CSC2002S

Practical Assignment 4

Concurrency Programming:

Falling Words

Niceta Nduku

NDKNIC001

30 September 2019

Introduction

Falling words is a game that allows a user to test their typing speed. Words fall at different speeds and they must be caught (user input) before they fall off the screen (panel). The objective of this report is to show how the game was made successfully thread safe using java concurrency methods.

Method

The skeleton code that was provided was insufficient to ensure that the game runs efficiently and therefore the following modifications were made.

Controller
This is a class that handles most of the work of the game. This class was added to help the game stick to Model View Controller architecture. Controller updates the view and communicates with the classes Score and WordRecord.

It has five boolean attributes; ended, paused, running, modified and newScore. These booleans give the state of the game at any one point.

A new contoller instance will take in the number of words falling at any moment, the total number of words, the array of words that are going to be falling at any moment and a score object. Controller will then set its attributes: noWords, totalWords, words (the array) and the score, to the ones that were passed when the new instance was created. An atomic integer is then set to zero which will be used to keep a count of the words falling. Still withing the constructor, an array of WordThread objects is created for each of the words in the words array. All the boolean attributes except ended are set to false.

The methods used in contoller will be discussed in the rest of the report

WordThread
WordThread is an inner class of Contoller. It extends Thread and it is used to control what happens to each word. A new WordThread is created by taking in a

word of type WordRecord and a controller instance. It the has attributes word, controller, and a boolean destroyed.

When run, while the game is not over and the thread has not been destroyed; if the game is paused, nothing is done. If the word was caught and totalWords minus the atomic integer, threadCount, is greater than noWords, the word is reset. If the difference between threadcount and totalWords is less than noWords, the thread is ended. If a word has been dropped, missed() in controller is called and the thread is also destroyed if there are no more words in the reserve. Otherwise, a word is dropped by one and the modified boolean in controller is set to true.

The thread will then sleep for a duration of the word's speed divided by 20 before rerunning.

WordPanel
Word panel is started when the start is clicked. In run(), the panel will repaint if there there is a a change in the contoller i.e. isChanged() is called that returns the boolean modified within contoller. If the game is paused, it continues without repainting. If the game is not running, repaint will place on the screen a prompt to press start and the current scores.

WordRecord
A boolean caught was added to Word record and method caught() that returns the boolean. When matchWord is called, caught is set to true and when the word is reset, it set back to false.

WordApp
When WordApp is run, on top of the task the main method was performing previously, it creates a new contoller insance of Controller. When setting up the GUI, after the labels for the scores are created, a new Jlabel table is created and the table scoresTable in controller is set to that table. This was done in order to update the scores without having to pass the values back to the mainApp.

The action listener for the text field was changed so that when the user types in a word, the action performed will be calling checkAnswer method within the controller class which will check if the answer is correct.

The action listener for the start button checks if the fame has been restarted, either while the game is running or has ended, and it refreshes the game (controller.resetGame()). If the button text is "Start", it will simply start the game by calling runGame() in the controller class.

The pause button will pause the game if the game is running and if pressed again, the game will continue (controller.continueGame()).

The end button calls controller's endGame() and the exit button terminates the whole game GUI.

Concurrency

In the controller, all the booleans, and the wordCount are variables that are modified and accessed by different thread (either a wordThread or the WordPanel). The booleans were therefore all set to volatile and the wordCount made an atomic integer.

The following methods are used by threads or change or use variables that are used by threads and were therefore made synchronized: CheckAnswer, missed, resetState, endThread, getScores, countThreads and all the methods that return the volatile booleans.

A number of WordThreads equal to the number of words to be falling at a time are created when the game is run (start/restart). The threads deal with individual words. WordRecord was already made with synchronized methods so when a thread calls a method from Word record, safety is ensured. Threads also do not manipulate the same WordRecord.

When a Word is dropped, missed in controller is called. Within missed, updateScores is called. Update scores contains a lock to esnure that the score labels are only being updated one thread a time. It is in a try/ finally block that updates the scores if they need to be, then unlocks. The block ensures that there is no deadlock whereby a thread hold the lock and others cannot manipulate the data.

Validation

The game was run with different number of words from 5 – 50 total words and different words falling at a time. The game was played in three ways. Firstly, all the words are missed. The next way was to catch all and finally miss some and catch all.

Bad interleaving was discovered when the last word is caught or missed. The game would end and the atomic count would be reset before a thread would end. The thread would then assume that there are words left and the game would not end, but one thread would remain running. This was modified by having an atomic integer that ensures that the number ofwords that were generated never exceeded the total amount. Threads would be terminated if the number of words fallen reached the total.

Data races were not found when the game was run multiple times and with extremly fast words. This was done by placing print statements in critical sections such as countThreads and missed where threads can call those frequently. The print statements would output an integer of the thread count.

Other print statements were added to check if threads were indeed being terminated when the the words are almost at the max i.e. difference between the theadCount and total is less than the number that should fall.

MVC

Controller

The controller in this game is the Controller class together with the WordThread inner class. When a user clicks a button or enters a word. Methods from controller will be called and controller will set specific variables as true for the panel thread to know. Contoller also updates the scores and hence the score labels when there is a new score, i.e. when missed or caught.

Once there is any modification made on the panel, it calls resetState in contoller to let it know that it saw the change (made by a word) and sets the modified boolean to false. A drop in a word sets the modified boolean t in controller to true which the panel will see and update.

View

The view is the WordPanel and WordApp. WordApp creates the main GUI and WordPanel is where the words fall. The user sees the falling words and can interract with the game through the buttons and the text input. When the user performs an action, the action listeners in WordApp will let the contoller know by calling the methods as described above.

Model

The model part of the game are the classes WordRecord, Score and WordDictionary. The controller will update the score and retreive score information before updating the view. The WordThreads will use methods in the WordRecord class to modify aspects about each word and get information on their state.

Beyond scope

The contoller class was a way to ensure that one class handles everything. It eased the work of the game by ensuring that threads do minimal work and important variables in the game such as the score and words are safely modified.

Other UI changes such as modifying the buttons when pressed and messages after a game is played were added.

Conclusion

Ensuring that the game is thread safe was done by synchronization and locking. Having one class that enured that important objects are not mishandled made the game more thread safe.