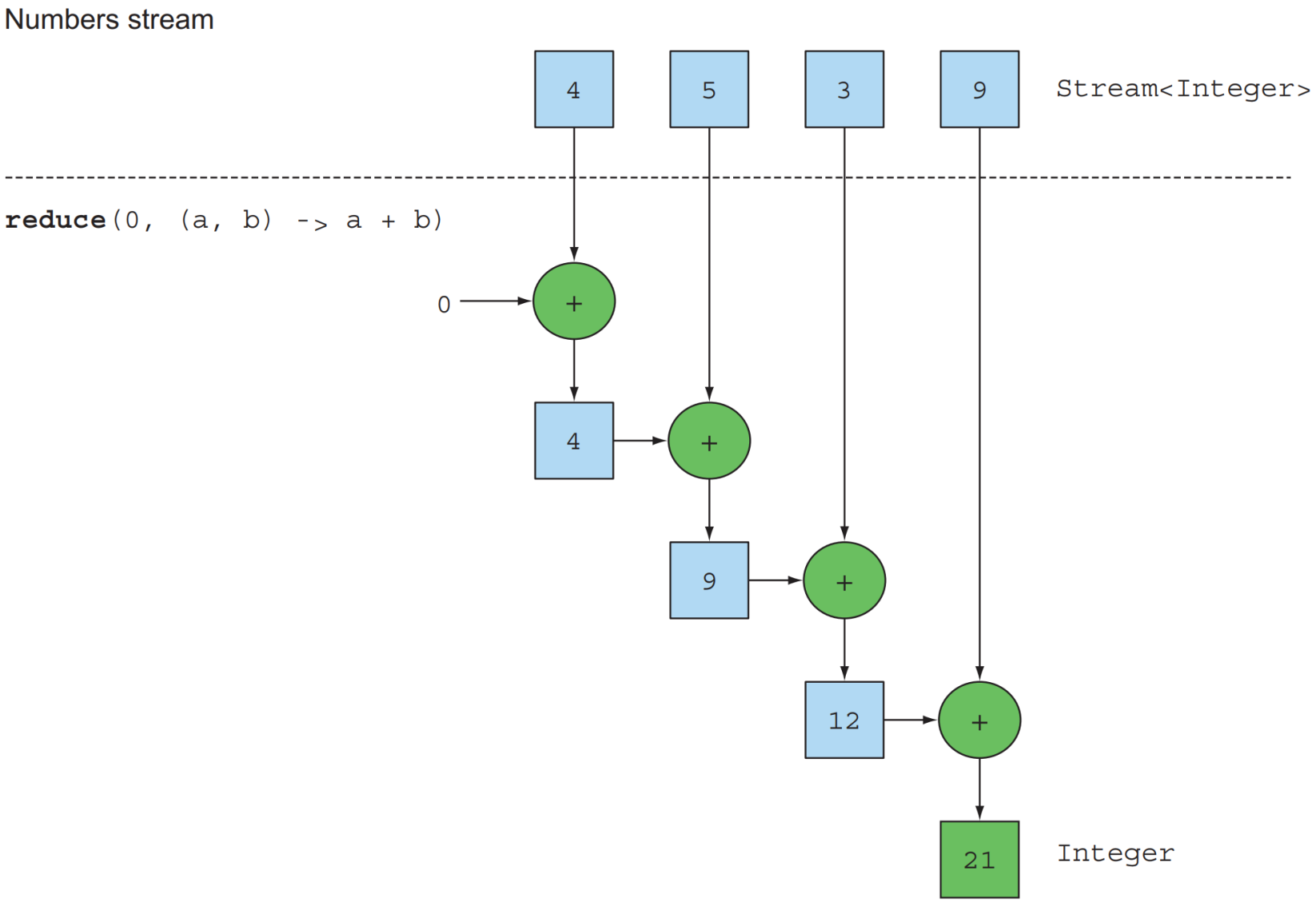


Figure 5.7 Using `reduce` to sum the numbers in a stream

Maximum and Minimum

- `reduce` takes two parameters:
  - An initial value
  - A lambda to combine two stream elements and produce a new value
- `Optional<Integer> max = numbers.stream().reduce(Integer::max);`
- `Optional<Integer> min = numbers.stream().reduce(Integer::min);`
- We'll investigate a more complex form of reductions using the `collect` method in the next chapter.
  - For example, instead of reducing a stream into an Integer, you can also reduce it into a Map if you want to group dishes by types.

