Concurrent Programming: Falling Words

Oyama C. Plati[†]
CSC2002S Class of 2019
University of Cape Town
South Africa
[†]PLTOYA001

I. Description of Classes and Modifications

In this project we follow the MVC design pattern. The model comprises of the *WordDictinary*, *WordRecord* and *Score* classes. The view consists of the *WordPanel* and *WordApp* classes. For the controller we implemented threads and managed thread safety for the animation.

A. Score

This shared object is used to keep the count for missed words, caught words and the game score.

```
private AtomicInteger missedWords;
private AtomicInteger caughtWords;
private AtomicInteger gameScore;
```

We used the thread safe Atomic variable class to prevent bad interleaving and protect the class state.

Having declared the state variables as Atomic integers does not prevent our methods from being preempted by other threads in compounds methods such as caughtWord() and producing unexpected results, such problems are called race conditions.

```
public synchronized void caughtWord(int length) {
        caughtWords.getAndIncrement();
        gameScore.getAndAdd(length);
}
```

We synchronized the method to prevent race conditions so that at most one thread has access and updates the class states.

B. WordRecord

```
WordRecord(WordRecord original) {
    if (original == null) {
        System.out.println("Fatal error.");
        System.exit(0);
    }
    text = original.text;
    x = original.x;
    y = original.y;
    maxY = original.maxY;
    dropped = original.dropped;
    fallingSpeed = original.fallingSpeed;
}
```

A copy constructor is used to implement array copy in Java.

```
public synchronized String toString() {
return ("text = " + text + "\n" +
        " x = " + x + "\n" +
        " y = " + y + "\n" +
        " maxY = " + maxY + "\n" +
        " dropped = " + dropped + "\n" +
        " fallingSpeed = " + fallingSpeed + "\n");
}
```

We added a toString() method for error detection and debugging while developing the application.

C. WordPanel

We implemented the Runnable interface and inside the run method we performed the animation using threading.

In the WordPanel constructor we initialise the WordRecord array field using a WordRecord copy constructor.

```
this.words = new WordRecord[wds.length];
for (int i = 0; i < wds.length; i++) {
        this.words[i] = new WordRecord(wds[i]);
}
```

We initialise the WordPanel thread using the start() method

```
public void start() {
    done = false;
    reset();
    if (animator == null) {
        animator = new Thread(this);
        animator.start();
    }
}
```

Inside the method we call the reset() method to set the word positions and score to zero. The start() method is called when the user pressed the Start button to begin the game.

We stop the WordPanel thread using the stop() method and clear the whole animation screen using repaint() method.

```
public void stop() {
    done = true;
    if (animator != null) {
        animator.stop();
        animator = null;
    }
    repaint();
}
```

The stop() method is called when the user pressed the End button to pause the game.

Animation will take place inside the thread 1. The run method is only called once. That is why we have a loop which ends when we reach the specified maximum number of words using endGame() method or the user pressed the End button.

```
public boolean endGame () {
    if (WordApp.score.getTotal() >= WordApp.totalWords) {
        return false;
    }
    else {
        return true;
    }
}
```

From the run() method, we call the UpdateThread thread instance for each word to move the word positions on a separate thread and we call the repaint method to perform the animation. Once the game loop is done we display an appropriate message on the user's performance.

D. UpdateThread

We created a separate thread for each word falling on the screen. The work done by each thread is to manage the falling speed of a word and moving the word positions.

```
public void run() {
    while(WordApp.done || endGame()) {
        UpdateThread[] updateThreads = new UpdateThread[noWords];
        for (int i = 0; i < noWords; i++) {
            updateThreads[i] = new UpdateThread(words[i], maxY);
            updateThreads[i].start();
            try {
                updateThreads[i].join();
            }
            catch (InterruptedException e1) {
                e1.printStackTrace();
            }
            repaint();
        }
    }

    System.out.println("Game Over!");
    System.out.println("Caught: " + WordApp.score.getCaught());
    System.out.println("Missed:" + WordApp.score.getMissed());
    System.out.println("Score:" + WordApp.score.getScore());
    System.exit(0);
}
```

