Part 2 Functional-Style Data Processing with Streams

4 Introducing streams

This chapter covers

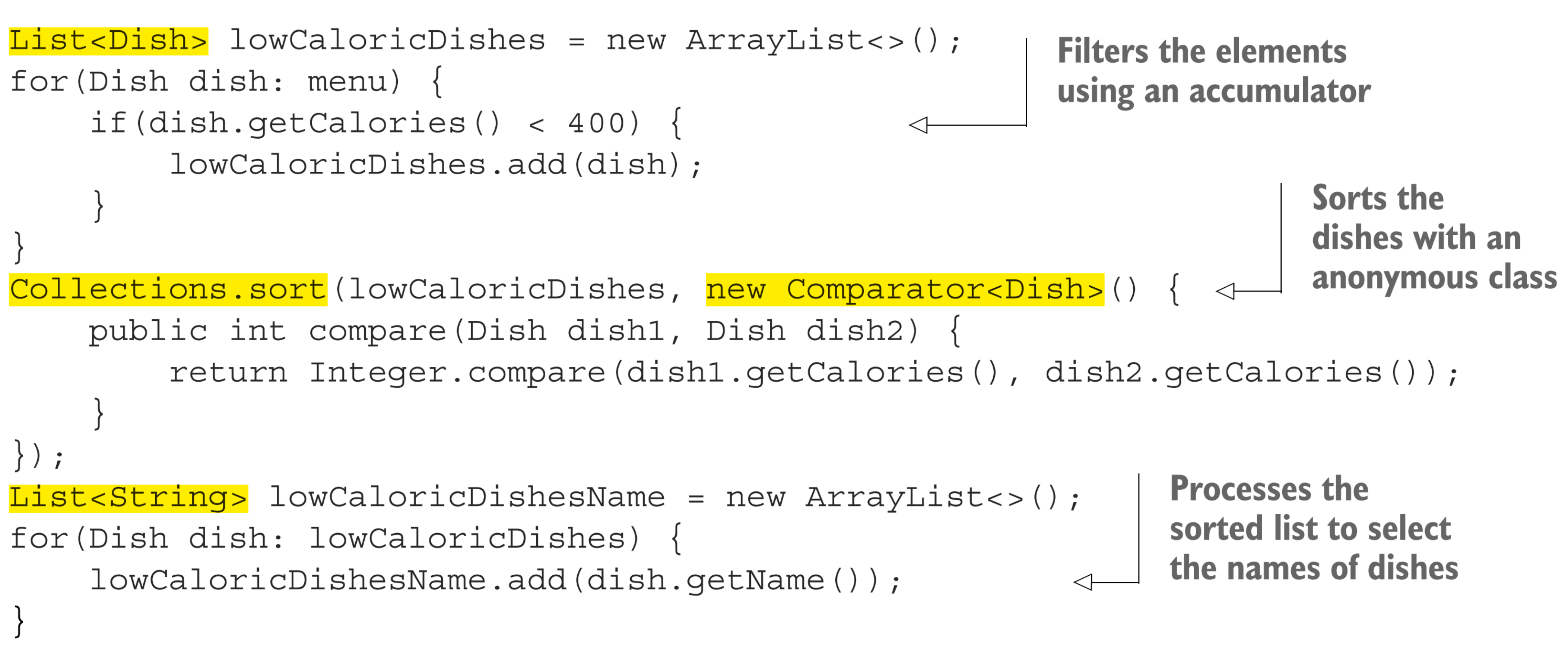
- What is a stream?
- Collections versus streams
- Internal versus external iteration
- Intermediate versus terminal operations

- Nearly every Java application *makes* and *processes* collections.
- Collections are fundamental to many programming tasks: they let you group and process data.
- Much business logic entails database-like operations
  - *grouping* a list of dishes by category (for example, all vegetarian dishes)
  - *finding* the most expensive dish.
  - SQL query like SELECT name FROM dishes WHERE calorie < 400.
  - You write what you wants instead of how to explicitly implement such a query.
- How would you process a large collection of elements?
  - To gain performance you'd need to process it in parallel and use multicore architectures.
- The answer is *streams*.

4.1 What are streams?

- *Streams*
  - an update to the Java API that let you manipulate collections of data in a declarative way.
  - fancy iterators over a collection of data
  - Streams can be processed in parallel *transparently*.
- The benefits of using Java 8 streams:
  - The code is written in a declarative way. You specify what you want to achieve.
  - You chain together several building-block operations to express a complicated data-processing pipeline.

Before (Java 7):



```
List<Dish> lowCaloricDishes = new ArrayList<>();
for(Dish dish: menu) {
    if(dish.getCalories() < 400) {
        lowCaloricDishes.add(dish);
    }
}
Collections.sort(lowCaloricDishes, new Comparator<Dish>() {
    public int compare(Dish dish1, Dish dish2) {
        return Integer.compare(dish1.getCalories(), dish2.getCalories());
    }
});
List<String> lowCaloricDishesName = new ArrayList<>();
for(Dish dish: lowCaloricDishes) {
    lowCaloricDishesName.add(dish.getName());
}
```

Filters the elements using an accumulator

Sorts the dishes with an anonymous class

Processes the sorted list to select the names of dishes

After (Java 8):

```
import static java.util.Comparator.comparing;
import static java.util.stream.Collectors.toList;
List<String> lowCaloricDishesName =
                menu.stream()
                    .filter(d -> d.getCalories() < 400)
                    .sorted(comparing(Dish::getCalories))
                    .map(Dish::getName)
                    .collect(toList());
```

Selects dishes that are below 400 calories

Sorts them by calories

Extracts the names of these dishes

Stores all the names in a List

To exploit a multicore architecture and execute this code in parallel, you need only to change `stream()` to `parallelStream()`:

```
List<String> lowCaloricDishesName =
               menu.parallelStream()
                   .filter(d -> d.getCalories() < 400)
                   .sorted(comparing(Dishes::getCalories))
                   .map(Dish::getName)
                   .collect(toList());
```

