THNGEO002

Concurrent Programming: Falling words

A multithreaded Java program, ensuring both thread safety and sufficient concurrency for it to work well.

Georgeo Thanathara

9/29/2019

Table of Contents

Classes and Modifications

Please note these are just descriptions of what the classes do, The handling of the concurrency safety aspects are talked about later.

wordThreads.java

I created a wordThreads class that implements the Runnable such that the instance of this class can be executed by a thread. Runnable objects are much more flexible that extending the Thread class, this is because Runnable class' allows extending from any other class.

The run method of this class contains the code that needs to be executed by the running Thread. This run method handles the dropping of the words on the screen by getting the getSpeed method in the WordRecord class, this allows each word to drop by a random amount. The gui is then repainted and the thread is put to sleep for 20 milliseconds.

The wordThread also handles the missed counter, by first checking if the words have reached the red bar on the bottom of the screen, If it has, the words are rest and brought to the top while the missed counter increments. But during this check the run method also checks whether the maximum number of words has been reached. In the case where it has been reached, a message window pops up on the screen showing the user their results of the game.

wordEntry.java

This is an inner class that I created In the WordApp class that also implements the Runnable interface. The run method in this class ensures that when a word is typed, it is compared against all the current words on the screen and depending on whether the correct word was typed, it will increment the caught and the score.

Like the wordThread class , this class also does a checking to see whether the maximum number of words has been reached by using the getTotal method within the Score class and similarly a message window pops up giving the user the results.

WordApp.java

The WordApp class was modified such that GUI is responsive to its respective buttons.

- **Start :** The start button on the screen first sets the volatile Boolean "check" in wordPanel to true (More about this later) and creates a thread while passing in the Runnable wordPanel object. The thread then begins executing by using the start Method. The visibility of the start button is then set to false such that it cannot be clicked again.

- **Pause :** The pause button only sets the check variable to false such that the items on the screen stop moving. It then sets the visibility of the start button to true to be able to continue the game from where is was stopped.

- **End** : The end button stops the game by setting the Boolean to false, A message dialogue then pops up showing the users results for the current game. When “ok” on the message dialogue is clicked, the screen is reset and the visibility of the start button is set to true.

- **Quit :** The quit button is to exit the application and end the program. To do this the dispose method, followed by System.exit is used to ensure safe quitting.

WordPanel.java

The run method in the wordPanel class was modified such that it creates a thread for every word on the screen. The thread begins its execution when it uses the start method to trigger the run method in the wordThread class. The run method also creates an object of the wordThread class by passing the “I” value and the class itself as parameters in the constructor within the for loop.

Score.java

I ensure that the score class used atomic variables rather than synchronizing every method to guarantee of isolation from interrupts by concurrent threads. More about this is explained in the next section.

Concurrency Features

A list and description of all the concurrency features that I have used within my program to ensure the safest and most accurate results. The next section explains how these features were used within the program.

Atomic Variables

Atomic operations in java are used such that a single unit of task can be performed while there is no interference from any other operations. When it comes to multi-threaded environments, this is very significant as it makes sure that there is isolation and is interrupt free when other threads are using the same data.

Atomic variables were necessary for this program as we have multiple threads that are accessing the same variable and so we can minimize the use of synchronization blocks and methods, thus the program is more readable and the code is less prone to errors. Atomic operations are also generally faster and thus more efficient than synchronize. Although, it does have certain downfalls and that is why I had to use synchronize in the other parts of my program.

Synchronize

Synchronized blocks and methods are built-in Java locking mechanism for enforcing atomicity via reentrant locks. It uses the “synchronized” keyword to distinguish between what is synchronized or not, and once a certain portion of the code is synchronized it means that only one thread can execute inside the block and thus, the other threads are blocked until the current thread within the synchronized block completes executing.

In this program I used synchronized methods and synchronized blocks. Synchronized methods were used in the score class where atomic operations were not useable due to bad interleavings. The provided WordRecord class also had synchronized methods which helped in concurrency aspects of this assignment. Synchornized methods ensure that threads execute on the instance that the method is part of (in this case it is the score and WordRecord classes). Whereas synchronized blocks were used in the run method of the executing threads, these were necessary as it ensured that counters were not over incremented and authenticity remained. The next sections explained how it maintained accuracy.\

Volatile Variables

The keyword volatile is an indication to the compiler and Threads to always read the value from main memory. This keyword also guarantees visibility and ordering and prevents reordering of code within synchronized blocks.

The volatile variable “check” was a very important variable for this code to start and stop the threads from executing as you will see in the next section.

Ensuring….

Thread Safety

Thread safety was an important factor to consider in this project. As there are multiple threads running in the background constantly doing checks, we are bound to get race conditions. To minimize these race conditions and ensure each thread is executing on its own without interrupting other threads, I used the concurrency features that are mentioned above.

