

CSCI 4401

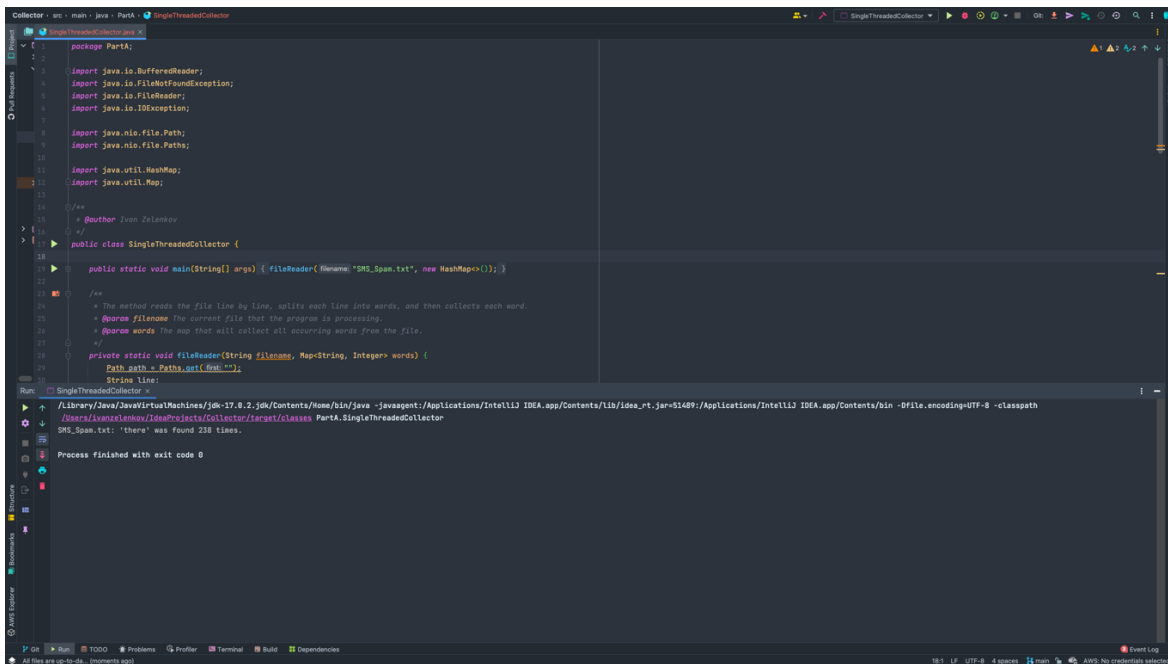
Dr. James Wagner

Ivan Zelenkov

30 October 2022

Observations

PartA.1



The screenshot shows the IntelliJ IDEA IDE with the `SingleThreadedCollector.java` file open. The code is as follows:

```
package PartA;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

import java.nio.file.Path;
import java.nio.file.Paths;

import java.util.HashMap;
import java.util.Map;

/**
 * @author Ivan Zelenkov
 */
public class SingleThreadedCollector {

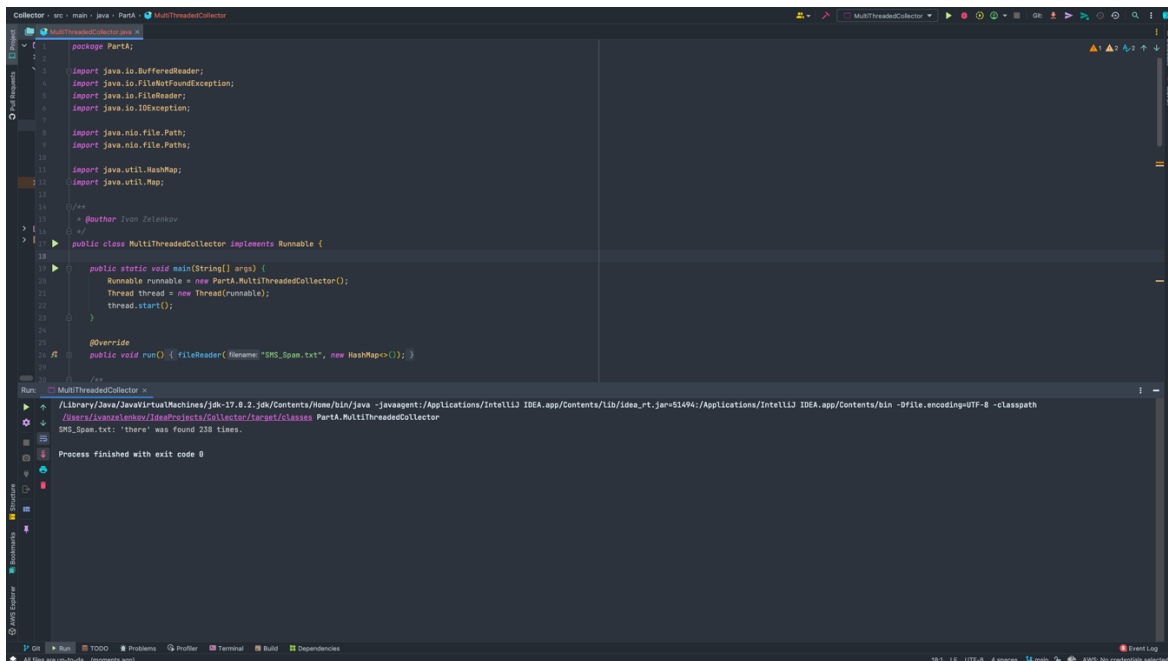
    public static void main(String[] args) {
        FileReader fr = new FileReader("SMS_Span.txt");
        new SingleThreadedCollector().fileReader(fr);
    }

    /**
     * The method reads the file line by line, splits each line into words, and then collects each word.
     * @param filename The current file that the program is processing.
     * @param words The map that will collect all occurring words from the file.
     */
    private static void fileReader(String filename, Map<String, Integer> words) {
        Path path = Paths.get(filename);
        String line;
        try {
            BufferedReader br = new BufferedReader(new FileReader(path.toFile()));
            while ((line = br.readLine()) != null) {
                String[] wordsArray = line.split(" ");
                for (String word : wordsArray) {
                    words.put(word, words.getOrDefault(word, 0) + 1);
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

The Run window shows the following output:

