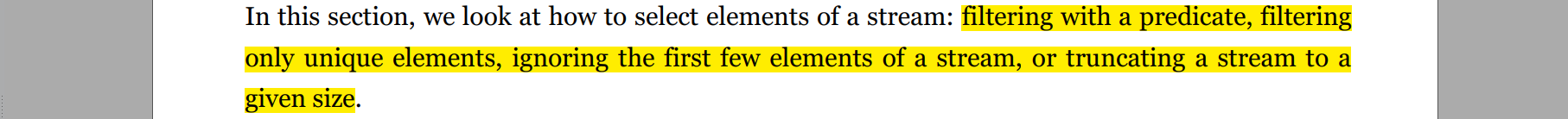
1. 
2. A close-up of a person's face

   Description automatically generated
3. Last chapter, streams let us move from external iteration to internal iteration.
4. We can use Stream API (Internal Iteration), which supports filter and collect operations, to manage the iteration over a collection.   
   All we need is to pass the filtering behavior as an argument to the filter method.
5. Due to internal iteration, Stream API can work our several optimizations behind the scenes and Stream can also run the code in parallel.  
   Using external iteration, this is not possible because we are committed to a single threaded step-by-step sequential iteration.
6. A close up of text

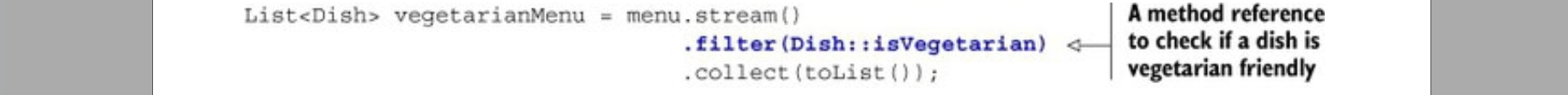
   Description automatically generated

## 5.1 Filtering and slicing

1. 

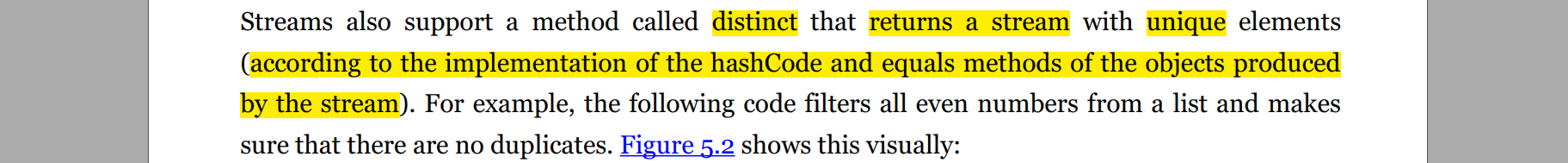
### 5.1.1 Filtering with a predicate

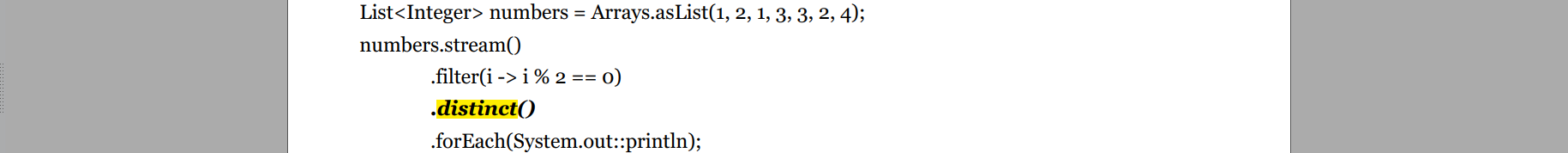
1. Streams Interface has a method Stream<P\_OUT> filter(Predicate<? super P\_OUT> predicate) which
   1. Takes one predicate as an argument and
   2. Returns Steam of elements matching the predicate.