There are a couple of places that require protecting of data. Firstly, the score class, the intial score class was just a class with setters and getters but once the threads were created, I noticed race conditions, where the results were not accurate. The caught and missed were being wrongly incremented because multiple threads were accessing it and wrongly changing the values. To resolve this issue, I made the variables in the score class atomic variables, this ensures that other threads would not interrupt the variables when one thread was reading and writing to these variables. Thus it was very important to change the variables from regular integers to atomic Integers. For example, in the condition where I check if the maximum number of words has been reached, I had to use the getTotal method from the score class, this was initially giving me issues as some threads hadn't executed yet, and the getTotal did not return the correct value, this issue was fixed by using atomicity.

The volatile variable "check" that I created was another enforcer of thread safety. Infact, it was one the most important steps, since there is no way to stop a thread anymore other than the interrupt() method(which is not ideal because we do not know when the thread will stop executing) I had to use a Boolean flag that was set to true in the run method in a while loop. This helped stop the threads at any point in the game (eg pause, end) it also helped the game automatically stop by setting the Boolean to false when the maximum words was reached. Once the Boolean is set to false, the threads are killed and cannot execute any more hence not allowing for race conditions as the variables cannot get incremented.

Thread Synchronization

Other than the provided synchronized methods that were in the given wordThread class, I made the caughtWord method in the score class synchronized aswell. I also used a synchronized method to return the typed text in my inner class to make sure that the correct word is compared to when the threads for checking is executing. This will allow words to be written even at super speed but still give the correct result. Thus eliminating race conditions where the incorrect word is compared to.

I also used thread blocks in both my run methods when I was comparing the totalWords to the maximum number of words in order to give the appropriate response. This was necessary, because if otherwise, the missed/caught counter may give in correct results because the threads may have not executed yet.

Liveness and DeadLocks

To prevent liveness and deadlocks I used synchronized methods, these methods were also independent of each other which resolved deadlock issues and none of threads accessed the shared data which would've causing them to provide incorrect results.

Validation of results

In order to validate the results, I made all the words on the screen to drop by the same amount, This would show me whether there were any race conditions when the threads were executing. Initially, I was getting wrong results (the total words exceeded the maximum words) but after careful analysis and implementation of the above stated concurrency tactics, I was able to resolve this and it gave me the correct results.

I played around with the program multiple times and can confirm that I did not see any problems to the eye. This does not mean that there are no errors at all ofcourse.

Model-View-Controller

The wordDictionary, wordRecord and Score class all fall under the model. This is because these classes are independent of the user interface. These classes are only used to directly manage the data and logic of the application.

The wordApp and wordPanel are in charge of the animation on the gui, thus this falls under the view component, because it is used to represent the information(in this case words, buttons and listeners) on the screen.

Finally, the WordThread and WordEntry is the controller, This is because these classes are in charge of the actual actions caused by the user. It allows the words to drop on the screen, it takes care of any word entered by the user and it is also constantly checking to see if there is any change that needs to manipulate the model (eg maximum words reached).

Additional Features

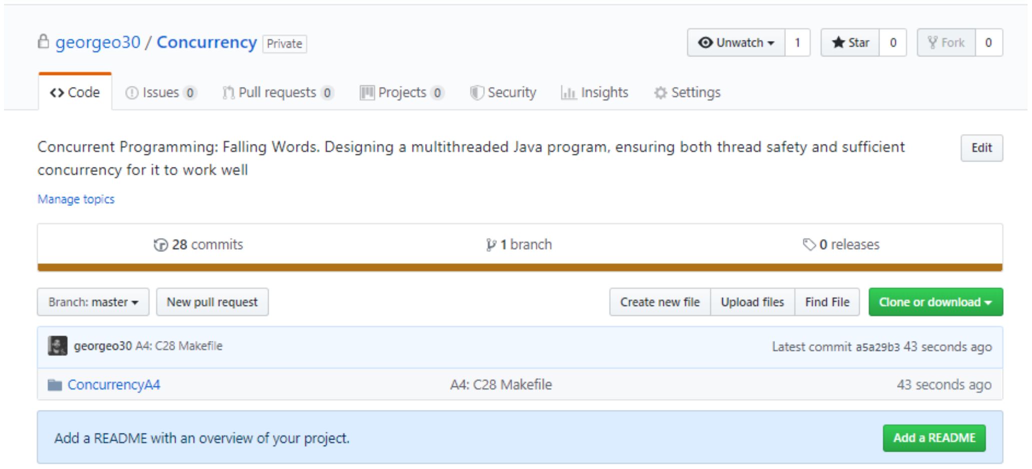
A highscore can be set. If your score is higher than the current high score then your high score will change, otherwise it stays the same. The high Score will be displayed whenever the game is over not when it is ended.

A pause button was also added.

GitHub

To find my repo for this assignment please click on the link : https://github.com/georgeo30/Concurrency

If the above link does not work, please click on this link and navigate yourself to the Concurrency repo: https://github.com/georgeo30



Concurrent Programming: Falling Words. Designing a multithreaded Java program, ensuring both thread safety and sufficient concurrency for it to work well

28 commits | 1 branch | 0 releases

georgeo30 A4: C28 Makefile — Latest commit a5a29b3 43 seconds ago

ConcurrencyA4 — A4: C28 Makefile — 43 seconds ago

Add a README with an overview of your project.