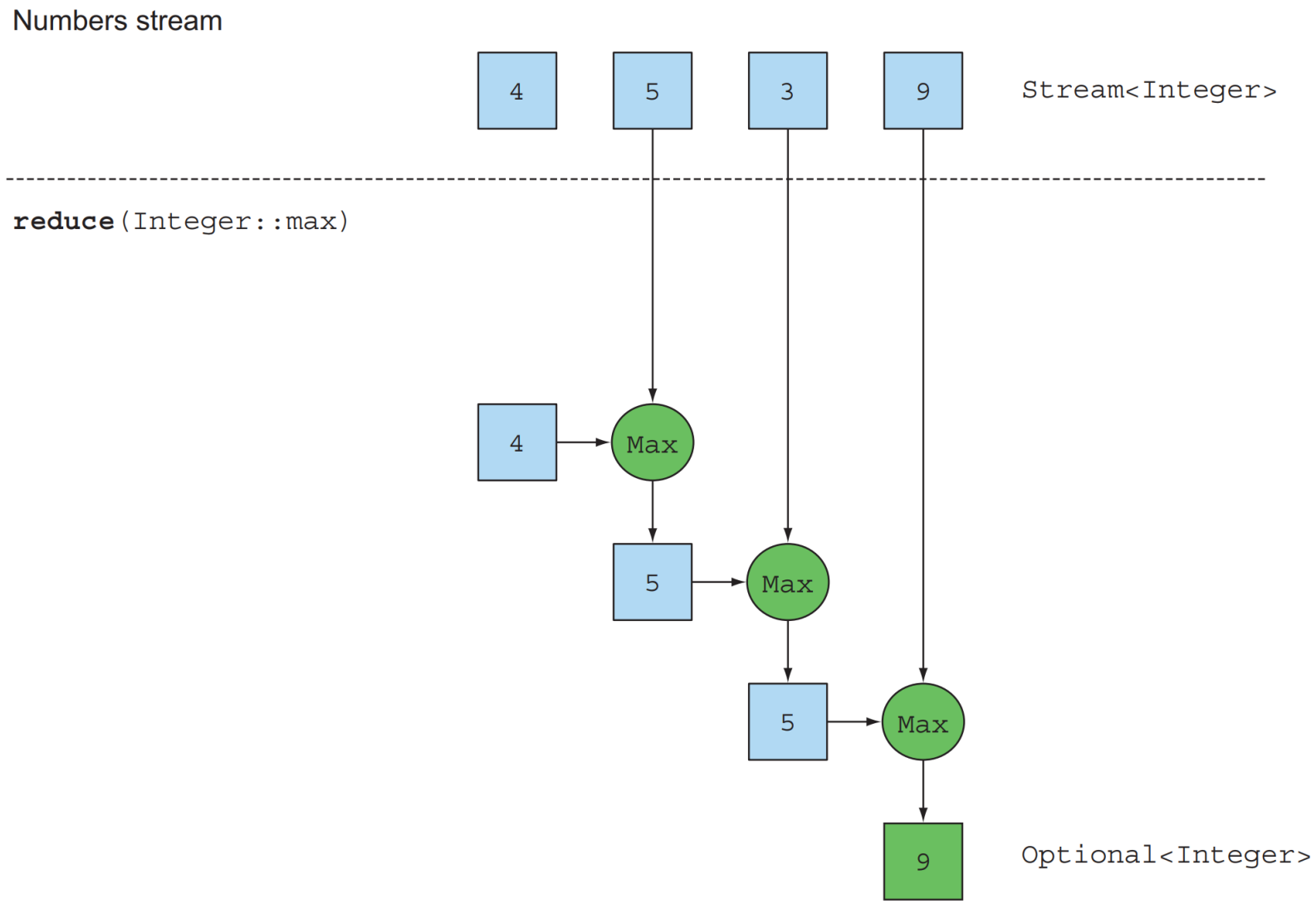


Figure 5.8 A `reduce` operation—calculating the maximum

Table 5.1 Intermediate and terminal operations

Operation	Type	Return type	Type/functional interface used	Function descriptor
`filter`	Intermediate	`Stream<T>`	`Predicate<T>`	`T -> boolean`
`distinct`	Intermediate (stateful-unbounded)	`Stream<T>`		
`takeWhile`	Intermediate	`Stream<T>`	`Predicate<T>`	`T -> boolean`
`dropWhile`	Intermediate	`Stream<T>`	`Predicate<T>`	`T -> boolean`
`skip`	Intermediate (stateful-bounded)	`Stream<T>`	`long`	
`limit`	Intermediate (stateful-bounded)	`Stream<T>`	`long`	
`map`	Intermediate	`Stream<R>`	`Function<T, R>`	`T -> R`
`flatMap`	Intermediate	`Stream<R>`	`Function<T, Stream<R>>`	`T -> Stream<R>`
`sorted`	Intermediate (stateful-unbounded)	`Stream<T>`	`Comparator<T>`	`(T, T) -> int`
`anyMatch`	Terminal	`boolean`	`Predicate<T>`	`T -> boolean`
`noneMatch`	Terminal	`boolean`	`Predicate<T>`	`T -> boolean`

Table 5.1 Intermediate and terminal operations *(continued)*

Operation	Type	Return type	Type/functional interface used	Function descriptor
`allMatch`	Terminal	`boolean`	`Predicate<T>`	`T -> boolean`
`findAny`	Terminal	`Optional<T>`		
`findFirst`	Terminal	`Optional<T>`		
`forEach`	Terminal	`void`	`Consumer<T>`	`T -> void`
`collect`	Terminal	`R`	`Collector<T, A, R>`	
`reduce`	Terminal (stateful-bounded)	`Optional<T>`	`BinaryOperator<T>`	`(T, T) -> T`
`count`	Terminal	`long`		

5.6 Putting it all into practice

1. Find all transactions in the year 2011 and sort them by value (small to high).
2. What are all the unique cities where the traders work?
3. Find all traders from Cambridge and sort them by name.
4. Return a string of all traders' names sorted alphabetically.
5. Are any traders based in Milan?
6. Print the values of all transactions from the traders living in Cambridge.
7. What's the highest value of all the transactions?
8. Find the transaction with the smallest value.

The domain: Traders and Transactions

