CSC2002S: Assignment 4

Leanne January

30 September 2019

Contents

1 Introduction

1.1 Aim

The aim of this project was to implement efficient multi-threading in a game using concurrency to ensure safety and correctness.

1.2 Game Functionality

On Start, a specified number of words fall down from the top of the screen towards a red rectangle. The player must type all of the falling words and press ENTER to clear them off the screen before they are inside the red rectangle. The player is scored on the number of missed words, caught words, and the sum of the length of the words that they caught.

2 Changes and Modifications

2.1 WordApp.java

The WordApp class was modified so that the buttons in the given framework worked to achieve what is described above. A thread was created for the WordPanel class and the ScoreView class.

2.2 WordPanel.java

The WordPanel class was modified so that a WordPanel thread would spawn a thread for each word falling using the WordThread class, and animate the falling of the words.

2.3 WordThread.java

The WordThread class is a new class that was created. The WordThread increments the position of a word so that it appears to be moving down the players screen.

2.4 ScoreView.java

The ScoreView class is a new class that was created. The ScoreView thread constantly checks for changes made to a players score, and reflects these changes on the player's screen. The ScoreView thread also stops the game once the maximum number of words has been reached.

2.5 MessageView.java

The MessageView class is a new class that was created in order to display an appropriate 'Game Over' message to the player.

3 Implementation of Concurrency

The implementation of multi-threading allowed for multiple different threads to have access to the same shared, mutable data - notably the words that were falling, their position and the score of the game. It was, therefore, very important to make sure that mutual exclusion was introduced in order to make sure that each change to the shared data was atomic.

3.1 Thread Safety and Thread Synchronisation

3.1.1 Synchronisation

The synchronized keyword uses mutual exclusion to achieve thread safety. Mutual Exclusion means that only one thread at a time is allowed to access or update the shared data. The synchronized keyword is an example of mutual exclusion as it allows one to execute a method that uses the shared data while blocking all other methods from having access to that data. This is because the synchronized keyword represents that a lock has been acquired at the start of the method, and will only be released on the method's final bracket.

In my game, I used the synchronized keyword on all get and set methods in the Score, and WordRecord class. I also made sure that any other method used by a thread that had meaningful access to the shared data was also synchronized. For example, resetWord() from the WordRecord class; caughtWord(lenth) and missedWord() from the Score class; paintComponent() from the WordPanel class; and getNewWord() from the WordDictionary class.

3.1.2 Volatile Variables

I also used a Volatile Variable in order to achieve thread safety in my game. Volatile variables help achieve thread safety using Visibility. Visibility means that threads can 'see' the state of shared variables in other threads. This is what the Volatile keyword implements.

In the WordApp.java class, I used a Volatile boolean called done which is false when the game is running and true when it is stopped. The done variable was used by both the WordPanel thread and the ScoreView thread. When one of the threads changed the variable - the other thread needed to be see that the variable had been changed in order to stop or start the game. The Volatile variable allowed for this to be implemented properly, ensuring thread safety.

3.2 Liveness and Deadlock

Liveness can be described as certain properties that concurrent code must have. Concurrent code has liveness if all threads that need to access or update the shared data all have a chance to execute and that threads do not end up waiting for long periods of time to execute.

In my game, all the methods that were synchronized were very short and simple to execute - meaning that no thread ever has to wait too long to have access to the code and progress is being made quickly, and there is no deadlock where threads are preventing other threads from running and freezing the game. Liveness is further enhanced due to not all methods being synchronized - only the ones that really required it.

4 Model-View-Controller

4.1 Model

The Model manages the data, logic and rules of the application. For the WordApp game, the Model consists of:
1. The WordRecord class: WordRecord is an object used in the game to represent the words that are falling.
2. The Score class: Score is an object used to represent the score of the game. The score consists of the number of words caught, missed and the total game score which is the sum of the length of the words that are caught.
3. The Dictionary class: Dictionary is an object used to represent all the possible words that can fall.

These classes contain the methods needed by the Controller to implement the game rules. The View then updates the user's screen in order to reflect these changes.

4.2 View

The View is used to present data to the user. For the WordApp game, the View consists of:
1. The WordPanel class: WordPanel animates the falling of the words from the top of the screen to the bottom.
2. The WordThread class: WordThread helps the WordPanel class by imcrementing the y co-ordinate of a word.
3. The ScoreView class: ScoreView constantly checks the score and updates it for the user to see when it has changed.
4. The MessageView class: MessageView creates the Game Over message which shows the user their final score when the game ends.

These classes create a visual experience of the game.

4.3 Controller

The Controller accepts input from the user and converts it to commands for the Model or the View. For the WordApp game, the Controller consists of:
1. The WordApp class: WordApp is the main class of the game. It creates the initial view, and responds to user interactions in the game by ensuring that the appropriate reaction takes place.

5 Additional Feature

As an additional feature, I modified the way the falling speed of the words was implemented so that the falling speed starts slow and gets faster as words are caught or missed. I did this in order to mimic classic typing games which get progressively harder as you play.

5.1 Implementation

Initially, the falling speed was set to a random integer between 150 and 1500. In order to achieve progressively faster speed, I set the lower bound to 150 and the upper bound to 200 to make sure that the words would all start falling slowly - still with a degree of variation to their speed. Each time a word was reset, as a result of being caught or missed, I incremented the upper and lower bounds of the range by 100.

The first issue I had was that after some time playing the game, the words would fall too fast to type and soon trigger the end of the game. To resolve this, I used an if statement to ensure that the bounds were only incremented while the upper bound was less than 1500 - effectively making 1599 the maximum speed that a word can fall.

The second issue I had was that after one game ended and a new one was started, the words still had the final falling speeds of the last game. I resolved this by creating a method that resets the values of the bounds to 150 and 200. This method is called upon the end of a game.