  
A diagram of a diagram

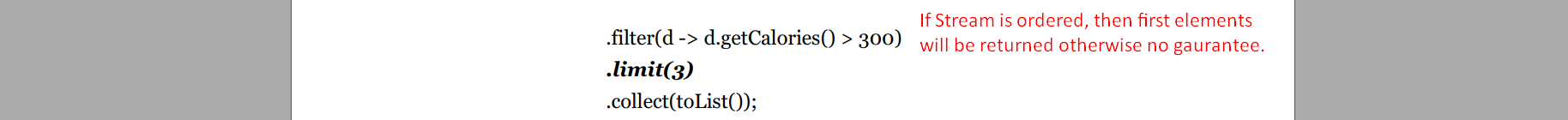
Description automatically generated

### 5.1.2 Filtering Unique Elements

1. 
2.  A diagram of a diagram

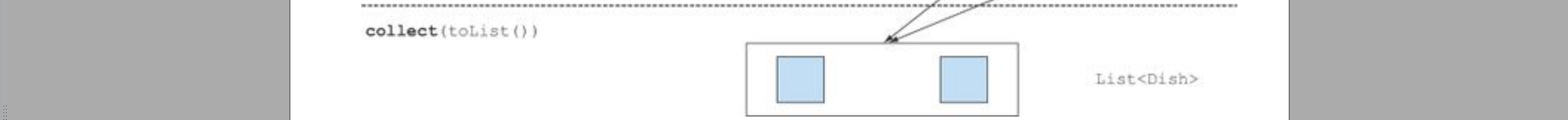
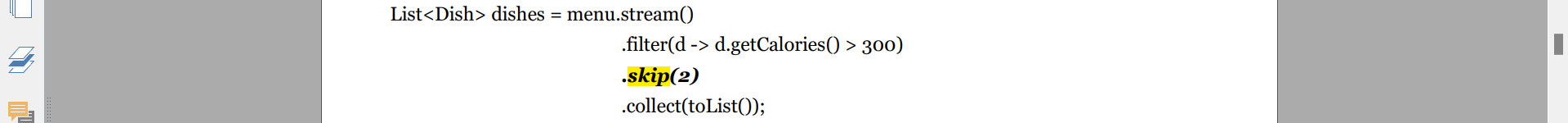
   Description automatically generated 

### 5.1.3 Truncating a Stream

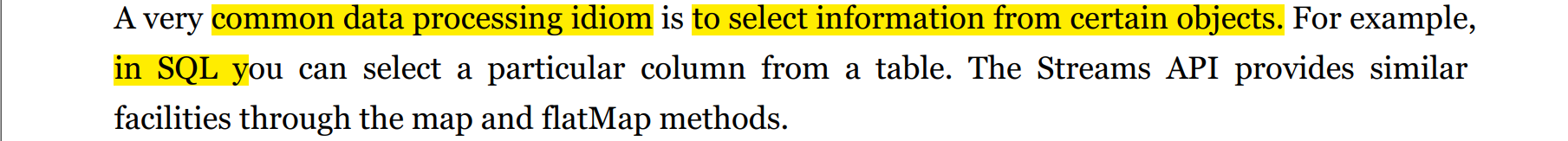
1. Returns another stream of elements having max the passed limit. If **Ordered Stream** will return 1st elements and **unordered stream** will not gaurantee.

### 5.1.4 Skipping Elements

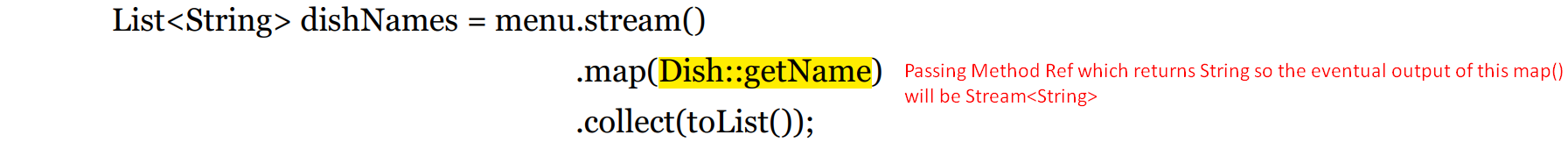
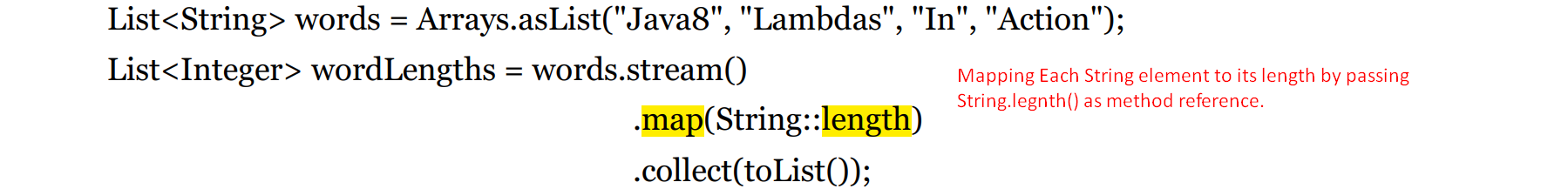
1. Stream.skip(n): Skips the 1st n element.
2. If fewer elements than n, then an empty stream is returned.
3. Stream.limit(n) and Stream.skip(n) are complimentary.
4. A diagram of a stream