Here's the domain you'll be working with, a list of `Traders` and `Transactions`:

```
Trader raoul = new Trader("Raoul", "Cambridge");
Trader mario = new Trader("Mario","Milan");
Trader alan = new Trader("Alan","Cambridge");
Trader brian = new Trader("Brian","Cambridge");
List<Transaction> transactions = Arrays.asList(
    new Transaction(brian, 2011, 300),
    new Transaction(raoul, 2012, 1000),
    new Transaction(raoul, 2011, 400),
    new Transaction(mario, 2012, 710),
    new Transaction(mario, 2012, 700),
    new Transaction(alan, 2012, 950)
);
```

Trader and Transaction are classes defined as follows:

```
public class Trader{
    private final String name;
    private final String city;
    public Trader(String n, String c){
        this.name = n;
        this.city = c;
    }
    public String getName(){
        return this.name;
    }
    public String getCity(){
        return this.city;
    }
    public String toString(){
        return "Trader:"+this.name + " in " + this.city;
    }
}
```

```
public class Transaction{
    private final Trader trader;
    private final int year;
    private final int value;
    public Transaction(Trader trader, int year, int value){
        this.trader = trader;
        this.year = year;
        this.value = value;
    }
    public Trader getTrader(){
        return this.trader;
    }
    public int getYear(){
        return this.year;
    }
    public int getValue(){
        return this.value;
    }
    public String toString(){
        return "{" + this.trader + ", " +
               "year: "+this.year+", " +
               "value:" + this.value +"}";
    }
}
```

Listing 5.1 Finds all transactions in 2011 and sort by value (small to high)

```
List<Transaction> tr2011 =
    transactions.stream()
                .filter(transaction -> transaction.getYear() == 2011)
                .sorted(comparing(Transaction::getValue))
                .collect(toList());
```

