Comfort Twala - TWLCOM001

CSC2002S: Assignment 2

Concurrency Programming: Falling Words

Introduction:

Concurrency allows programs to deal with a lot of tasks at once. Unlike parallel programming, which divides workload, concurrency can do multiple different tasks at one. For a lot of games, a lot of tasks need to be done at the same time and can't be achieved in a linear sequential manner. Concurrency is key here. It has its disadvantages like race conditions and thread issues, so needs to be implemented properly.

Class Design:

- Score
  - This class deals with everything related to the game score. From updating it, to resetting it.
  - To ensure encapsulation, get methods are created and used to get score from external classes and updating methods are added to modify score accordingly.
- WordDictionary
  - This is the data/information class. Uses a text file and populates the array of words (theDict) that are required by the external class.
  - To prevent already caught words from showing up again (Leading to possible game finishing without using all words available) I added an caughtWord(String caught) method that removes caught words from the active game dictionary.
  - Since I remove already caught words from the dictionary whenever a user restarts a new game in the current session, the dictionary needs to be reset. Therefore an original array was added to store original dictionary words, and a reset method was added to repopulate theDict array used by external classes back to the initial words.
- WordRecord
  - This is the specific word data/information class. Stores and modifies everything related to the word, from changing its text, its position and even its fallingSpeed, etc.
  - To ensure smoother animation, minWait and maxWait were adjusted so that the words don't drop too quickly at a time.
  - Since - as above mentioned - when caught words are reset, they're removed from the WordDictionary dict, I added an resetCaught(String word) method that will send that word to caughtWord() method in WordDictionary, so that the new word isn't one that is already caught.

- WordPanel
  - This class is the container class for all WordRecord instances (words).
  - Since it implements Runnable, it has a run() method, which contains the code that is implemented by a Thread.
  - To have more control of my Thread speed and what it should do, in the run() method I instantiate a Timer which executes the Thread whenever the delay period elapses and performs the ActionListener (second parameter)
  - Created an ActionListener which does the animation, by dropping the words by their falling speed and also checks if it's dropped or not to reset the word and incrementing counter dropped for the missed words.
  - Added a stop method to be able to stop Timer Thread from external classes.

```
@Override
public void run() {
    timer = new Timer(250, animation);
    timer.start();
}

public void stop() {
    timer.stop();
}

/**
 * ActionListener Method for the Word panel class
 */
ActionListener animation = new ActionListener(){
    public void actionPerformed(ActionEvent e) {
        for (WordRecord word : words) {
            word.drop(word.getSpeed());
            if (word.dropped()){
                dropped++;
                word.resetWord();
            }
        }
        repaint();
    }
};
```

- WordApp
  - This is the main driver class.
  - The SetupGUI method is where the layout of the program is set up and all the components are created and where their functions are defined.

Concurrency features:

First feature that was used is synchronized methods. This is to ensure multiple threads don't access the same resource simultaneously as this will cause thread interference and create consistency problems.
Volatile variable was used since it is stored in Main Memory ensuring the threads access the same variable in memory.

The following was done to ensure thread safety:
Swing comes with its own currency features which ensure Thread safety and minimise the program from "freezing" and is always responsive no matter what.
From the Swing Concurrency features I made use of Initial threads, which execute from the initial start of application.
This is done by using SwingUtilities.invokeLater(runnable). Since animation needs to start when the start button is pressed, inside the startB ActionListener I call SwingUtilities.invokeLater(w) which starts the WordPanel Thread since it implements Runnable.
Another SwingUtilities.invokeLater() was used to create a runnable that will keep track of dropped words, update score, play 'missed' sound when such happens. It is also responsible for keeping track of the missed score and ending the game whenever the player loses.
Thread synchronization wasn't necessary as Threads were coded in a way that they each focus on their own respectful actions that don't interfere with the other thread.
As mentioned, since Threads were coded in a way that they aren't dependent on each other, liveness wasn't really implemented as there is no need for it in this situation.
To also prevent deadlock caused by Threads waiting for each other to release the lock, threads were coded in a way that whichever lock is required, it is available.

System validation:

To validate if the system works as expected, I tested all possible outcomes by replaying the game multiple times to see if the tested outcome is achieved. If it was not, I would try to debug and retest again until I'm satisfied by the result.
Since I would not be able to spot, and test all possible outcomes, I compiled an executable and sent it out to some of my friends to find any defects or issues. Most of them stumbled across subtle errors when they played with high numbers, etc. which caused race conditions. I fine tuned the code and ensured race conditions were eradicated by making use of more synchronized methods and shared features.

Model-View-Control:

Since the Swing framework loosely follows the MVC architecture, my design also doesn't fully comply with the MVC pattern.

MVC states that the Model (Component's data), View (Visual representation of the Component's data) and Controller (user input/computation) should be done seperately. In Swing the Model is separate, while the View and Controller are implemented together in the User Interface elements. Due to this my design is as follows:

- Model
    - WordDictionary
    - WordRecord
    - Score

  These classes store the data.
- View-and -Controller
    - WordApp
    - WordPanel

  These classes are responsible for the user interface and how the components interact with each other to update the View.

Additional Features:

To improve on the User experience and make the game more game-like and enjoyable I added the following features:

- Game Sounds that provide feedback to the user.
- Game ends and the user loses whenever the missed counter is equal to the total number of words.
- Rules dialog box in the beginning to make the game easier to understand.
- Message Dialog displaying game stats whenever the game ends.