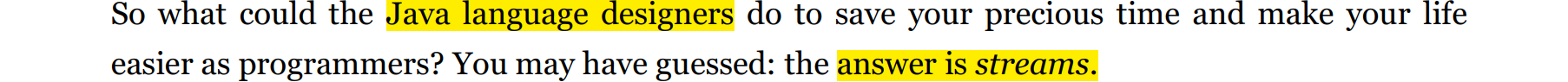
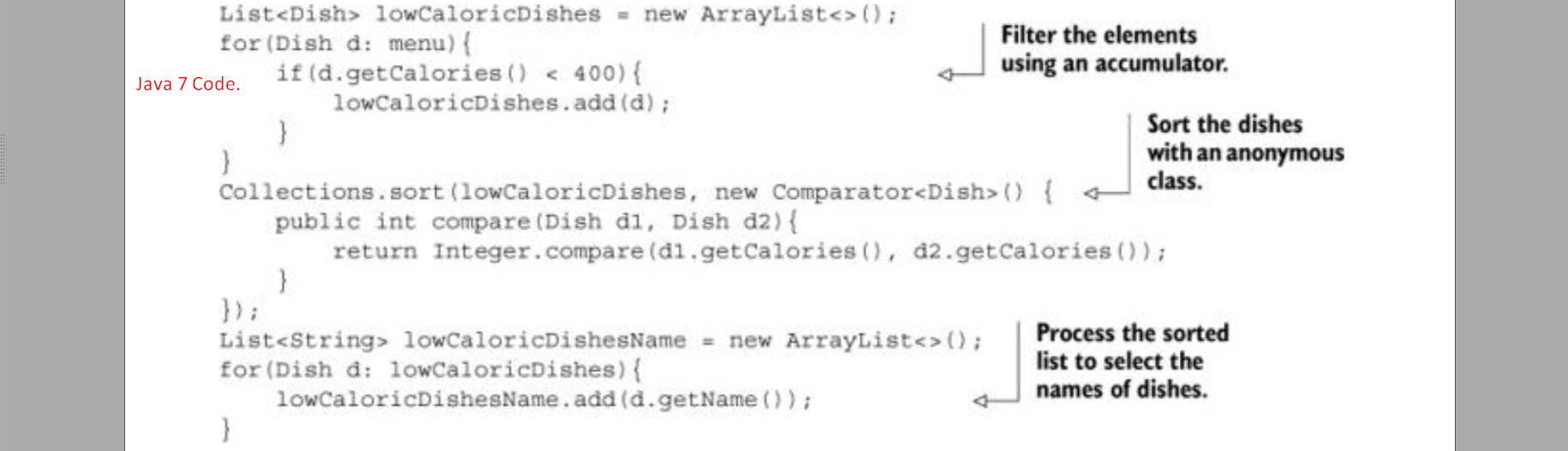
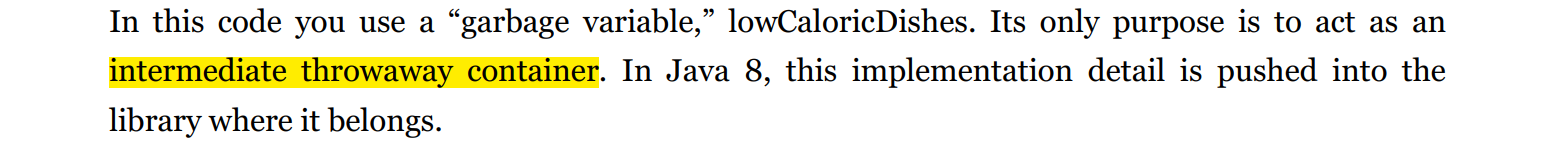


