Concurrency Assignment

Ryan Anderson (ANDRYA005)

19 September 2019

Description of Classes

We will start by describing the classes that were added, followed by noting all of the modifications made to the existing classes in the skeleton code.

Classes Added

1. `WordThread.java`

The `WordThread` class was created in order to implement threading. The class contains two **fields**: `word` and `timer`.

The `word` field is of type `WordRecord`. This variable stores the `WordRecord` object to be used within the thread. Each time the word on the screen relating to the thread is either correctly matched or reaches the red, this `WordRecord` object is modified. This is done by generating a new word from the word dictionary and setting the `text` of the relevant `word` to be equal to this new word. Furthermore, the `fallingSpeed` of the `word` is also changed.

The `timer` field is of type `Swing.Timer`. This variable is used to mimic a while loop with a slight pause, ensuring that the thread continues to run until the `timer.stop()` method is called. When the `WordThread` object is run, the program calls `timer.start()` in order to start the timer. The timer has an `ActionListener`, and each time the action listener is triggered, the thread first checks to see if (1) the thread's `word` is the empty string, suggesting that the thread should stop or (2) the "end" button has been pressed, stopping the thread. If these are false, the word is then dropped. Then the thread checks to see if the `word` is in the red. If it is, then a sound is played and the missed word is recorded. It then checks if the total number of words displayed is still below the total number desired and either creates a new word, if it is, or stops the thread, if it is not.

Finally, the class contains two **methods** (excluding `run()` and the constructor), `updateTotals(String word)` and `match(String testWord)`.

The `updateTotals` method simply plays a sound and updates the model when a word is caught.

The `match` method tests for equality between the inputted word and the current word in the thread. If the words are the same, `updateTotals` is called.

2. `EndGui.java`

The `EndGui` class was created purely to display the player's final results. The constructor creates a `JFrame` which displays the amount of words caught, missed and total score for their game. When the user clicks the "CloseWindow" button, the GUI is disposed of and all scores are reset for the next game.

Classes Modified

1. `Score.java`

All of the methods were individually synchronized.

2. `WordPanel.java`

A `Swing.Timer` object was added in order to repaint the panel frequently - accommodating any updates that might have occurred. The `actionPerformed` method associated with this timer repaints the Panel (as mentioned in the previous sentence) and checks if the game is finished (i.e. if the desired number of words had been displayed). If the game is finished, it then creates and shows the `EndGui` and stops the timer.

The `run()` method was then used to start the timer, and create threads for each "column" of words in the display (i.e. one thread for each initial visible word). These threads were then started.

It has a `testWord` method which is used to call the `match` method for each thread to check for equality of the entered word with a word currently on the screen.

3. `WordApp.java`

The main changes made here were to the `actionPerformed` methods for `textEntry`, `startB` and `endB`. We also added a "Quit" button and made some variables public and static in order to be accessible from outside the class.

The `actionPerformed` for `textEntry` was changed in the following ways. The entered word is only tested if it is not empty (to avoid complications). Then if the word does not match with any word on the screen, an error sound is played. But, if it does match, then a success sound is played (but that is handled in the specific thread that contains the word).

The `actionPerformed` for `startB` was changed in the following ways. The score is immediately reset. Then the caught, missed and score values on the GUI are updated. We then called `createWordArray()` to create the array of words to be used. Finally, the `WordPanel` object is then run.

The `actionPerformed` for `endB` was changed in the following ways. An instance variable `ended` is set to `true` in order to stop all of the threads. The score is reset and the labels for caught, missed and score are updated to reflect the reset.

A "Quit" button was added. When this button is pressed, the `System.exit(0)` is called, exiting the program.

Finally, the `Jlabel` objects `caught`, `missed` and `scr` were made public and static in order for them to be accessed (and update-able) from outside the class.

4. `WordRecord.java`

As mentioned later in the *Additional Features/Extensions* section, we modified the `fallingSpeed` calculation to account for the total amount of words that have been caught.

One method was created, called `exceededTest()`. This method checks to see if the number of words that have been displayed is more than the total number of words that were desired to be displayed. If the total number of words that have either been caught or missed is equal to the desired total, then the program is notified that the game should be finished.