Passes a predicate to filter to select transactions in year 2011

Collects all the elements of the resulting Stream into a List

Sorts them by using the value of the transaction

Listing 5.2 What are all the unique cities where the traders work?

```
List<String> cities =
    transactions.stream()
                .map(transaction -> transaction.getTrader().getCity())
                .distinct()
                .collect(toList());
```

Extracts the city from each trader associated with the transaction

Selects only unique cities

You haven't seen this yet, but you could also drop `distinct()` and use `toSet()` instead, which would convert the stream into a set. You'll learn more about it in chapter 6.

```
Set<String> cities =
    transactions.stream()
                .map(transaction -> transaction.getTrader().getCity())
                .collect(toSet());
```

Listing 5.3 Finds all traders from Cambridge and sort them by name

```
List<Trader> traders =
    transactions.stream()
                .map(Transaction::getTrader)
                .filter(trader -> trader.getCity().equals("Cambridge"))
                .distinct()
                .sorted(comparing(Trader::getName))
                .collect(toList());
```

Extracts all traders from the transactions

Selects only the traders from Cambridge

Removes any duplicates

Sorts the resulting stream of traders by their names

Listing 5.4 Returns a string of all traders' names sorted alphabetically

```
String traderStr =
    transactions.stream()
                .map(transaction -> transaction.getTrader().getName())
                .distinct()
                .sorted()
                .reduce("", (n1, n2) -> n1 + n2);
```

Extracts all the names of the traders as a Stream of Strings

Removes duplicate names

Sorts the names alphabetically

Combines the names one by one to form a String that concatenates all the names

Note that this solution is inefficient (all Strings are repeatedly concatenated, which creates a new `String` object at each iteration). In the next chapter, you'll see a more efficient solution that uses `joining()` as follows (which internally makes use of a `StringBuilder`):

```
String traderStr =
    transactions.stream()
                .map(transaction -> transaction.getTrader().getName())
                .distinct()
                .sorted()
                .collect(joining());
```

Listing 5.5 Are any traders based in Milan?

```
boolean milanBased =
    transactions.stream()
                .anyMatch(transaction -> transaction.getTrader()
                                            .getCity()
                                            .equals("Milan"));
```

Pass a predicate to anyMatch to check if there's a trader from Milan.

Listing 5.6 Prints all transactions' values from the traders living in Cambridge

```
transactions.stream()
            .filter(t -> "Cambridge".equals(t.getTrader().getCity()))
            .map(Transaction::getValue)
            .forEach(System.out::println);
```

Selects the transactions where the traders live in Cambridge

Prints each value

Extracts the values of these trades

Listing 5.7 What's the highest value of all the transactions?

```
Optional<Integer> highestValue =
    transactions.stream()
                .map(Transaction::getValue)
                .reduce(Integer::max);
```

Extracts the value of each transaction

Calculates the max of the resulting stream

Listing 5.8 Finds the transaction with the smallest value

```
Optional<Transaction> smallestTransaction =
  transactions.stream()
              .reduce((t1, t2) ->
                      t1.getValue() < t2.getValue() ? t1 : t2);
```

Finds the smallest transaction by repeatedly comparing the values of each transaction

You can do better. A stream supports the methods `min` and `max` that take a `Comparator` as argument to specify which key to compare with when calculating the minimum or maximum:

```
Optional<Transaction> smallestTransaction =
  transactions.stream()
              .min(comparing(Transaction::getValue));
```

5.7 Numeric streams

- You can calculate the number of calories in the menu as follows:

```
int calories = menu.stream()
                   .map(Dish::getCalories)
                   .reduce(0, Integer::sum); // boxing cost
```

- Streams API also supplies *primitive stream specializations* that support specialized methods to work with streams of numbers.
- Java 8 introduces three primitive specialized stream interfaces :
  - IntStream
  - DoubleStream
  - LongStream

Primitive stream specializations

- Mapping to a Numeric Stream
  - mapToInt, mapToDouble, and mapToLong.
  - you can use `mapToInt` as follows to calculate the sum of calories in the `menu`:

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

Returns a Stream<Dish>

Returns an IntStream

Here, the method `mapToInt` extracts all the calories from each dish (represented as an `Integer`) and returns an `IntStream` as the result (rather than a `Stream<Integer>`).