   Description automatically generated 

## 5.2 Mapping

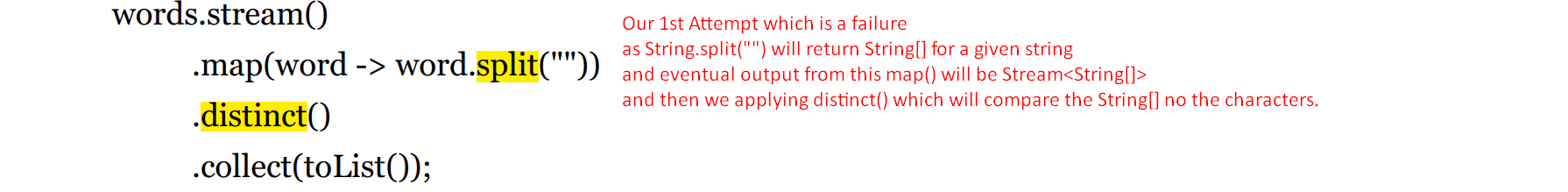
1. 

### 5.2.1 Applying a Function to each element of a Stream

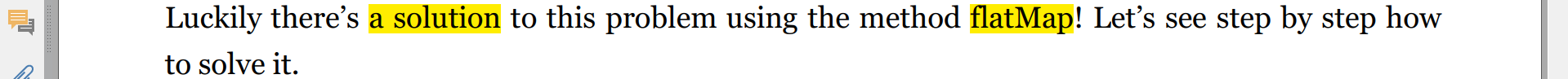
1. Streams support the method ma().
2. It takes a function as an argument which is applied to each element and mapping it to a new element.
3. 
4. 

### 5.2.2 Flattening Streams

1. A yellow text with black text

   Description automatically generated
2. 
3. A screenshot of a computer

   Description automatically generatedA diagram of a diagram

   Description automatically generated
4.   
   Jatin: <https://www.youtube.com/watch?v=CwvlS3ViGFQ&t=719s>  
   A screenshot of a computer

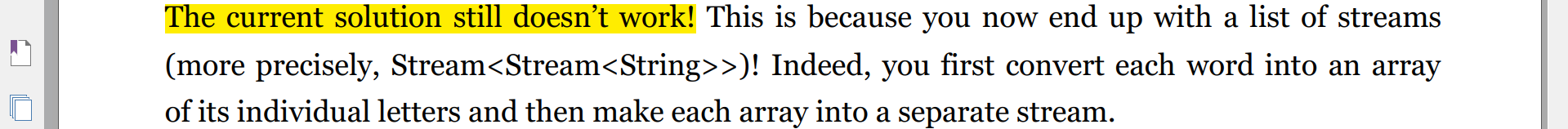
   Description automatically generated  
   A screen shot of a computer