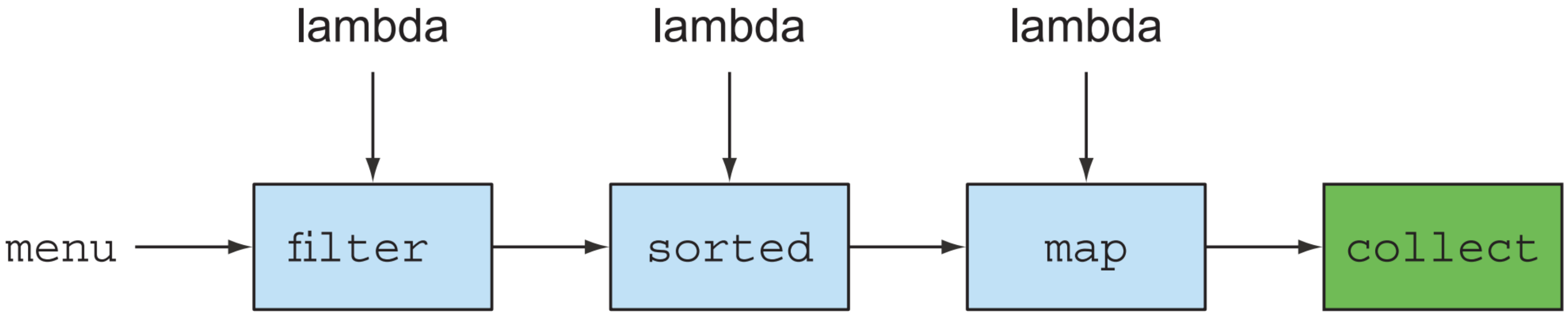


Figure 4.1 Chaining stream operations forming a stream pipeline

- The new Streams API is expressive. For example,

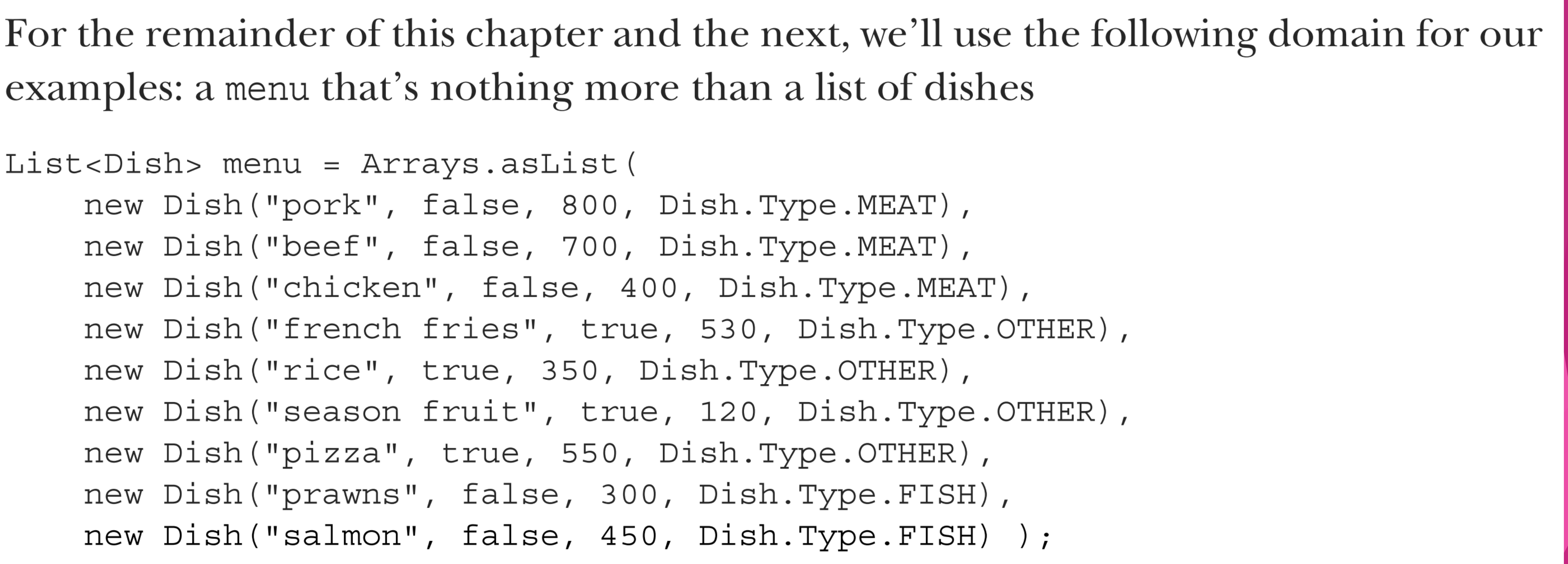
```
Map<Dish.Type, List<Dish>> dishesByType =
    menu.stream().collect(groupingBy(Dish::getType));
```

- The following result:

```
{FISH=[prawns, salmon],
 OTHER=[french fries, rice, season fruit, pizza],
 MEAT=[pork, beef, chicken]}
```

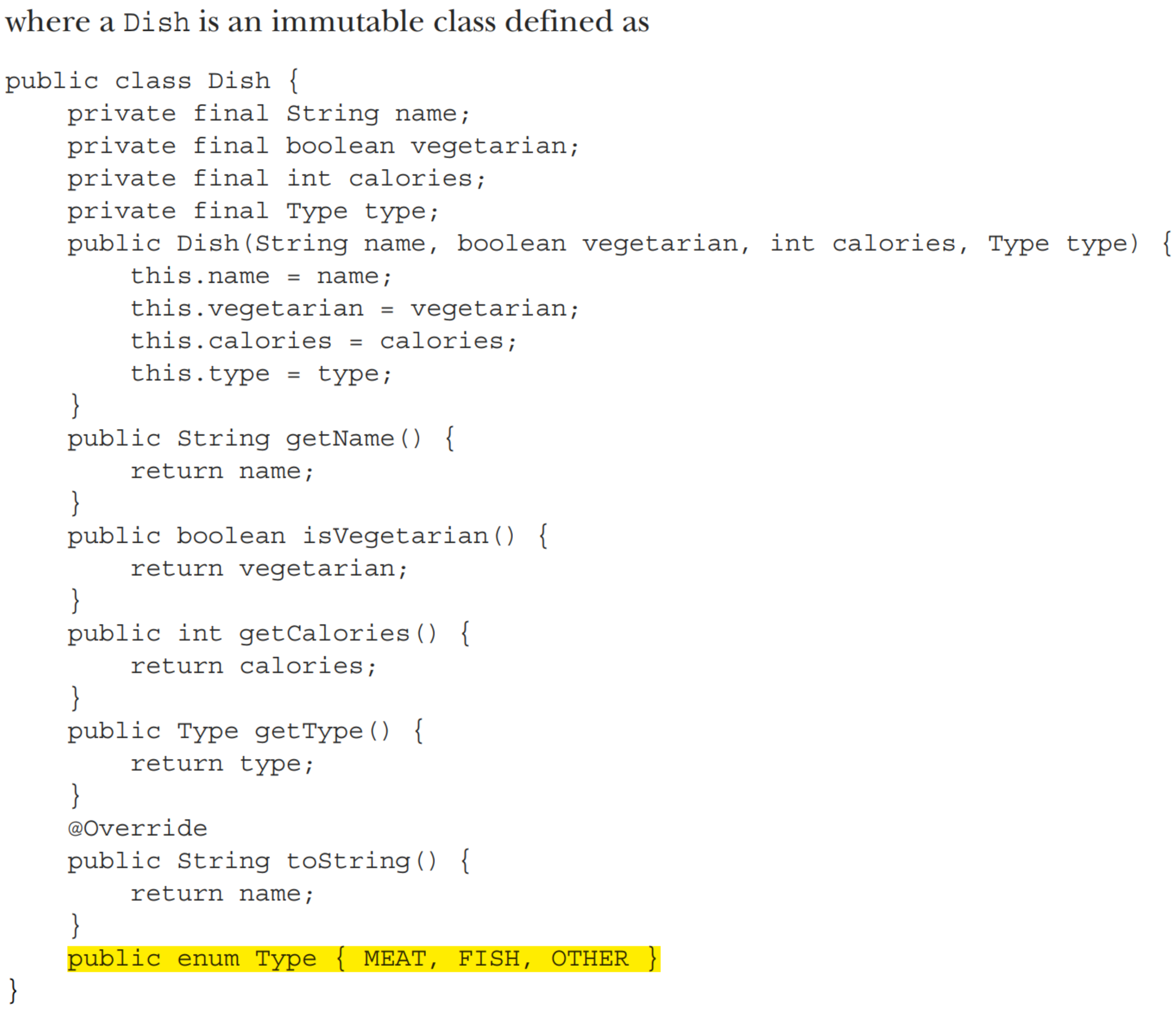
What are streams? (Cont.)