Converting Back to a Stream of Objects

- To convert from a primitive stream to a general stream (each int will be boxed to an Integer) you can use the method boxed, as follows:

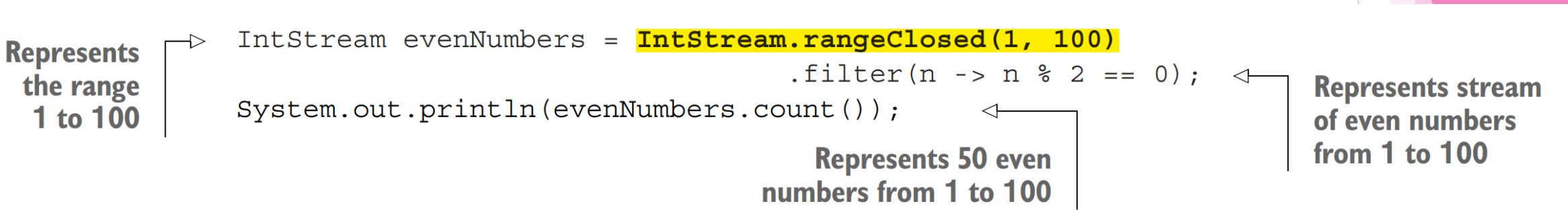
```
IntStream intStream = menu.stream().mapToInt(Dish::getCalories);
Stream<Integer> stream = intStream.boxed();
```

Converts a Stream to a numeric stream

Converts the numeric stream to a Stream

Numeric Range

- Java 8 introduces two static methods available on `IntStream` and `LongStream` to help generate such ranges: `range` and `rangeClosed`.



Default Values: `OptionalInt`

- Three primitive stream specializations:

  `OptionalInt`, `OptionalDouble`, and `OptionalLong`.

- You can find the maximal element of an `IntStream` by calling the `max` method, which returns an `OptionalInt`:

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

You can now process the `OptionalInt` explicitly to define a default value if there's no maximum:

```
int max = maxCalories.orElse(1);
```

Provides an explicit default maximum if there's no value

Putting numerical streams into practice: Pythagorean triples

- Create a stream of Pythagorean triples.
  - For example, (3, 4, 5) is a valid Pythagorean triple because 3 * 3 + 4 * 4 = 5 * 5 or 9 + 16 = 25. There are an infinite number of such triples. For example, (5, 12, 13), (6, 8, 10), and (7, 24, 25) are all valid Pythagorean triples.
  - For example, `new int[]{3, 4, 5}` to represent the tuple (3, 4, 5).
  - Test whether the square root of a * a + b * b is a whole number. This is expressed in Java as `Math.sqrt(a*a + b*b) % 1 == 0`.
  - Uses this idea in a filter operation (you'll see how to use this later to form valid code):

```
filter(b -> Math.sqrt(a*a + b*b) % 1 == 0)    // Assume a give value a
```