Listing 1. Modification made in run() method in WordPanel

```
public void run () {
        if (!wordRecord.dropped()) {
        CaughtThread caughtThread = new
            CaughtThread (wordRecord, WordApp.textFromUser);
        caughtThread.start();

        try {
            caughtThread.join();
        } catch (InterruptedException e2) {
            e2.printStackTrace();
        }

                try {
                        Thread.sleep(100);
                }
                catch (InterruptedException e0) {
                  System.out.println(e0);
                }
        }
    else {
        wordRecord.resetWord();
        WordApp.score.missedWord();
        WordApp.missed.setText("Missed:" +
            WordApp.score.getMissed() + "    ");
    }
}
```

Listing 2. Implementation of run() method in UpdateThread

In listing 2 the run method we set up each falling word as separate thread. alternate between sleeping with the sleep() method and incrementing the position of the word in the CaughtThread thread.

E. Caughthread

We implement the action of adding and removing words as necessary. Also updating the counters and score.

In listing 3 in the run() method we increment the position of the word if it does not match the user entry. Otherwise, we update the score.

F. WordApp

We made all the WordApp components static fields to access and change their value from different classes.

```
static JFrame frame;
static JPanel graphic, txt, b;
static JLabel caught, missed, scr;
static final JTextField textEntry = new JTextField("", 20);
static JButton startB, endB, quitB;
static String textFromUser = "";
```

We implemented the action listener to handle the event generated when the Start button is pressed.

```
public void run () {
    if (!word.matchWord(entry)){
        word.drop(word.getSpeed()/100);
    }
    else {
        WordApp.score.caughtWord(entry.length());
        WordApp.caught.setText("Caught: " +
            WordApp.score.getCaught() + "    ");
        WordApp.scr.setText("Score:" +
            WordApp.score.getScore()+ "    ");
    }
}
```

Listing 3. Implementation of run() method in CaughtThread

```
startB = new JButton("Start");
startB.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        done = false;
        w.start();
        caught.setText("Caught: " + score.getCaught() + "    ");
        missed.setText("Missed:" + score.getMissed()+ "    ");
        scr.setText("Score:" + score.getScore()+ "    ");
        textEntry.requestFocus();
    }
});
```

In the actionPerformed() method we call the WordPanel thread for redrawing the animation using the start() method.

We implemented the action listener to handle the event generated when the End button is pressed.

```
endB = new JButton("End");
endB.addActionListener(new ActionListener()
{
  public void actionPerformed(ActionEvent e)
  {
   done = true;
          w.stop();
  }
});
```

In the actionPerformed() method we call the WordPanel thread to stop animation using the stop() method.

We implemented the Quit button and action listener to handle the event generated when the Quit button is pressed.

```
quitB = new JButton("Quit");
quitB.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e) {
        System.exit(0);
    }
});
b.add(quitB);
```

The button has given the label ”Quit”. In the actionPerformed() method we exit the current program using System.exit(0).

II. Methodology

A. Thread Safety

Having the Score object declared as AtomicInteger does not prevent the thread from being preempted in the middle of method execution (if it’s not synchronized). Therefore we synchronized all the getters and setters and compound methods in the class declaration.

B. Thread synchronization

Multiple thread will be making changes to the model. We ensured that the model was thread safe using synchronization and locks.

C. Liveness

We ensured all activity is on the event dispatch thread and that there was no other time-consuming activity executed on

this thread, repaint() was only done in the WordPanel thread. This ensured that the Updatethread and CaughtThread do execute and result in the GUI being unresponsive.

D. Deadlock

Since events are processed in a separate thread from the main thread, we did not modify or query the GUI for an application from the main thread once the GUI had been displayed or was ready to be displayed. Otherwise, this may have resulted in a deadlock.

E. MVC

We followed the model-view-controller (MVC) pattern. We let all threads update a (thread safe) model. Whenever there was an update of the model, we invoked repaint. The repaint() method scheduled the UI-thread to call the proper paint-methods. The paint-method read the state of the model, and drew the component accordingly.