- To summarize, the Streams API in Java 8 lets you write code that’s
  - *Declarative* — More concise and readable
  - *Composable* — Greater flexibility
  - *Parallelizable* — Better performance



For the remainder of this chapter and the next, we'll use the following domain for our examples: a `menu` that's nothing more than a list of dishes

```
List<Dish> menu = Arrays.asList(
    new Dish("pork", false, 800, Dish.Type.MEAT),
    new Dish("beef", false, 700, Dish.Type.MEAT),
    new Dish("chicken", false, 400, Dish.Type.MEAT),
    new Dish("french fries", true, 530, Dish.Type.OTHER),
    new Dish("rice", true, 350, Dish.Type.OTHER),
    new Dish("season fruit", true, 120, Dish.Type.OTHER),
    new Dish("pizza", true, 550, Dish.Type.OTHER),
    new Dish("prawns", false, 300, Dish.Type.FISH),
    new Dish("salmon", false, 450, Dish.Type.FISH) );
```



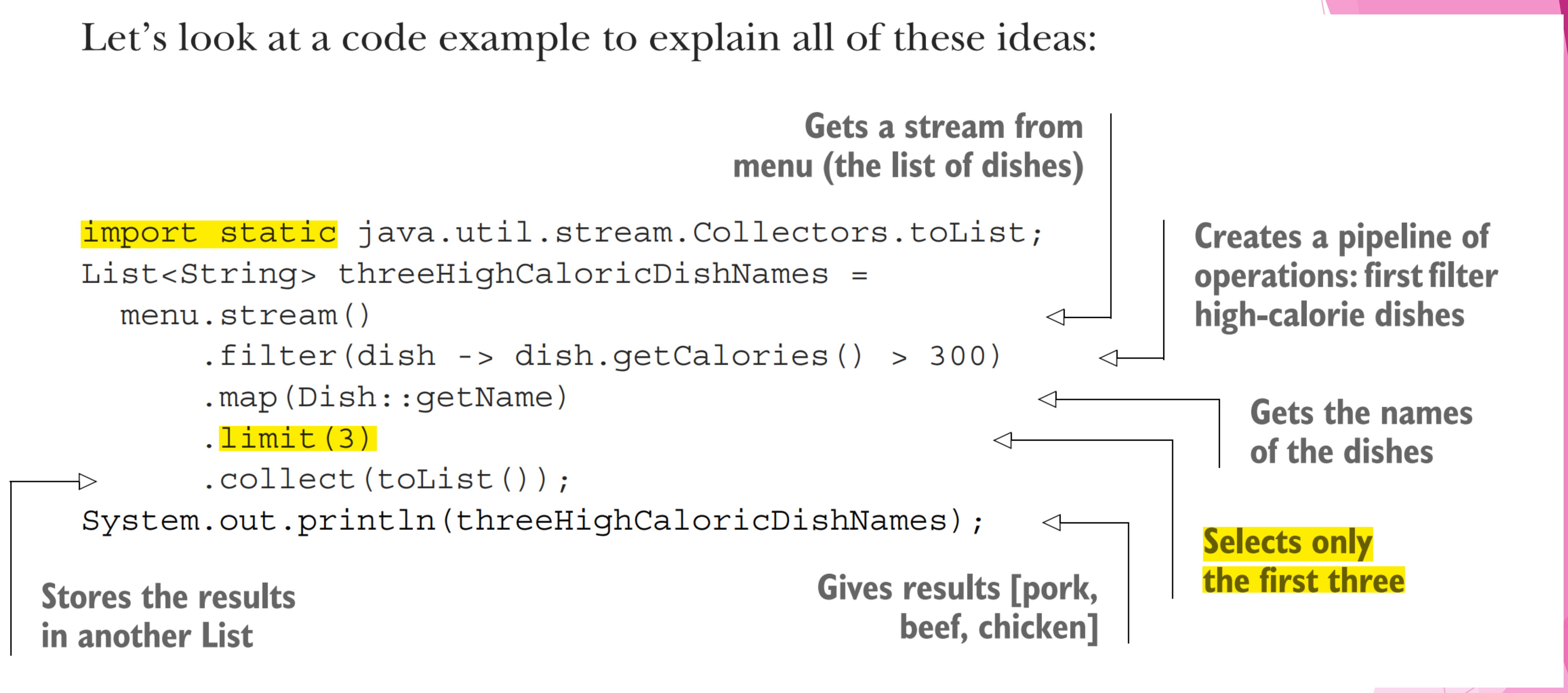
where a Dish is an immutable class defined as

```
public class 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) {
        this.name = name;
        this.vegetarian = vegetarian;
        this.calories = calories;
        this.type = type;
    }
    public String getName() {
        return name;
    }
    public boolean isVegetarian() {
        return vegetarian;
    }
    public int getCalories() {
        return calories;
    }
    public Type getType() {
        return type;
    }
    @Override
    public String toString() {
        return name;
    }
    public enum Type { MEAT, FISH, OTHER }
}
```

4.2 Getting started with streams

- Collections in Java 8 support a new stream method that returns a stream (the interface defined in java.util.stream.Stream).
- What exactly is a *stream*? A short definition is “a sequence of elements from a source that supports data-processing operations.”
  - *Sequence of elements*
    - Streams are about expressing computations (filter, sorted, and map).
    - Collections are about data.
  - *Sources*
    - Collections, arrays, or I/O resources.
  - *Data-processing operations*
    - Streams supports database-like operations and common operations from functional programming languages to manipulate data such as filter, map, reduce, find, match, sort, and so on.

- Two important characteristics of stream operations:
  - *Pipelining* — Many stream operations return a stream themselves, allowing operations to be chained to form a larger pipeline.
  - *Internal iteration* — In contrast to collections, which are iterated explicitly using an iterator, stream operations do the iteration behind the scenes for you.



Let’s look at a code example to explain all of these ideas:

```
import static java.util.stream.Collectors.toList;
List<String> threeHighCaloricDishNames =
  menu.stream()
      .filter(dish -> dish.getCalories() > 300)
      .map(Dish::getName)
      .limit(3)
      .collect(toList());
System.out.println(threeHighCaloricDishNames);
```

Gets a stream from menu (the list of dishes)