1. 
2. 
3. Collection is very fundamental in any programming language.   
   Many problems involve collection and its processing.
4. **But despite collections being necessary for almost any Java App, manipulating collections is far from perfect**:
   1. Much business logic entails (involves) Database-Like Operations such as **grouping a list of dishes** by category (Vegetarian Dishes) or finding the most expensive dish.
   2. Many times, we reimplement same logic again and again.
   3. Most DB lets us specify such operations **declaratively** such as   
      SELECT name FROM dishes WHERE calorie < 400. (No need to use **iterator and accumulator by us**).  
      The basic idea is that we do not need to implement this but already done.  
      Why can’t we do similar with Collection?
   4. How to process large collection of elements where we need to process in parallel and leverage multicore architectures to gain performance.   
      But writing **parallel code** is complicated in comparison to working with iterators.
5. 

# 4.1 What are Streams?

1. Streams are an update to the Java API that lets you manipulate collections of data in a **declarative way (we express a query rather than code)**.  
   Think of it as fancy iterator over a collection of data.
2. In addition, streams can be processed in parallel without us to write any additional code for it.  
   Chapter 07: We will see how streams and **parallelization** **work**.
3. Let’s understand the following problem in Java 7 and Java 8 (Do not worry about Java 8. We will talk about it).  
   **Given**: A collection of Dishes.   
   **Task**: Collect name of those Dishes in sorted order whose calories are less than 400
4. **Java 7 Solution**:  
     
     
     
   **Java 8 Solution:**A screenshot of a computer code

   Description automatically generated
5. A white background with black text

   Description automatically generated  
   
6. Chapter 07: Will cover all the points like.
   1. What exactly happens when we call **parallelStream**()?
   2. How many threads are being used?
   3. What are the performance benefits?
7. For now, we can see that the new approach offers several immediate benefits from a Software Engineering point of view.
   1. The code is written in **declarative Way**.
8. A text on a page

   Description automatically generated
9. A diagram of a computer program

   Description automatically generated  
   A yellow line on a white background

   Description automatically generated  
   A close up of a text

   Description automatically generated  
   A white background with black text

   Description automatically generated

# 4.2 Getting Started with Streams

1. Collections in Java 8 supports a new Stream method that returns a stream.
2. We will also see we will be getting streams other ways like.
   1. Generating stream elements from a numeric range or
   2. From an I/O resource.
3. **Stream**: A sequence of elements from a source that supports data processing operations.

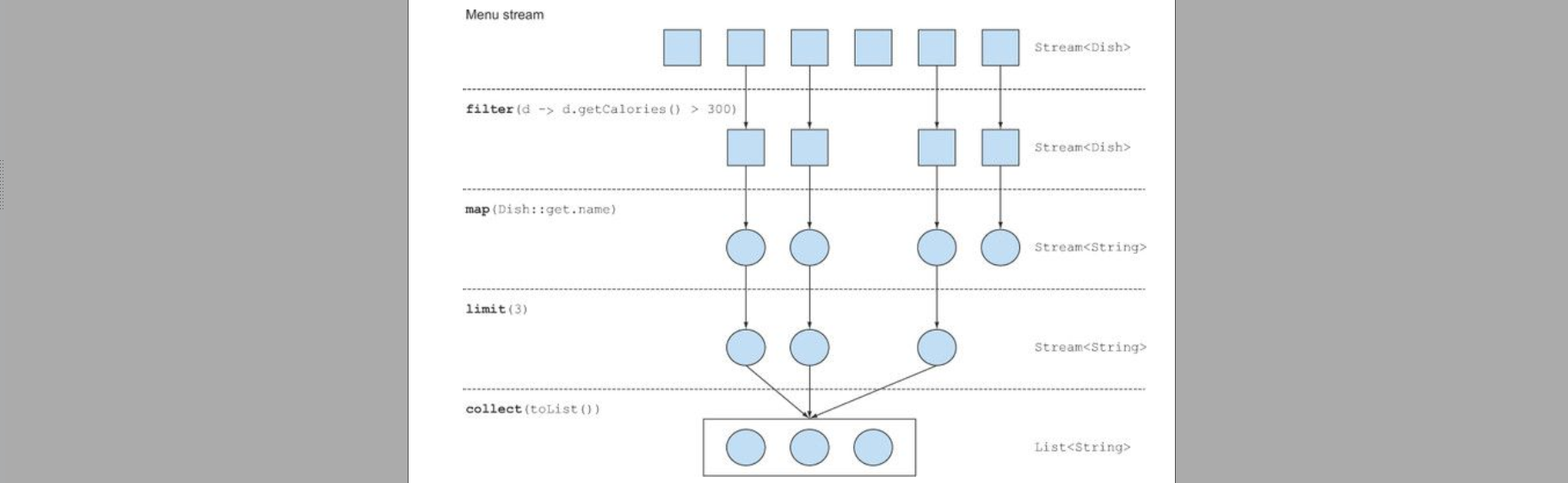
**Let us break down this definition step by step :  
Sequence of Elements:**

