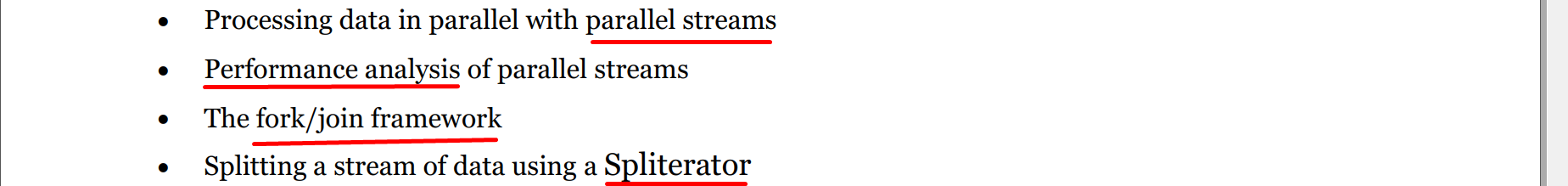
1. 
2. 
3. 
4. In last 3 chapters,
   1. we saw how Stream Interfaces let us to manipulate data in a **Declarative Ways.**
   2. We also saw that the shift from external to internal iteration enables the native java library to gain the control over processing the elements of a stream. Thus, relieving the developers from implementing the explicit optimization necessary to speed up the processing of collections of data.
5. **Issue when processing collection of data in parallel up to Java 7**
   1. 1st explicitly split the Data Structure containing data into subparts.
   2. Assign each subpart to a different thread.
   3. Synchronize them to avoid race condition.
   4. Wait for completion of all threads.
   5. Finally, combine all the partial results.

Java 7 introduced **Fork/Join Framework** to perform these operations more consistently and in less error-prone style.  
We will explore this framework in **section 7.2.**

1. In this Chapter, we will see how Stream Interface gives us the opportunity to execute operations in parallel on a collection of data without much effort.
2. It lets you **declaratively** turn a **sequence stream** into **parallel stream**.  
   It uses **Fork/Join Framework** for this.
3. We will see.
   1. How parallel streams works under the hood using Fork/Join Framework.  
      Because if we ignore it, we can obtain unexpected (more likely wrong) results by misusing them.

# 7.1 Parallel Streams

1. **Parallel Stream**: A stream which splits its elements into multiple chunks and process each chunk with a different thread.
2. Thus, we can automatically partition the workload on all the cores of our multicore processor and keep them equally busy.
3. Let’s take one example to understand this.
4. Suppose a method which sum all the numbers from 1 to the passed argument.
5. Start reading from here. 