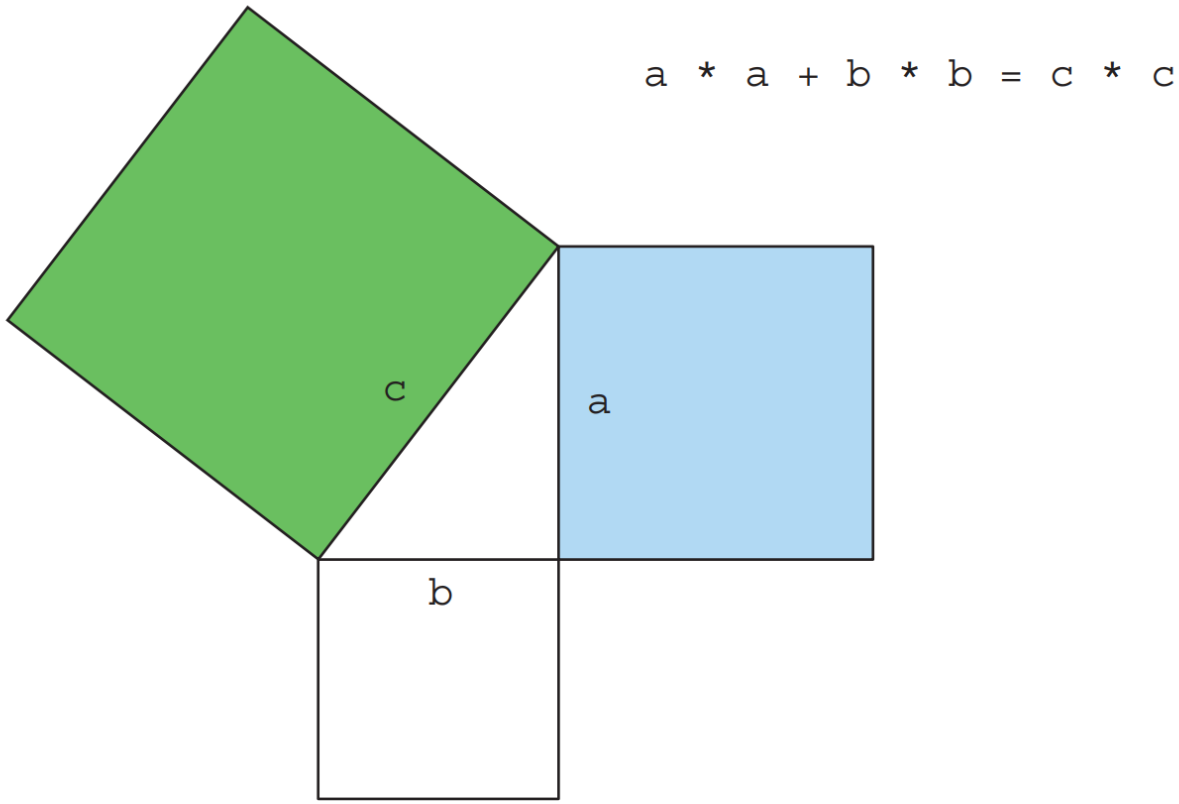


Figure 5.9 The Pythagorean theorem

Generating Tuples

```
stream.filter(b -> Math.sqrt(a*a + b*b) % 1 == 0)
        .map(b -> new int[]{a, b, (int) Math.sqrt(a * a + b * b)});
```

Generating B Values

- You can use the following code to provide numeric values for `b`, here 1 to 100:

```
IntStream.rangeClosed(1, 100)
.filter(b -> Math.sqrt(a*a + b*b) % 1 == 0)
.boxed()
.map(b -> new int[]{a, b, (int) Math.sqrt(a * a + b * b)});
```

- You can rewrite this using the method `mapToObj` of an `IntStream`, which returns an object-valued stream:

```
IntStream.rangeClosed(1, 100)
.filter(b -> Math.sqrt(a*a + b*b) % 1 == 0)
.mapToObj(b -> new int[]{a, b, (int) Math.sqrt(a * a + b * b)});
```

Generating A Values

- The final solution is as follows:

```
Stream<int[]> pythagoreanTriples =
        IntStream.rangeClosed(1, 100).boxed()
        .flatMap(a -> IntStream.rangeClosed(a, 100)
        .filter(b -> Math.sqrt(a*a + b*b) % 1 == 0)
        .mapToObj(b ->
              new int[]{a, b, (int)Math.sqrt(a * a + b * b)})
        );
```

RUNNING THE CODE

You can now run your solution and select explicitly how many triples you'd like to return from the generated stream using the `limit` operation that you saw earlier:

```
pythagoreanTriples.limit(5)
                  .forEach(t ->
                     System.out.println(t[0] + ", " + t[1] + ", " + t[2]));
```

This will print

```
3, 4, 5
5, 12, 13
6, 8, 10
7, 24, 25
8, 15, 17
```

CAN YOU DO BETTER?