Creates a pipeline of operations: first filter high-calorie dishes

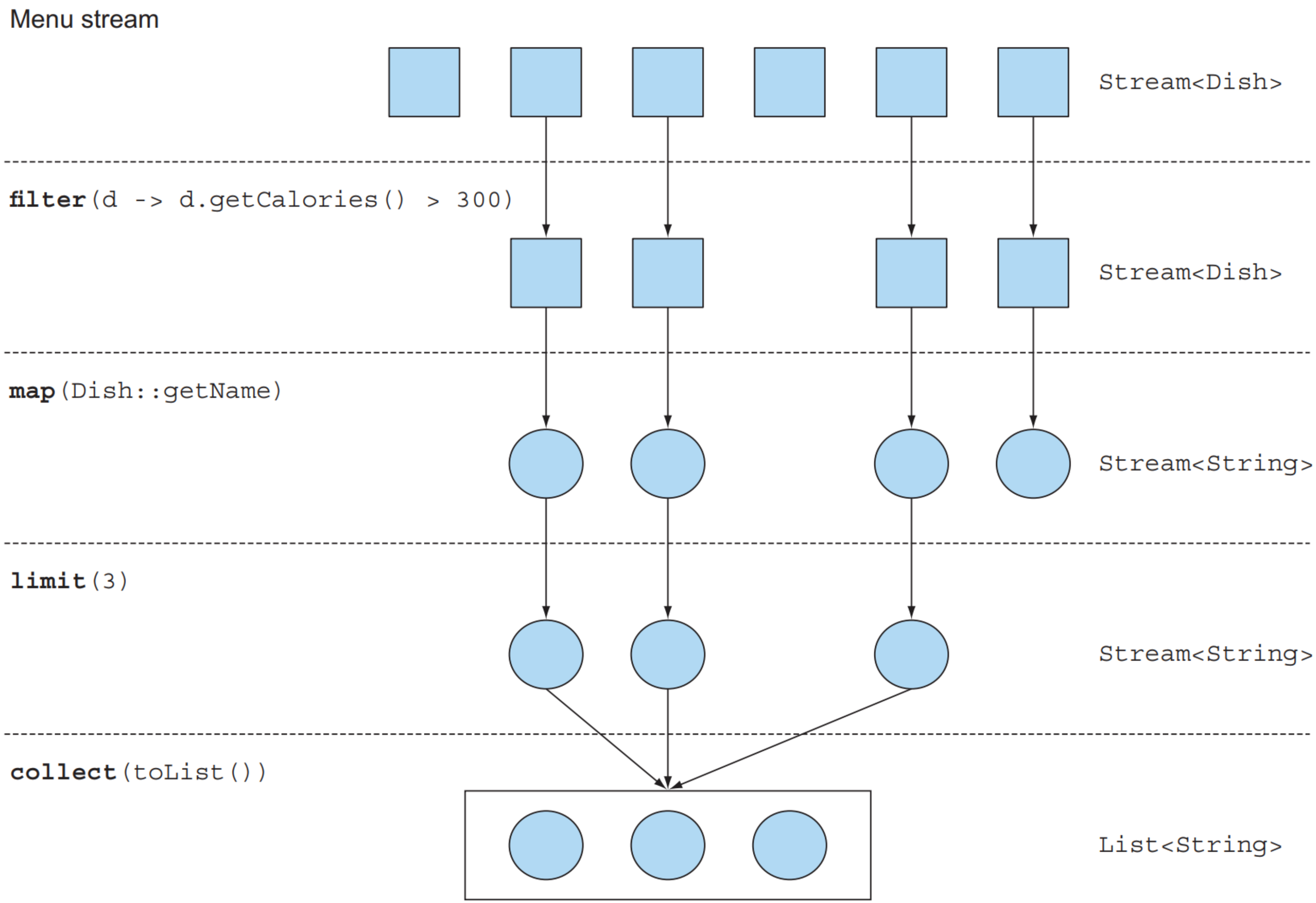
Gets the names of the dishes

Selects only the first three

Gives results [pork, beef, chicken]

Stores the results in another List

“Find names of three high-calorie dishes.”



Menu stream

Stream<Dish>

filter(d -> d.getCalories() > 300)

Stream<Dish>

map(Dish::getName)

Stream<String>

limit(3)

Stream<String>

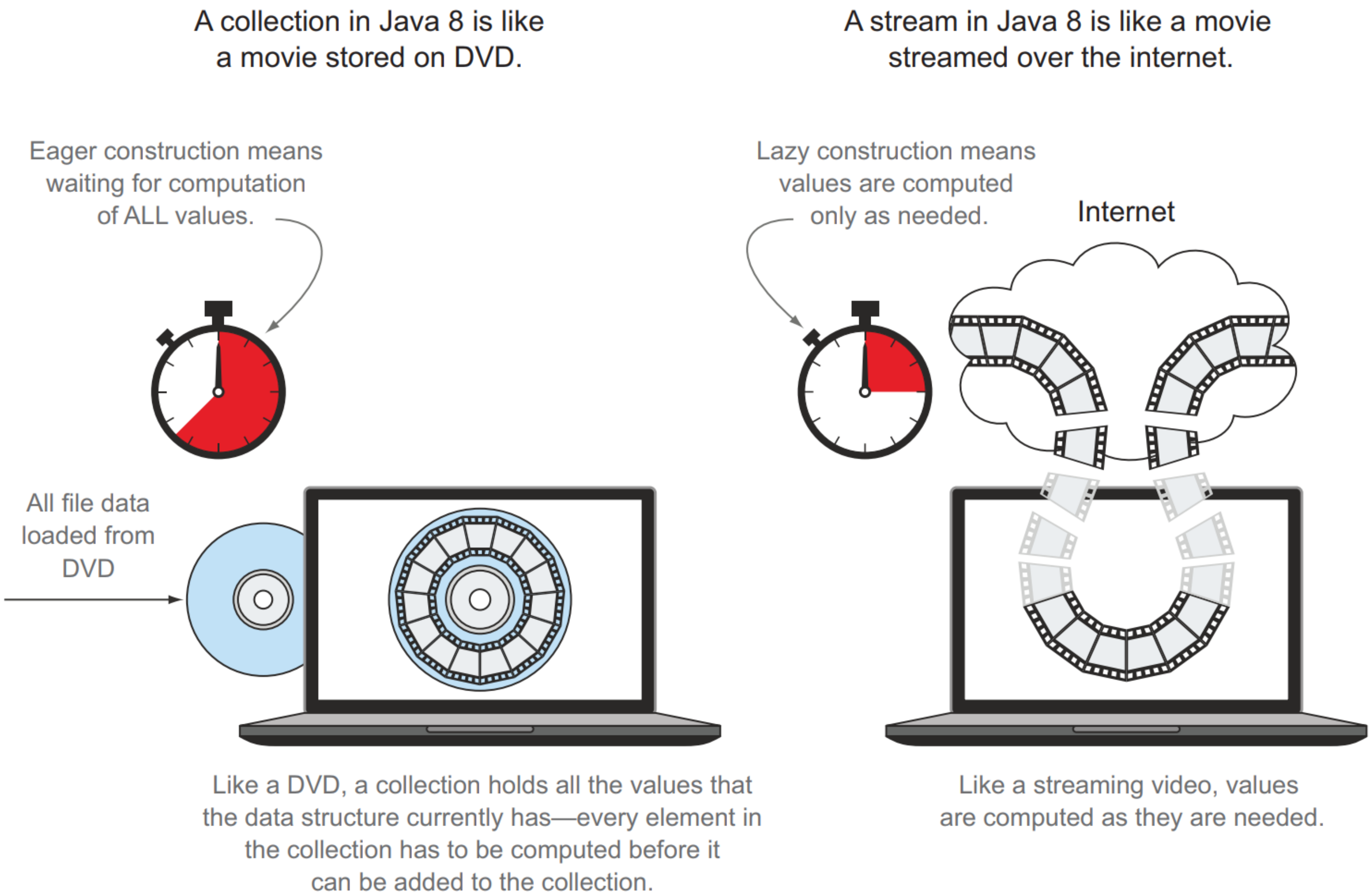
collect(toList())

List<String>

Figure 4.2 Filtering a menu using a stream to find out three high-calorie dish names

4.3 Streams vs. collections

- Java Collections vs. Java Streams
  - Streams provide interfaces to data structures representing a sequence set of values of the element type.
  - A collection is an in-memory data structure that holds all the values the data structure currently has – every element in the collection has to be computed before it can be added to the collection. (You can add things to and removed from them, the collection.)
  - A stream is a conceptually fixed data structure (you can't add or remove elements from it) whose elements are *computed on demand*.
  - A user will extract only the values they require from a stream and these elements are produced – invisibly to the user – only *as* and when required. This is a form of a **producer-consumer** relationship.