```
Process finished with exit code 0
```

PartA.2



The screenshot shows the IntelliJ IDEA IDE with the `MultiThreadedCollector.java` file open. The code is as follows:

```
package PartA;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

import java.nio.file.Path;
import java.nio.file.Paths;

import java.util.HashMap;
import java.util.Map;

/**
 * @author Ivan Zelenkov
 */
public class MultiThreadedCollector implements Runnable {

    public static void main(String[] args) {
        Runnable runnable = new PartA.MultiThreadedCollector();
        Thread thread = new Thread(runnable);
        thread.start();
    }

    @Override
    public void run() {
        fileReader("SMS_Span.txt", new HashMap<>());
    }

    /**
     * The method reads the file line by line, splits each line into words, and then collects each word.
     * @param filename The current file that the program is processing.
     * @param words The map that will collect all occurring words from the file.
     */
    private static void fileReader(String filename, Map<String, Integer> words) {
        Path path = Paths.get(filename);
        String line;
        try {
            BufferedReader br = new BufferedReader(new FileReader(path.toFile()));
            while ((line = br.readLine()) != null) {
                String[] wordsArray = line.split(" ");
                for (String word : wordsArray) {
                    words.put(word, words.getOrDefault(word, 0) + 1);
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

The Run window shows the following output:

```
Process finished with exit code 0
```

PartB.3

The screenshot shows the IntelliJ IDEA IDE with the `SingleThreadedCollector.java` file open. The code is a Java class that reads a directory of CSV files and counts the occurrences of words. The output window shows the results of the execution, including the execution time and the count of words for various files.

```
import java.nio.file.Paths;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class SingleThreadedCollector {

    public static void main(String[] args) {
        double startTime = System.currentTimeMillis();
        System.out.println("Reading in progress...");

        for (String filename : fileIndex()) {
            FileReader(filename, new HashMap<>());
        }

        double endTime = System.currentTimeMillis();
        double executionTotalTime = (endTime - startTime) / 1000;
        System.out.println("Execution time: " + executionTotalTime + " seconds");
    }

    // The method searches for the files in a specified directory and stores filenames in a List.
}

// Author: Ivan Zelensky
// collector class root files in a folder with executor service, count occurred words,
// and return an output that will contain the filename, the most occurred
// word in that file, and the number of times that word was repeated.

public class SingleThreadedCollector {

    public static void main(String[] args) throws ExecutionException, InterruptedException {
        System.out.println("Reading in progress...");
        double startTime = System.currentTimeMillis();

        // Create a thread pool with 4 threads
        ExecutorService executorService = Executors.newFixedThreadPool(4);

        List<Callable<Task>> list = new ArrayList<>();

        // Future represents the result of an asynchronous computation.
    }
}
```

Run

Execution time: 165.297 seconds

Process finished with exit code 0

Build completed successfully in 1 sec, 455 ms (13 minutes ago)