|  |  |
| --- | --- |
| **Collection** | **Stream** |
| Provides Data Structure for a sequenced set of elements. As DS, so mostly about storing and accessing elements in DS. | Provides way of expressing computation |

**Source:**

Streams consume from data providing source such as collection, array, I/O resources.  
Stream generated from an ordered collection preserves the ordering.

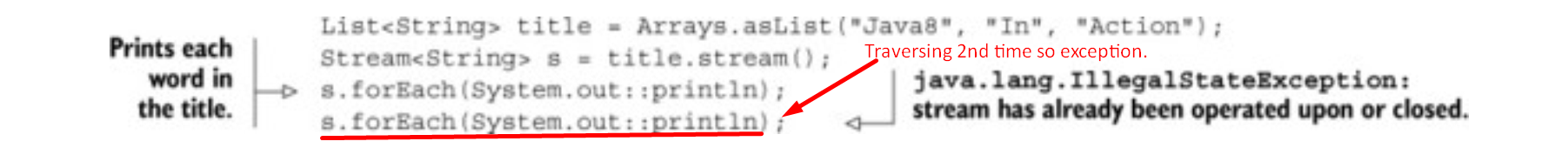
1. **In addition, Stream operations have two important characteristics:**
   1. **Pipelining:** Many Stream operations return a stream to form a larger stream pipeline which offer certain optimizations such as **laziness** and **short-circuiting**.
   2. **Internal Iteration:** Collection does iteration explicitly using an iterator whereas Stream operations do the iteration behind the scene.
2. A screenshot of a computer code

   Description automatically generated
3. **See our** requirement was to find names of three high-calorie dishes.  
   We didn’t implement filtering (filter), extracting (map), and truncating (limit) functionalities.   
   These functionalities are available through stream library.  
   As a result, the Stream API has more flexibility to decide how to optimize this pipeline.  
   For example, filtering, extracting, and truncating steps could be merged into a single pass and stop as soon as three dishes are found. We will show this in next chapter.
4. **Let’s** now see the conceptual different b/w Collection API and Stream API.

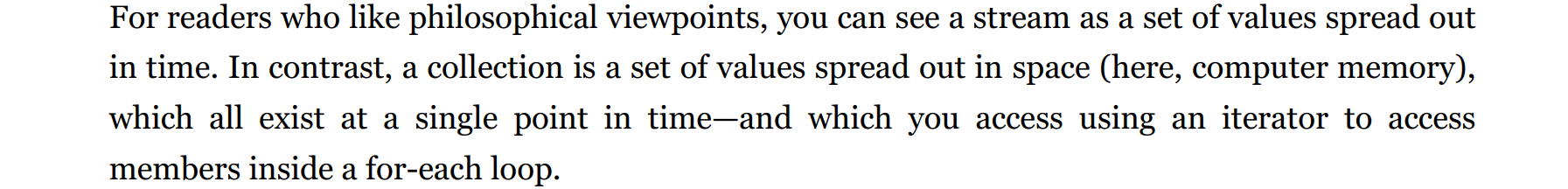
# 4.3 Streams vs. Collections

1. Nothing Special, just analogy b/w Collection and Stream using a movie on a DVD and streaming it through net.

# 4.3.1 Traversable Only Once

1. Similarly to Iterators, a stream can be traversed only once. Then the stream is said to be consumed.
2. We can get stream again if the data source is repeatable not I/O Channel.
3. 