A collection in Java 8 is like a movie stored on DVD.

A stream in Java 8 is like a movie streamed over the internet.

Eager construction means waiting for computation of ALL values.

Lazy construction means values are computed only as needed.

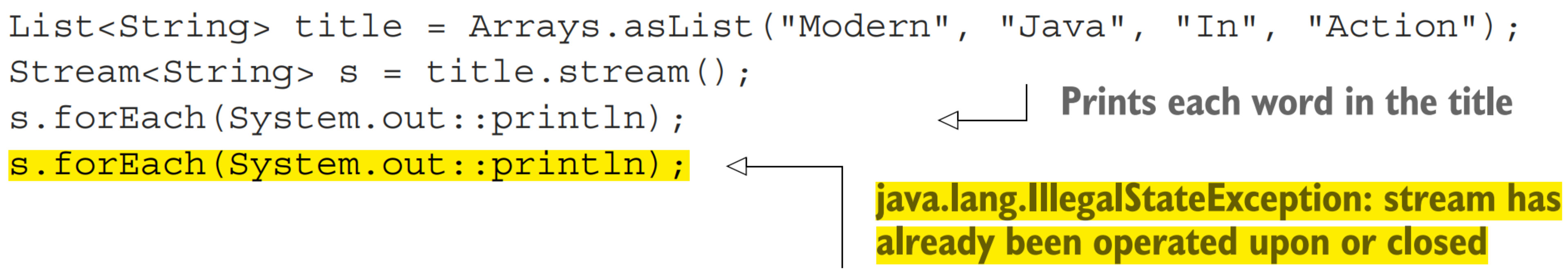
All file data loaded from DVD

Like a DVD, a collection holds all the values that the data structure currently has—every element in the collection has to be computed before it can be added to the collection.

Like a streaming video, values are computed as they are needed.

Figure 4.3 Streams versus collections

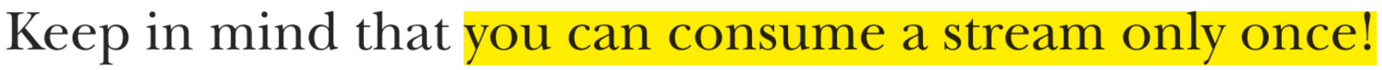
- Traversable only once
  - Note that, similarly to iterators, a stream can be traversed only once. After that a stream is said to be consumed.



```
List<String> title = Arrays.asList("Modern", "Java", "In", "Action");
Stream<String> s = title.stream();
s.forEach(System.out::println);
s.forEach(System.out::println);
```

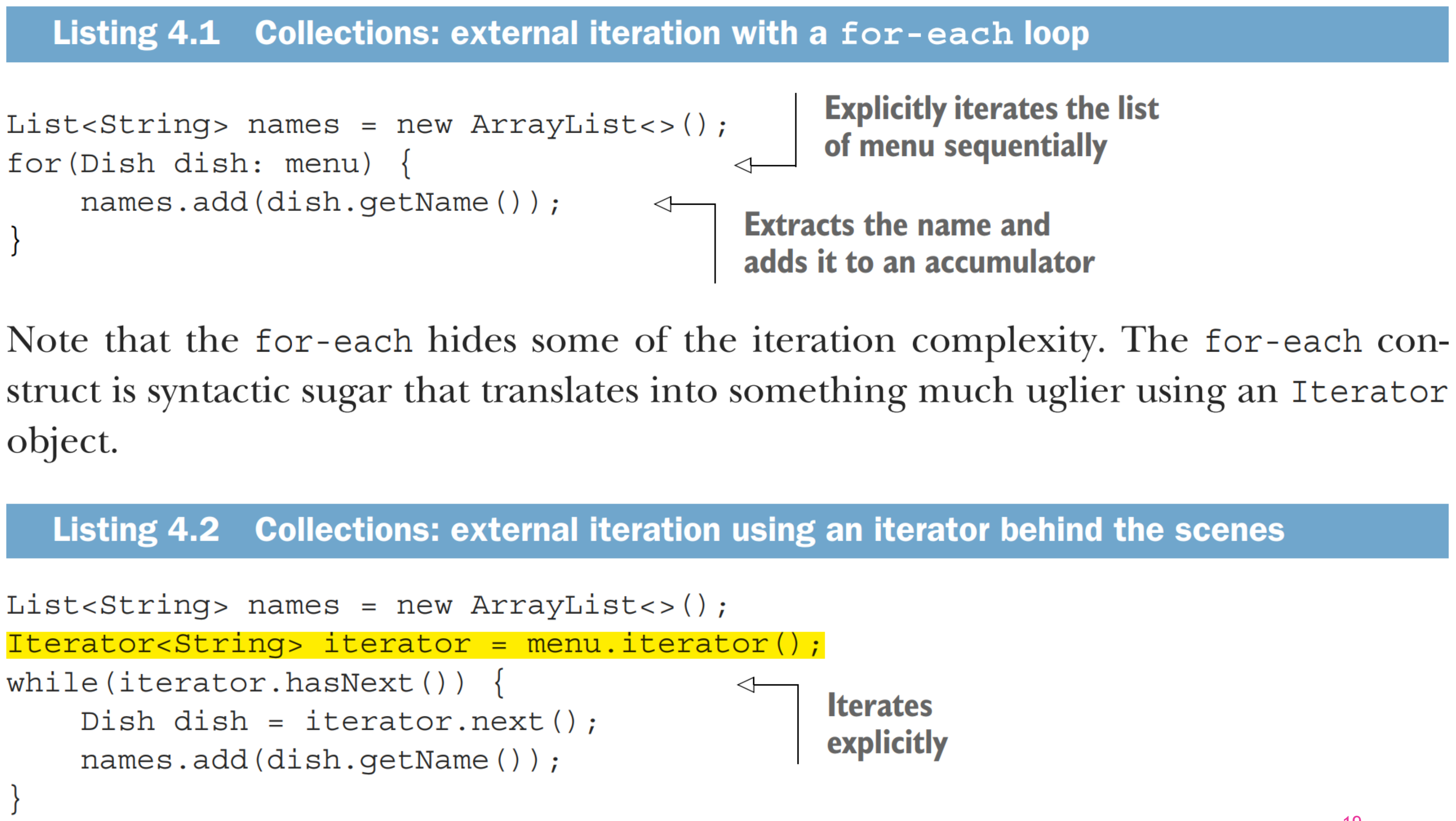
Prints each word in the title

java.lang.IllegalStateException: stream has already been operated upon or closed



Keep in mind that you can consume a stream only once!