   Description automatically generatedA screenshot of a computer

   Description automatically generated

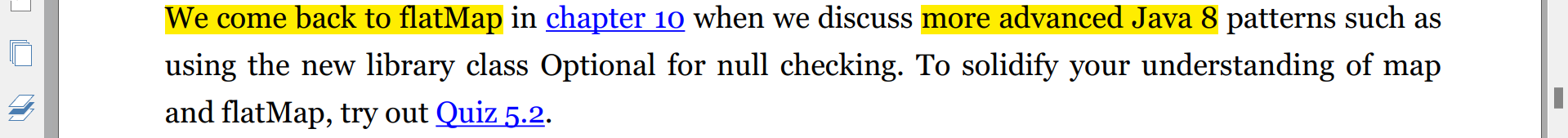
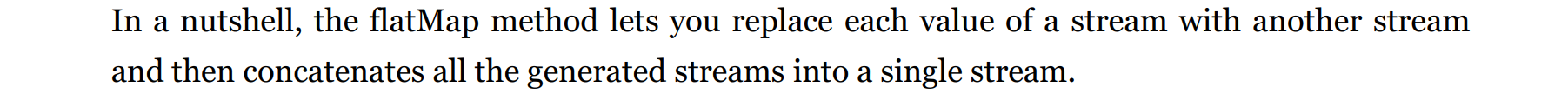
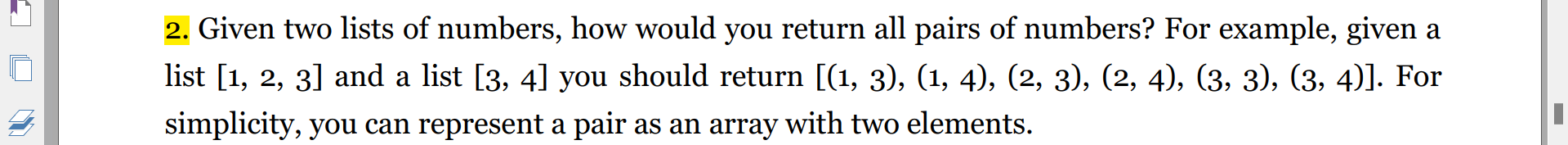
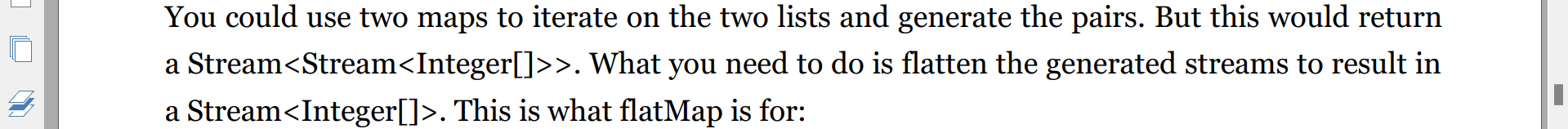
#### Attempting Using map and Arrays.stream

1. A close-up of a white paper

   Description automatically generated
2. 

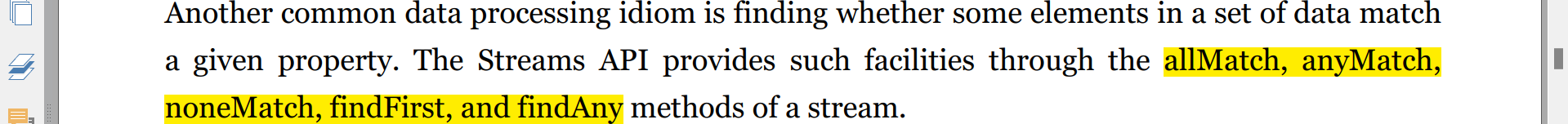
#### Using flatMap

1. A red line with black text

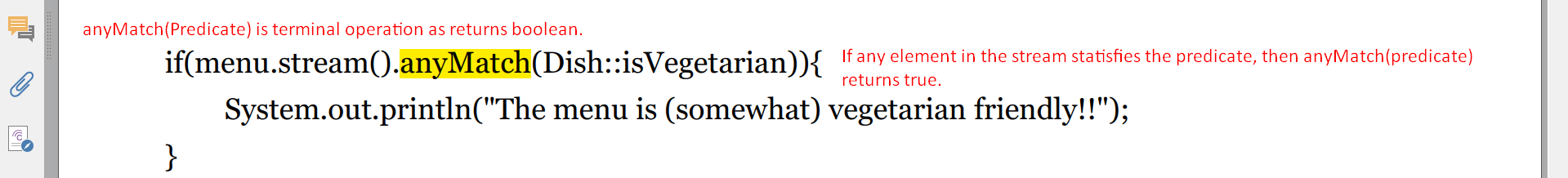
   Description automatically generated
2. 
3.   
   A white background with black text