## Streams and Collections philosophically

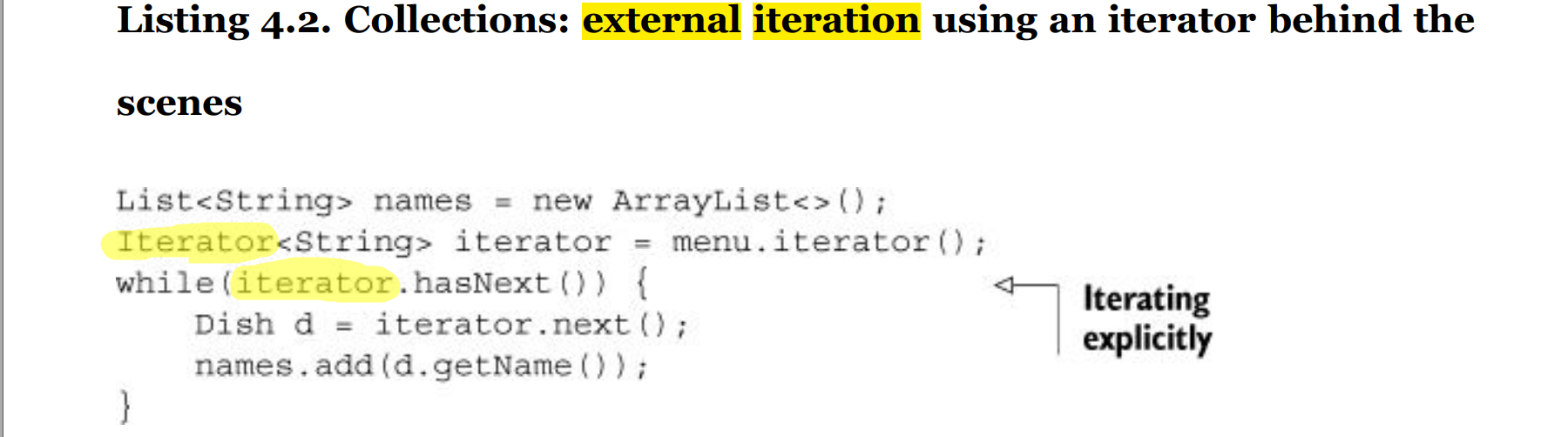
1. 

### 4.3.2 External vs. Internal Iteration

1. Using the Collection interface requires iteration to be done by the user (for example, using for-each).

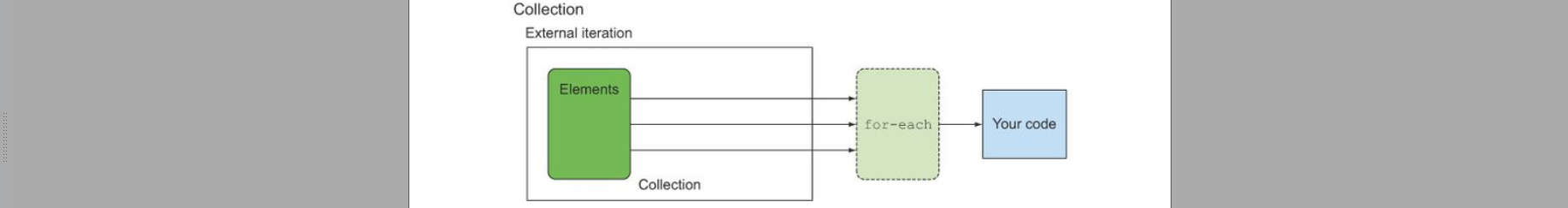
It is called **external iteration**.