External vs. Internal Iteration



Listing 4.1 Collections: external iteration with a `for-each` loop

```
List<String> names = new ArrayList<>();
for(Dish dish: menu) {
    names.add(dish.getName());
}
```

Explicitly iterates the list of menu sequentially

Extracts the name and adds it to an accumulator

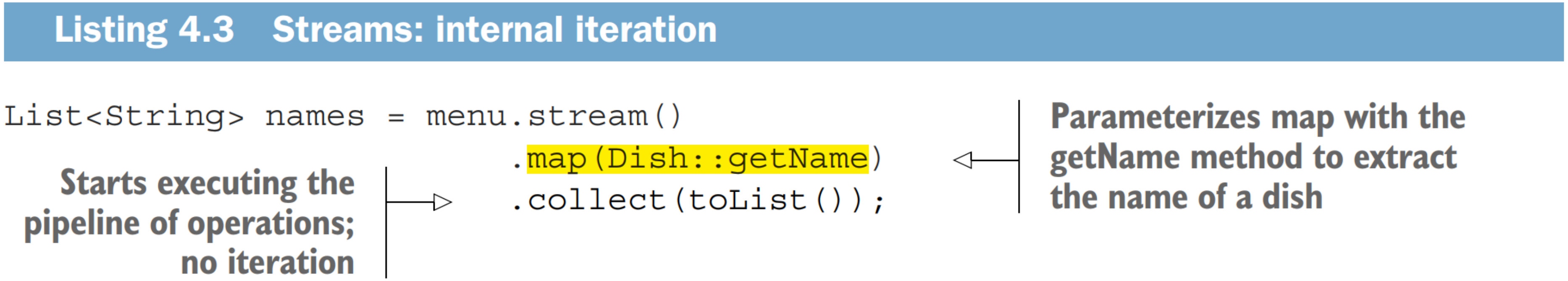
Note that the `for-each` hides some of the iteration complexity. The `for-each` construct is syntactic sugar that translates into something much uglier using an `Iterator` object.

Listing 4.2 Collections: external iteration using an iterator behind the scenes

```
List<String> names = new ArrayList<>();
Iterator<String> iterator = menu.iterator();
while(iterator.hasNext()) {
    Dish dish = iterator.next();
    names.add(dish.getName());
}
```

Iterates explicitly

External vs. Internal Iteration (Cont.)



Listing 4.3 Streams: internal iteration

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

Parameterizes map with the getName method to extract the name of a dish

Starts executing the pipeline of operations; no iteration

- You iterate a collection *externally*, explicitly pulling out and processing the items one by one.
- Using an internal iteration, the processing of items could be transparently done in parallel or in a different order that may be more optimized.
- The internal iteration in the Streams library can automatically choose a data representation and implementation of parallelism to match your hardware.

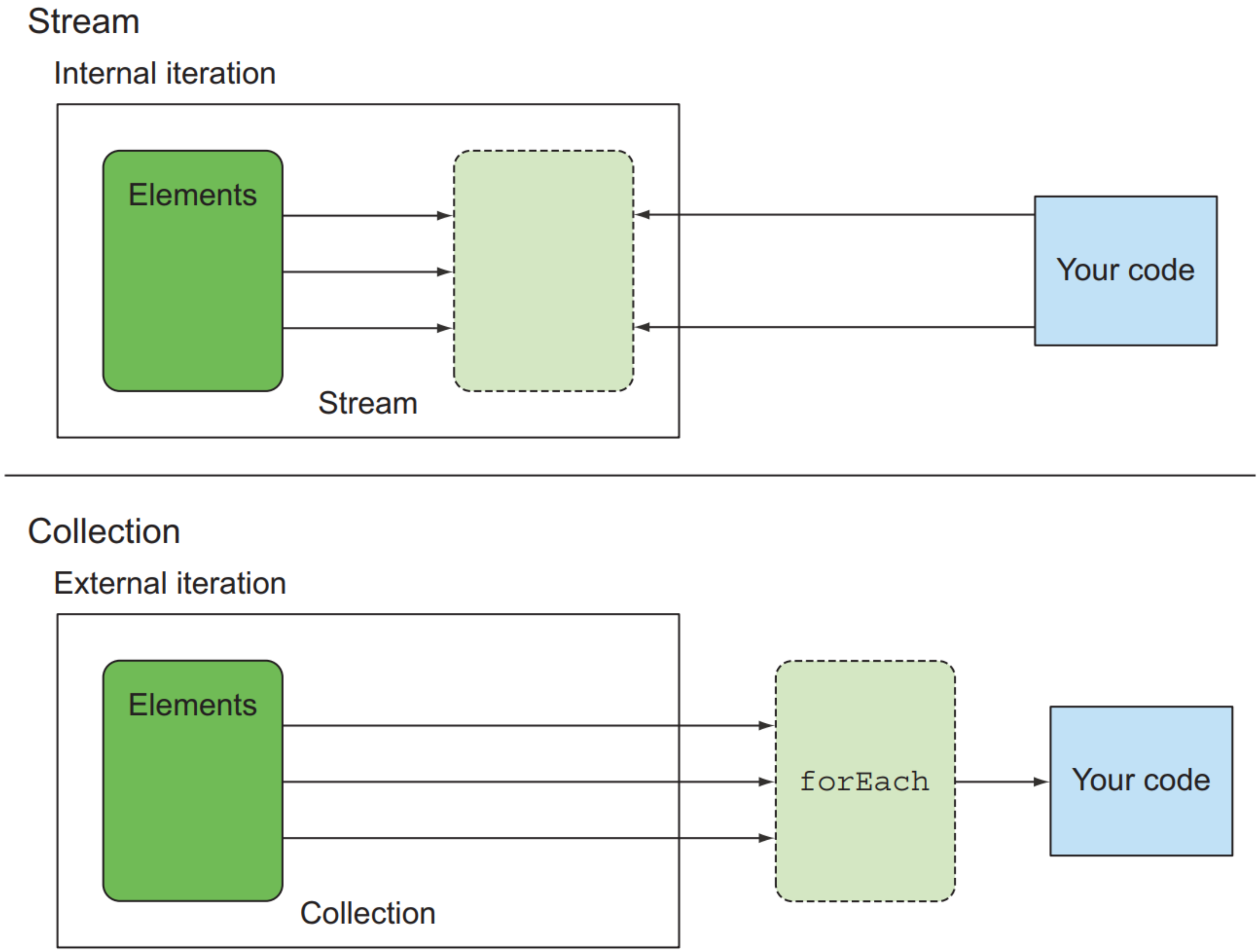
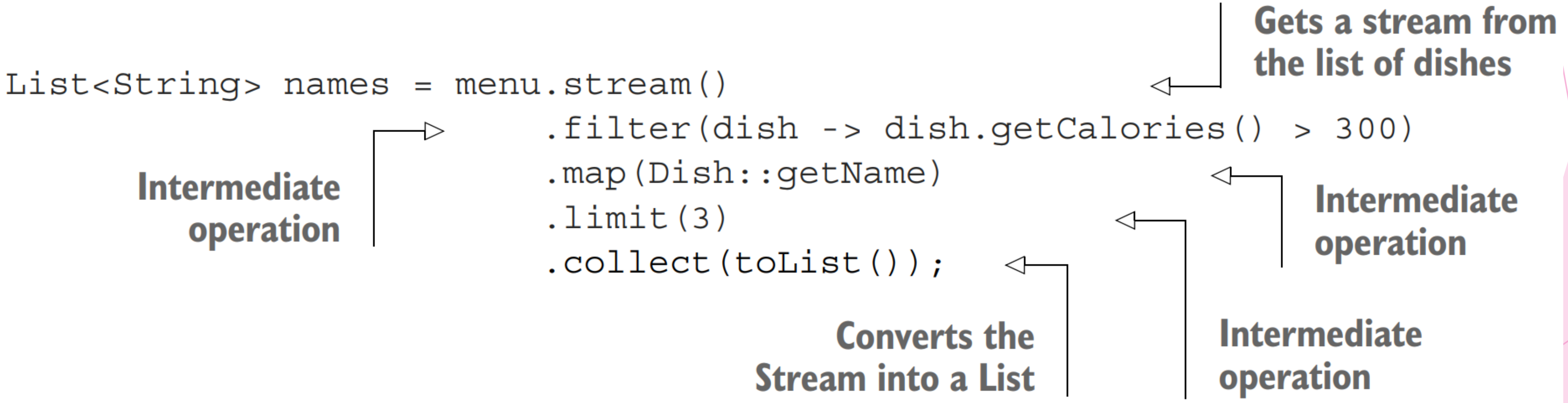