   Description automatically generated

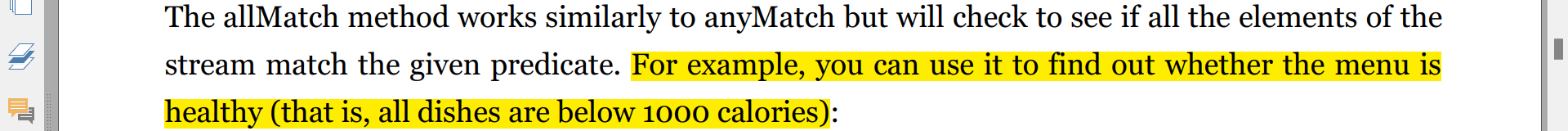
## 5.3 Finding and Matching

1. 

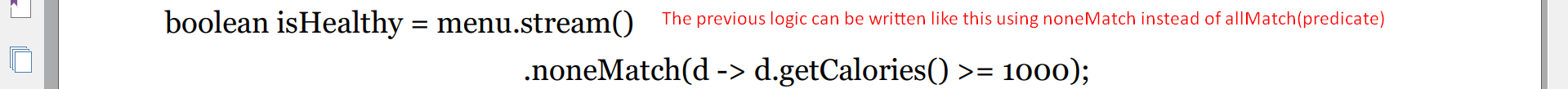
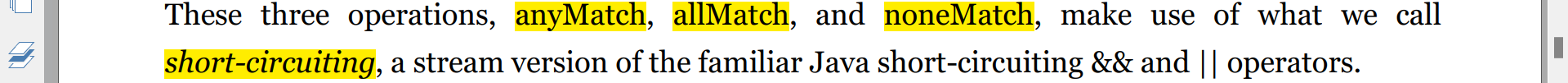
### 5.3.1 Checking to see if a predicate matches at least one element.

1. Following scenario to check if a given menu has any veg dish.

### 5.3.2 Checking to see if a predicate matches all elements.

1. 
2. 

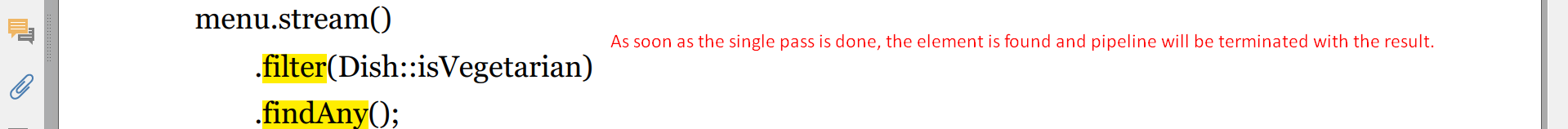
#### noneMatch

1. 
2. 

#### Short-Circuiting Evaluation

1. Some operations don’t need to evaluate the whole stream to produce a result.
2. **For Example**: A long Boolean expression with AND. As soon as any sub-expression is equal to false, the whole expression can be evaluated as false.  
   That is what short-circuiting refers to.
3. In relation to streams, certain operations such as **allMatch, anyMatch, noneMatch, findFirst, limit, and findAny** don’t need to process the whole stream to produce a result.
4. This behavior is very important when limit is applied on streams of infinite size because limit can turn an infinite stream into a stream of fine size.

### 5.3.3 Finding an Element

1. The findAny() returns an arbitrary element of a given stream.
2. The stream pipeline will be optimized behind the scenes to perform a single pass and finish as soon as a result is found by using **short-circuiting.**
3. From the given dishes, find the first veg dish.  
   

#### Optional in Nutshell

1. **java.util.Optional** is a **container** class to represent the existence or absence of a value.
2. In the previous code, findAny doesn’t find any element and instead of returning null which is error-prone, it returns Optional which we will discuss in Chapter 10 in detail.
3. Optional supports some methods which force us to explicitly check for the presence or absence of a value.
4. A close up of text

   Description automatically generated
5. A screen shot of a computer

   Description automatically generated

### 5.3.4 Finding the First Element

1. Some streams have **encounter order** that specifies the order in which items logically appear in the stream (for example a stream generated from a List or from a sorted sequence of data).
2. For such streams you may