1. On the other hand, Streams library uses **internal iteration -** it does the iterations for us and stores the resulting stream value somewhere; We merely provide function saying what is to be done.
2. A screenshot of a computer

   Description automatically generated  
   Using Iterator behind the scene, for-each loop does the iteration which is hidden from us.
3. 
4. 
5. A screenshot of a computer screen

   Description automatically generated
6. It would be far better if we could just tell Sofia, “Put all the toys that are on the floor inside the box”.
7. A yellow and black text

   Description automatically generated
8. Similarly, using an internal iteration, the processing of items could be transparently done in parallel  
   or in different order that is more optimized.   
   Internal iteration in Stream processing can choose a data representation and implementation of parallelism to match our hardward.
9. A diagram of a diagram

   Description automatically generated
10. 

## 4.4 Stream Operations

1. **java.util.stream.Stream** defines many operations which can be classified into two categories.
2. A screenshot of a computer code

   Description automatically generated
3. **Intermediate Operation**: The stream operations that can be connected.
4. **Terminal Operations**: Operations that close a stream.
5. A diagram of a diagram

   Description automatically generated

### 4.4.1 Intermediate Operations

1. Intermediate operations such as filter(), map() return another stream as the **return type**.  
   This allows the operations to be connected to form a query.
2. Intermediate operations are **lazy** meaning they are not executed until **terminal operation** is not invoked on this pipeline.
3. To understand what is happening inside the stream pipeline, put print statement in lambda.   
   A screenshot of a computer program

   Description automatically generated  
     
   A white background with black and white clouds

   Description automatically generated
4. A close up of text

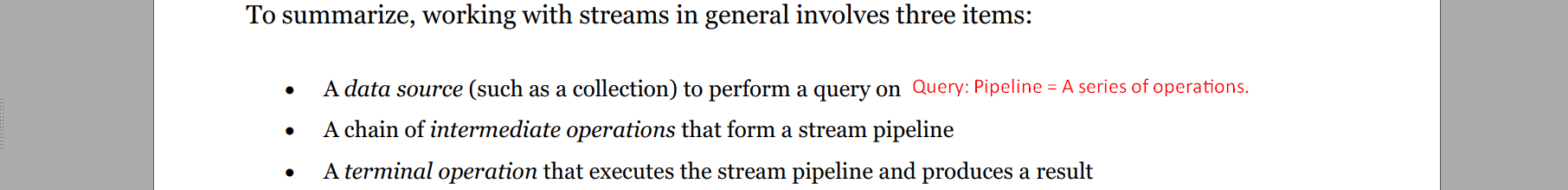
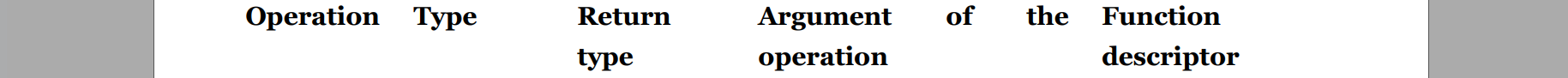
   Description automatically generated

### 4.4.2 Terminal Operations

1. Terminal operation returns a result which is non-stream value such as Collection, Wrapper type or even void.   
   The return type of forEach() is void and it applies the lambda to each element from the given stream.
2. A screenshot of a computer

   Description automatically generated

### 4.4.3 Working with Streams

1. 
2. The idea behind a stream pipeline is similar to builder pattern.  
   **Builder Pattern**: A chain of calls to set up a configuration.  
   **Stream**: A chain of intermediate operations and terminal operations.
3.    
   A group of black and white text

   Description automatically generated A close up of text

   Description automatically generated