PartB.4

The screenshot shows the IntelliJ IDEA IDE with the `MultiThreadedCollector.java` file open. The code is a Java class that reads a directory of CSV files and counts the occurrences of words using a multi-threaded approach. The output window shows the results of the execution, including the execution time and the count of words for various files.

```
import java.util.Map;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

// Author: Ivan Zelensky
// collector class root files in a folder with executor service, count occurred words,
// and return an output that will contain the filename, the most occurred
// word in that file, and the number of times that word was repeated.

public class MultiThreadedCollector {

    public static void main(String[] args) throws ExecutionException, InterruptedException {
        System.out.println("Reading in progress...");
        double startTime = System.currentTimeMillis();

        // Create a thread pool with 4 threads
        ExecutorService executorService = Executors.newFixedThreadPool(4);

        List<Callable<Task>> list = new ArrayList<>();

        // Future represents the result of an asynchronous computation.
    }
}
```

Run

Execution time: 15.194 seconds

Process finished with exit code 0

All files are up-to-date. (no restart app)

Was there a difference in your runtimes between Part B.3 and Part B.4? Why or why not?

The runtime of the single-threaded version (PartB.3) was 165.297 seconds, and the multithreaded version using 14 threads (PartB.4) was 15.194 seconds. From this data, I can infer that the multithreaded version is 10.879 times faster than the single-threaded version.

The implementations are different and as we can see the second multi-threaded version (PartB.4) is much faster because I used an Executor with a thread pool. There was no good result when I was converting the single-threaded version (PartB.3) to the multithreaded version by implementing the Runnable interface because it did not give me such great speed up and was very cumbersome as opposite using Executor and implementing the Callable interface.

Implementing a Runnable interface, creating a Thread object, passing Runnable to its constructor, and then starting threads one by one where each of them will call the run() method is not a good way to work with multithreading in Java nowadays. Instead, Oracle simplified things for us by providing an Executor that can process tasks asynchronously without having to deal with threads directly.

With such a simple implementation, I was able to test the multithreaded version with any number of threads. All I had to do is change an argument when calling the newFixedThreadPool method of class Executors.

		Comparison with the previous implementation of the single-threaded version (165.297 sec) (n times faster)
Number of Threads	Execution Time (sec)	
1	60.065	2.750
2	34.474	4.794
3	36.011	4.590
4	36.902	4.479
5	40.185	4.113
6	53.091	3.113
7	44.779	3.691

8	40.603	4.071
9	55.315	2.988
10	59.663	2.770
11	17.664	9.357
12	24.649	6.706
13	22.566	7.325
14	15.194	10.879

Based on the table above, we can see that a multithreaded program with 14 threads runs in 15.194 seconds, which is the fastest execution time I got. If we run this program with only 1 thread, then it will give us 60.065 seconds, but it is still much faster than the single-threaded version with a different implementation by 2.75 times.

Was there a difference in your output between Part B.3 and Part B.4? Why or why not?

When I was working with multithreaded version that was implementing Runnable interface, I had a different output comparing to the single-threaded version. Single-threaded program returns result about files as they are stored and processed in alphabetical order. The output of the multithreaded first version was every time different because some thread-n was finishing reading its assigned files and returning result faster than others which is normal for multithreaded programs.

The implementation that uses Executor returns the output the same as a single-threaded program. The tasks are added to the list of callable tasks where each of them represents an object of CallableTask class. That class implements a Callable interface where I implemented a required call() method that will be called when the task is submitted.

```
callableTasks.add(new CallableTask(f.getAbsolutePath()));
```

Then they will be submitted in the same order to the result list,

```
result.add(executorService.submit(callableTask));
```

processed asynchronously and returned to result list as Future objects that contain the result of computations. All that we have left to do is to iterate through Future objects as they are stored in a result list and output the result of their computations to the user's screen.

CarAds.csv: 'other' was found 1964289 times.
Traffic_Violations.csv: 'district' was found 1253020 times.
MoviePlots.csv: 'their' was found 52952 times.
Oscar2019Tweets.csv: 'profile' was found 1646901 times.
ConsumerComplaints.csv: 'closed' was found 487805 times.
FakeNews.csv: 'trump' was found 89279 times.
IMDB.csv: 'https' was found 120971 times.
Resume.csv: 'class' was found 320978 times.
NYC_Restaurants.csv: 'inspection' was found 739947 times.
Hotel_Reviews.csv: 'united' was found 553549 times.
SMS_Spam.txt: 'there' was found 233 times.
WineReviews.csv: 'valley' was found 77460 times.
TimeUse.csv: 'related' was found 82 times.
UFOReports.txt: 'minutes' was found 7785 times.