Figure 4.4 Internal versus external iteration

4.4 Stream operations

- Two Categories of Stream Operations
  - Intermediate Operations
  - Terminal Operations



```
List<String> names = menu.stream()
                         .filter(dish -> dish.getCalories() > 300)
                         .map(Dish::getName)
                         .limit(3)
                         .collect(toList());
```

Gets a stream from the list of dishes

Intermediate operation

Intermediate operation

Intermediate operation

Converts the Stream into a List

You can see two groups of operations:

- `filter`, `map`, and `limit` can be connected together to form a pipeline.
- `collect` causes the pipeline to be executed and closes it.

Stream operations that can be connected are called *intermediate operations*, and operations that close a stream are called *terminal operations*. Figure 4.5 highlights these two groups. Why is the distinction important?

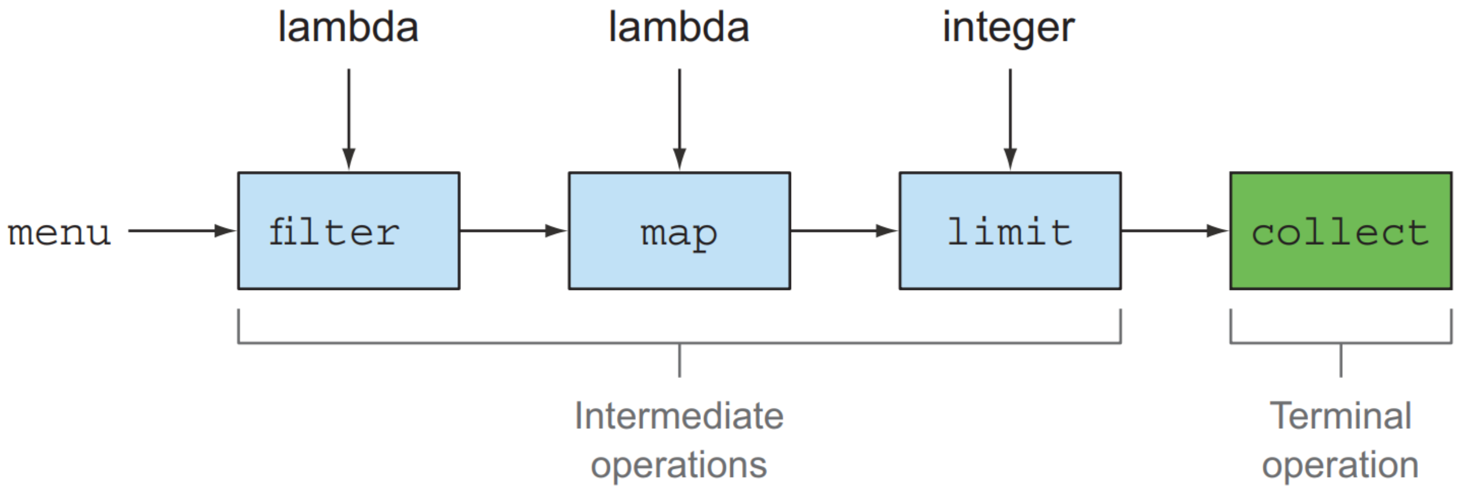
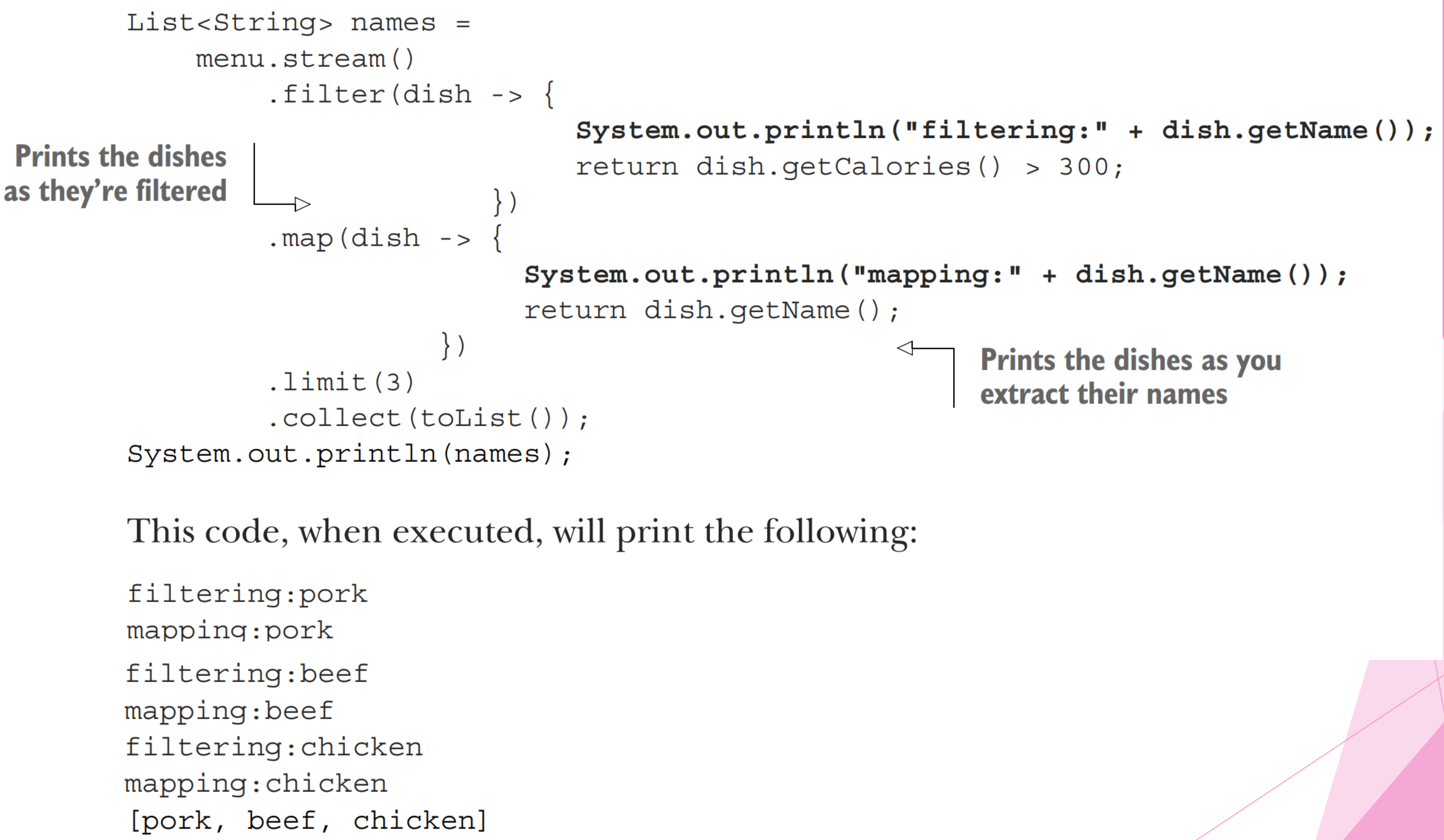