The `matchWord(String typedText)` method was modified. The method still checks for equality between the two words of interest, however, it then uses `exceededTest` to determine whether a new word should be displayed. It does this by using a Boolean variable `exceeded`. The `resetWord()` method is then called in order to generate a new word, however, if `exceeded==true`, then the word is set to the empty string, and the relevant thread is stopped. Otherwise, a new word is generated and displayed.

Java Concurrency Features

The Java concurrency features used are as follows:

1. volatile variables
2. synchronized methods

Volatile variables

Three Boolean variables are declared as `volatile`, namely `ended` and `exceeded`. The variables `ended` and `exceeded` fulfill a similar role, they are a signal to tell the threads when to stop. They need to have the most up-to-date values of the Boolean because stale values could result in a bad interleaving, where a thread either matches with a word or reaches the red when it should have been stopped immediately. So, the most recent version of the variable is needed. Thus, declaring them as `volatile` is necessary.

Synchronized methods

Multiple methods within the `Score` and `WordRecord` classes are declared as `synchronized`.

The methods in the `Score` class had to be synchronized as they are used by multiple threads to read from and write to the score variables. If they aren't `synchronized` then you could have a situation where one thread writes a change to the variables but is overwritten by another thread (as in the counter example used in class). This would be an example of a data race.

Why would you need to have `WordRecord` synchronized.

Atomic Integers

The variables `missedWords`, `caughtWords` and `gameScore` of the `Score` class were declared as Atomic. The reason for this was that, even though the methods which get and set the values of these variables are `synchronized`, there could be a situation where one thread is reading the variable using its `get` method, while at the same time, another thread is incrementing the variable. This could result in the reading thread observing stale data or observing data that is too recent (i.e. it was meant to read the unincremented value).

Method for "Safe" Coding

The following sections outlines how we wrote code to ensure:

1. Thread safety (for both shared variables and the Swing library)
2. Thread synchronization where necessary
3. Liveness
4. No deadlock.

Thread Safety

In a program, you need to implement thread safety for any object that has a shared, mutable state. If there are multiple threads accessing an object you need to ensure that there is no room for *Data Races* and *Bad Interleavings* (i.e. Regardless of the scheduling or interleaving by the run time environment). If the object is immutable or not shared (i.e. can only be accessed by one thread and cannot be changed), then we do not need to worry about the thread safety of that object.

In this assignment, the data that could be accessed and changed by multiple threads include the variables of the `Score` class and the GUI components. The `missedWords` and `caughtWords` variables of the `Score` class are read and written to by the threads of the `WordThread` class, and thus they needed to be made safe. The GUI components are updated in the `WordPanel` class and are changed in the `WordThread` class. Thus, these also needed to be made thread-safe.

We will now explain how we ensured thread-safety for both the shared variables and Swing library (the GUI).

a. Shared variables

This was discussed on *Java Concurrency Features.*

b. Swing library

The Swing library is not thread safe. For it to be thread safe, you need to ensure that all changes made to the GUI are done through the Event Dispatcher Thread (EDT). We used the `Swing.Timer` class to achieve this. The `ActionEvent`handlers of `Swing.Timer` execute on the EDT. Thus, every time we needed to make changes to the GUI (updates and repainting etc.), these changes were made within the `actionPerformed` event handler of the timer, ensuring changes were made safely.

Thread synchronization where necessary

This was mentioned in *Java Concurrency Features* section.

Liveness

This is the concurrent programs ability to execute in a timely manner. The two main forms of liveness are *freedom from deadlock* (discussed under *No deadlock*) and *freedom from starvation*. The latter will be discussed now.

Starvation occurs when a thread is unable to gain regular access to shared resource and is unable to make progress which occurs when particular threads make the shared resource unavailable for long periods. We ensured that this wasn't the case by making the critical sections of our synchronized methods as short as possible and making all threads execute the same methods when necessary, such as `caughtWord`. This increased the general liveness of the program because threads are never blocked for a long period before being able to execute the method themselves. Furthermore, this prevented starvation in that no thread could be "greedier" (making the shared resource unavailable for longer) than another thread, because each thread ultimately performed the same operation on the shared resource.

No deadlock

We ensured that the program was deadlock-free by checking that no cycles of waiting threads could occur. This was done by in a few ways: (1) by ensuring no threads hold several locks at once. Thus, the order that the locks were aquired by the thread is irrelevant. (2) The blocks applied to the threads (in the *Score* class) were created in such a way that no blocking method uses another blocking method. Without any dependencies between these methods, there is no situation in which a thread could be waiting for one thread to release its lock, while another thread is waiting for that same thread to release its lock.

