Introduction

This report will consist of the following sections. A description of the classes of the program and modifications made to the given classes. A description of all the Java concurrency features used as well as the reason why they were necessary. Explanation of how the code ensures thread safety, thread synchronization, liveness and deadlock. Validation of the program and how error were tested for and finally an explanation of how the design conforms to the Model-View-Controller pattern.

Description of Classes

The program makes use of the following classes: Flow.java(Given)
FlowPanel.java(Given)
Terrain.java(Given)
Water.java

Created Class

Water.java :	This class is holds the details of the water contained in a grid position. It holds the grid's elevation, number of water units and waterdepth. It has accessor and mutable methods for setting and getting the water units(getWaterUnits, addWaterUnits, addWaterUnit and removeWaterUnit) and waterSurface.

Modication of Given Classes

Flow.java : This class is the main class. It creates the Graphical user Interface(GUI) and creates and initialises the terrain of the data. All of the modifications are made in the SetupGUI() method, or to be used by it. These include creating and adding buttons(Play, Pause, Reset and End), and a JLabel called Counter to the Main Panel of the GUI.The Play button starts the simulation of the waterflow on the GUI by setting the pause Atomic boolean to false. The Pause button pauses the simulation, by setting the pause Atomic Boolean to true. The Reset button removes all water on the terrain, by setting the unit values in all the water classes in the waters array to 0 and the End button closes the GUI. The Counter shows the timestep of the simulation.
An MouseListener is included to pick up user input, when the user wants to add water to a grid position. It finds the location of the grid and adds 5 units of water to that position as well as surrounding positions in a reach of 2 in each direction. This is done by the initialClick() method.

Terrain.java : This class is responsible for creating the terrain from the input data file. I added an extra BufferedImage for the Water Layer.

FlowPanel : This class is responsible for the simulation of water flow. It makes use of 4 threads. It uses the following shared variables:

- pause : Boolean used to set the state(pause or play) of the simulation.
- waters : An array containing the water class of all the grid points.
- land : The Terrain class.
- Counter : Used to show the timestep of the simulation.
- hi and lo : Used to set the bounds of the grid points a thread is responsible for.
- CounterLabel : The Jlabel used to show the Counter on the user Interface.
- end : Used to end the thread execution.

This class uses the following methods:

- run : This method is called to initialise the execution of the 4 Threads to run the simulation.
- lowestNeighbour : This finds the neigbouring position with the smallest water surface

- WaterFlow : This is the main method used to transfer water from the current position to its lowest neighbour.
- paint : This is used by the paintLoop.
- paintLoop : This is used after WaterFlow to paint changes to the water Image.
- pauseLoop : Used to temporarily pause the simulation when the pause button is clicked.
- setPause : Used by the Flow class to control the state of the simulation.
- refresh : This is used to render changes to the water image.

<u>Java Concurrency features</u>

I made use of the following java concurrency features- an atomicBoolean, and the synchronize lock class. The AtomicBoolean is the pause boolean in the FlowPanel which is used to pause/play the threads, and since only want one thread is required to change that value at a time, an atomicBoolean is used to prevent data races.
I also used the synchronize lock class on both variables, methods and code blocks within methods. The synchronize locks are used on shared variables that don't have an Atomic type, to prevent data races. These variables are those that are shared by the thread class. On methods, the synchronize locks are used to prevent 2 methods from the same water class running at the same time, this prevents data races in the Water class. For code blocks in methods, synchronize lock used to prevent bad interleavings in the WaterFlow methods in the FlowPanel class.

<u>Thread Safety</u>

My program has ensured that shared and mutable data are protected for data races, and bad interleavings. Concurrency is only necessary for mutable shared data by multiple threads. Since most threads working on the array have their own portion of the water array, there are limited opportunities for data races to occur. The is one locations for a potential data race which occurs if a neighbour of a position falls in the jurisdiction of another thread. The 2 locations for which this occurs is in the lowestNeighbour() method method and then later in the waterFlow() method where we add to the neighbour units and remove from the current position.These are locked using synchronized locks.

The water class has specific methods for incrementing and decrementing the number of units in the class, and they are incremented and decremented in synchronized blocks of code thus preventing bad interleavings. The other shared variable is in the pause AtomicBoolean, shared by the 4 running threads. The AtomicBoolean is perfect for this variable because only 1 thread is required to change the state of the variable. Besides these situations, the other variables are only mutable by the first thread, such as the Counter, CounterLabel and end variables, so no concurrency is required for them.

There is also access to the water class by swing elements. This is all done once at the end of the timestep, so they don't coincide with the execution of the 4 simulation threads – thus the water values do not change. When painting the changes of the previous timestep to the waterImage, the swing elements use the getWaterUnits method, however they do not change the data and the accessor methods of the water class are locked using a synchronized lock so there will be no concurrency issues.

<u>Thread Synchronization</u>

To ensure synchronization, the run() method in the FlowPanel class has 2 parts and I used an if statement to separate them. The if statement does the movement and painting of water for its portion of the water array, while the else statement would be responsible for starting each timestep. The else statement would create and initialise 4 threads for each timestep, each responsible for a certain portion of the water array. The threads are synchronized in that before moving to the next timestep, the finished threads wait for the others to finish because of the join() method used on all the threads, afterwhich the Counter is incremented by one, and the changes are rendered to the image with the repaint() method.

Liveness and Deadlock

Deadlock refers to the blocking of thread execution as a result of incorrect concurrency. For my program, there is unlikely to be a deadlock. There is a very small region for which deadlock can occur, and the program uses locks very spatingly. And the only code block(in the waterFlow()) where multiple locks are required, is protected with a lock and since the accessor and mutator methods are dependent on only one class, there is no scenario where multiple locks are required or a single method, so the locks will eventually be released or other threads.

Validation of program

Validation of the system is done through thorough debugging cases, where each methods is tested to ensure that multiple access to shared resources is secure.

Conformation of program to the Model-View-Controller

The Model-View-Controller consists of 3 component- Model, View and controller.
The Model contains only the pure applicational data, and has no Idea of how to use them to present information to the user.
The View presents the model's data to the user.
The Controller exists between the View and the model, it listens to events triggered by the view and executes an appropriate reaction through the method and returns the results to the View.

For my program, the Model refers to the Terrain and Water classes. These obtain and store the data of the terrain and water. The View refers to the Graphical User Interface made in the Flow class, with buttons capable of trigerring events(adding water, start, pause, reset and end) and shows the simulation to the user. The Controller refers to the FlowPanel class. It uses threads and the various methods in the class to move water across the terrain and to paint the changes to the View.

Conclusion
The program is shows fluid conservation and allows for the smooth movement of water across the terrain. This is accomplished through the use of multithreading and sufficient concurency which allows for the protection of shared and mutable data.