Figure 4.5 Intermediate versus terminal operations

- Intermediate Operations
  - These operations can be connected to form a query.
  - Intermediate operations don’t perform any processing until a terminal operation is invoked on the stream pipeline – they are lazy.
- Terminal Operations
  - Terminal operations produce a result from a stream pipeline.

```
menu.stream().forEach(System.out::println);
```



```
List<String> names =
    menu.stream()
        .filter(dish -> {
                          System.out.println("filtering:" + dish.getName());
                          return dish.getCalories() > 300;
                     })
        .map(dish -> {
                       System.out.println("mapping:" + dish.getName());
                       return dish.getName();
                  })
        .limit(3)
        .collect(toList());
System.out.println(names);
```

Prints the dishes as they're filtered

Prints the dishes as you extract their names

This code, when executed, will print the following:

```
filtering:pork
mapping:pork
filtering:beef
mapping:beef
filtering:chicken
mapping:chicken
[pork, beef, chicken]
```

- Working wit Streams
  - To summarize, working with streams in general involves three items:
    - A *data source* (such as a collection) to perform a query on
    - A chain of *intermediate operations* that form a stream pipeline
    - A *terminal operation* that executes the stream pipeline and produces a result
  - The idea behind a stream pipeline is similar to the builder pattern (see http://en.wikipedia.org/wiki/Builder_pattern).

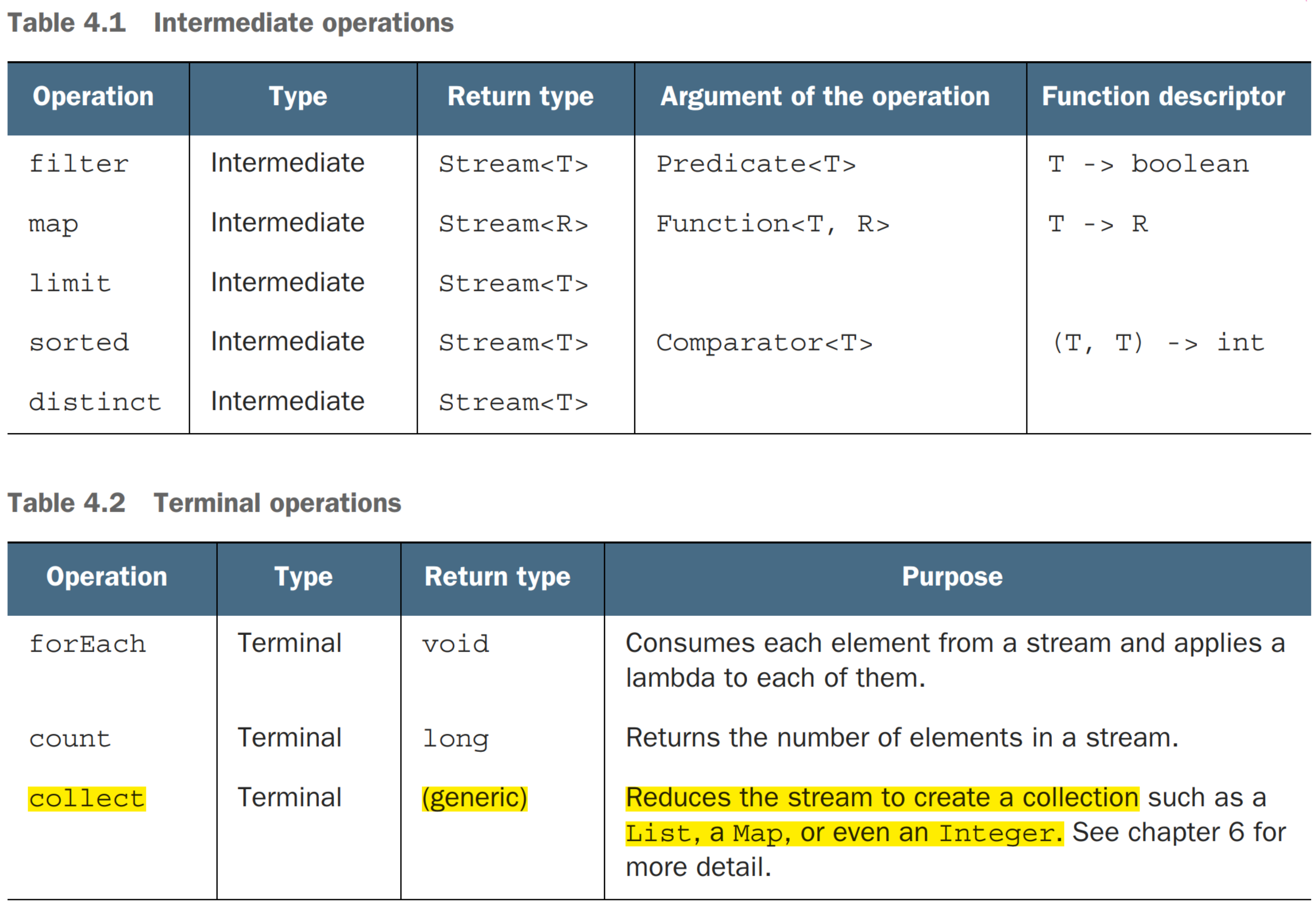


Table 4.1 Intermediate operations

Operation	Type	Return type	Argument of the operation	Function descriptor
filter	Intermediate	Stream<T>	Predicate<T>	T -> boolean
map	Intermediate	Stream<R>	Function<T, R>	T -> R
limit	Intermediate	Stream<T>		
sorted	Intermediate	Stream<T>	Comparator<T>	(T, T) -> int
distinct	Intermediate	Stream<T>		

Table 4.2 Terminal operations

Operation	Type	Return type	Purpose
forEach	Terminal	void	Consumes each element from a stream and applies a lambda to each of them.
count	Terminal	long	Returns the number of elements in a stream.
collect	Terminal	(generic)	Reduces the stream to create a collection such as a List, a Map, or even an Integer. See chapter 6 for more detail.

4.5 Road map

- In the next chapter, we'll detail the available stream operations with use cases so you can see what kinds of queries you can express with them. We look at many patterns such as filtering, slicing, finding, matching, mapping, and reducing, which can be used to express sophisticated data-processing queries.
- Chapter 6 then explores collectors in detail. In this chapter we have only made use of the collect() terminal operation on streams (see table 4.2) in the stylized form of collect(toList()), which creates a List whose elements are the same as those of the stream it's applied to.