Validation and Error Checking

As we know, this is one of the trickiest aspects of parallel and concurrent programming. There is no certain way of testing the program to ensure that it is thread-safe as a race condition could only occur in the one millionth run. However, we did the following to try promote a situation where something could go wrong:

1. We tested our game on one word, with the number of words displayed on the screen set to 1000. We then entered the word into the text box to match with them all at once. This was a good test of how the program handled different threads matching with the same word concurrently, and if the relevant score values were safely updated (i.e. if the `caught` variable was 1000).
2. We tested our game on 1000 words, letting them all fall to the red at the same speed. This was a good test of whether concurrent "misses" were handled correctly, checking whether the `missed` variable contained 1000 words.
3. We tested our game on 1000 words again, but this time we tested the `ended` and `exceeded` Booleans. We did this by matching with arbitrary words and missing arbitrary words. When the game was finished, we checked whether `caught` plus `missed` was equal to 1000, ensuring that there was no lost writes etc.

For each of these tests, the program was run 10 times, in order to get more variety in the possible ways that the compiler was executing the instructions.

Conformity to Model-View-Controller

We will now discuss how we conformed to the MVC pattern.

The **model** comprises the classes `WordDictionary`, the array of `WordRecords` and the `Score` class. These classes clearly stored the data (i.e. the words to be used, the properties of the words and the various performance statistics, respectively). Furthermore, they manage the logic of how to update or use this data. Here is one example from each class to illustrate this: (1) in the `Score` class, there is the method

`missedWord()` which increments the `missedWords` counter. (2) In the `WordRecords` class, there is the method `drop(int inc)` which you can call to increase the `y` value of the `WordRecord` object by the `inc` value. (3) In the `WordDictionary` class, you can call the `getNewWord()` method to extract a new word from the dictionary of words.

The **view** comprises the GUIs (contained in the `WordApp` class and the `EndGUI` class) and the `WordPanel` class, which forms part of the `WordApp` GUI. This view is the representation of the model for the user. Clearly the two GUIs are representing the data contained in the model. Here are some examples: (1) The words being displayed on the GUI represent a subset of the words in the `WordDictionary` object which are stored in `WordRecord` objects. (2) The position of the words and the speed at which they fall is a representation of the `x` and `fallingSpeed` attributes of the `WordRecord` class. (3) The various score statistics displayed in the GUIs is a representation of the fields of the `Score` object.

The **controller** comprises the threads created for each "column" of falling words in the display (represented by `WordThread` objects). The controller is receiving input from the GUI and controlling the subsequent actions on the model and view. For example, when the user enters a word into the text box, the action event triggers a call to the method `testWord(String word)` contained in the `WordPanel` class. This method then calls the `match(String testWord)` method in each thread and tests for equality. If equality is achieved, the `caughtWords` counter is incremented. Thus, the controller (the thread class) is ultimately receiving the input from the GUI, testing the input and deciding how to update the model accordingly.

Their interactions can be summarized as follows: The user inputs a word into the `WordApp` GUI. The thread class then tests for equality based on the word entered and the current words on the screen (via the `WordPanel` class). The thread class then updates the model (i.e. the `Score` fields and extracting a new word from `WordDictionary`) and the text fields containing these models are updated to reflect these changes (by the thread class). The `WordPanel` object then reflects these changes when `repaint()` is called on it (by itself - using a `Swing.Timer` object).

Additional Features/Extensions

We have included three additional features:

1. General increasing trend in `fallingSpeed` for the `WordRecord` objects.

We used the following line of code:

```
fallingSpeed=(int)((WordApp.score.getTotal()+1) * Math.random() * (maxWait-minWait)+minWait);
```

to increase the general falling speed of each word as the total number of words (caught or missed) increases. Thus, not only are the falling speeds arbitrary, but they increase linearly as you progress through the game, steadily making the users' task harder.

2. Sound effects.

To further improve the game, we used sound effects. Sound effects are triggered on the opening of the game, when a word is caught or missed, and when the user enters a word that is not currently displayed on the screen. This helps users keep track of how they are doing, and increases the "fun" associated with the game.

3. Final results GUI

We created a new GUI to display the final results from the game.