The current solution isn't optimal because you calculate the square root twice. One possible way to make your code more compact is to generate all triples of the form `(a*a, b*b, a*a+b*b)` and then filter the ones that match your criteria:

```
Stream<double[]> pythagoreanTriples2 =
  IntStream.rangeClosed(1, 100).boxed()
           .flatMap(a ->
              IntStream.rangeClosed(a, 100)
                       .mapToObj(
                          b -> new double[]{a, b, Math.sqrt(a*a + b*b)})
                       .filter(t -> t[2] % 1 == 0));
```

Produces triples

The third element of the tuple must be a whole number.

5.8 Building streams

- You can create a stream from a sequence of values, from an array, from a file, and even from a generative function to create infinite streams!
- Streams from values

```
Stream<String> stream = Stream.of("Modern ", "Java ", "In ", "Action");
stream.map(String::toUpperCase).forEach(System.out::println);
```

You can get an empty stream using the `empty` method as follows:

```
Stream<String> emptyStream = Stream.empty();
```

Streams from nullable

- The method System.getProperty returns null if there is no property with the given key.

```
String homeValue = System.getProperty("home");
Stream<String> homeValueStream
    = homeValue == null ? Stream.empty() : Stream.of(value);
```

Using `Stream.ofNullable` you can rewrite this code more simply:

```
Stream<String> homeValueStream
    = Stream.ofNullable(System.getProperty("home"));
```

This pattern can be particularly handy in conjunction with `flatMap` and a stream of values that may include nullable objects:

```
Stream<String> values =
   Stream.of("config", "home", "user")
         .flatMap(key -> Stream.ofNullable(System.getProperty(key)));
```

- Streams from arrays

```
int[] numbers = {2, 3, 5, 7, 11, 13};
int sum = Arrays.stream(numbers).sum();
```

The sum is 41.

- Streams from files
  - Java's NIO API (non-blocking I/O)
  - Many static methods in java.nio.file.Files return a stream.
  - Files.lines: returns a stream of lines as strings from a given file.

```
long uniqueWords = 0;
try(Stream<String> lines =
          Files.lines(Paths.get("data.txt"), Charset.defaultCharset())){
uniqueWords = lines.flatMap(line -> Arrays.stream(line.split(" ")))
                   .distinct()
                   .count();
}
catch(IOException e){

}
```

Streams are AutoCloseable so there's no need for try-finally

Generates a stream of words

Removes duplicates

Counts the number of unique words

Deals with the exception if one occurs when opening the file

Streams from functions: creating infinite streams!

ITERATE

- The iterate method takes an initial value, here 0, and a lambda (of type UnaryOperator<T>) to apply successively on each new value produced.

```
Stream.iterate(0, n -> n + 2)
      .limit(10)
      .forEach(System.out::println);
```

- In Java 9, the iterate method was enhanced with support for a predicate.

```
IntStream.iterate(0, n -> n < 100, n -> n + 4)
         .forEach(System.out::println);

IntStream.iterate(0, n -> n + 4)
         .filter(n -> n < 100)
         .forEach(System.out::println);

IntStream.iterate(0, n -> n + 4)
         .takeWhile(n -> n < 100)
         .forEach(System.out::println);
```

GENERATE

- The method generate lets you produce an infinite stream of values computed on demand. It takes a lambda of type Supplier<T> to provide new values.

```
Stream.generate(Math::random)
        .limit(5)
        .forEach(System.out::println);
```

This code will generate a stream of five random double numbers from 0 to 1. For example, one run gives the following:

```
0.941081029410612 9
0.658627075563459 2
0.959285911726687 3
0.137433966594870 06
0.394277603765124 1
```

- Generate an infinite stream of ones (twos):

```
IntStream ones = IntStream.generate(() -> 1);

IntStream twos = IntStream.generate(new IntSupplier(){
            public int getAsInt(){
                return 2;
            }
        });
```

- Create an IntSupplier that will return the next Fibonacci element when it's called:

```
IntSupplier fib = new IntSupplier(){
    private int previous = 0;
    private int current = 1;
    public int getAsInt(){
        int oldPrevious = this.previous;
        int nextValue = this.previous + this.current;
        this.previous = this.current;
        this.current = nextValue;
        return oldPrevious;
    }
};
IntStream.generate(fib).limit(10).forEach(System.out::println);
```