**PROJECT B EXPLANATION:**

In this project, the task was to create two cooperating threads (A and B). They each have three function calls as shown in the Figure 1 below:

Figure 1

The requirements were then specified in more detail that it was to be done using the this Figure 2 below:

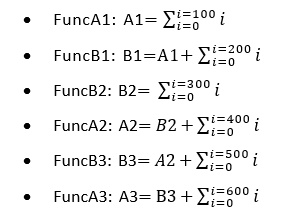


Figure 2

A picture containing font, white, typography, calligraphy

Description automatically generated

Figure 3

The appendix for summation was given as:

The code needed additional synchronization code and no Active wait or Thread sleep methods were not to be used during implementation.

The following code below provides an explanation of how synchronization is implemented in the code:

The *‘Lock’* object was created. This will be used to monitor for the synchronization.

**public** **static** **void** main(String[] args) {

Object lock = **new** Object();

Thread ThreadA = **new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

**synchronized** (lock) {

A1 = Utility.sum(100);

lock.notify(); // A1 then proceeds to B1 in ThreadB after it is finished calculating

**try** {

lock.wait(); // Here it is waiting for B2 to finish calculating from ThreadB in order for A2 to be next

A2 = B2 + Utility.sum(400);

lock.notify(); // It then notifies ThreadB to switch the order from A2 to B3

lock.wait(); // Here A3 is waiting for B3 to finish calculating in order to finish the sequence as shown in the figure

A3 = B3 + Utility.sum(600);

lock.notify(); // Checks if there is any other task left at ThreadB else it will continue the program

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

});

Thread ThreadB = **new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

**synchronized** (lock) {

**try** {

lock.wait(); // Here, B1 is waiting for A1 to finish calculating from ThreadA

B1 = A1 + Utility.sum(200);

// At this line, the figure displays that it goes from B1 to B2 so no notify or wait is needed

B2 = Utility.sum(300);

lock.notify(); // The program then switches back to ThreadA to calculate A2.

lock.wait(); // Wait for ThreadA to finish A2 calculation

B3 = A2 + Utility.sum(500);

lock.notify(); // Notifies A3 to proceed its calculation (as the figure 2.1 says)

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

} });

The two Threads above are dependent on each other. As you can see, they are synchronized on the *‘Lock’* object and that it is using the *‘***synchronized***’* variable. This is done to ensure that only one thread can execute the block at a time.

The synchronization code is set so that it compiles the code in the order of which Figure 1 had given:

Firstly, *‘A1’* starts summing from *‘*ThreadA*’*. After it is done, the thread calls *‘*lock.notify()*’*, which then notifies any waiting thread in the program (which in this case is *‘*ThreadB*’*) after that, ‘ThreadA’ calls ‘lock.wait()’ where the lock is released and waits until the next thread (which is B) notifies it has done compiling its part.

This thread then proceeds compiling the *‘B1’* function which was defined in Figure 2 (‘*FuncB1’*). It then proceeds compiling the *‘B2’* function (note there is no notify or wait after B1; This is because Figure 1 shows that it goes straight from B1 to B2 in the same thread, so lock is not needed). The B2 then calls *‘*lock.notify()*’* which notifies *‘*ThreadA*’*. ‘ThreadB’ then call the ‘lock.wait()’ and waits until the other thread notifies.

*‘A2’* is then compiled with its function from Figure 2 (FuncA2). This same process is done as mentioned before, however since Figure 1 shows that *‘A2’* goes to *‘B3’*. It notifies ‘ThreadB’ and waits till *‘B3’* is finished. Finally, the last one is *‘A3’* where ‘ThreadB’ notifies ‘ThreadA’ and that it is finished calculating *‘B3’* therefore doing the final calculation result for *‘A3’* as mentioned in Figure 2. ‘ThreadA’ must notify ‘ThreadB’ that it has finished the calculation so that after that the code continues to its normal operation.

Moreover, the program above is just theoretically explained since these codes did not start compiling YET as they have not been started. The code below will show how it will run.

ThreadA.start();

ThreadB.start();

Here shows that ‘ThreadA.start()’ and ‘ThreadB.start()’ has a start method. These methods being their execution which calculates all these variables.

**try** {

ThreadA.join(); // waits for both of the threads to finish before compiling the next set of codes below

ThreadB.join(); // this means that the code is technically paused until the Threads A and B are finished (which then continues after)

} **catch** (InterruptedException e) {

e.printStackTrace();

}

This code above basically means that the join method waits for the completion of both threads before the program continues back to the next line.

**class** Utility **extends** ProjB {

// the sum calculation as shown in the appendix (word file)

**public** **static** **int** sum(**int** n) {

**return** n \* (n + 1) / 2;

}

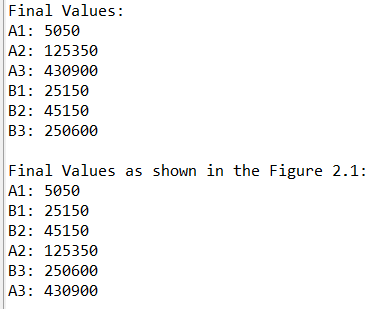
}

A dedicated class was created for the sum function from the appendix (Figure 3) so that it can be used in any class within the same project when called.

After this the results for each function (A1 to B3) are displayed below, one is just in alphabetical order and the other is the same as shown in Figure 1 (where A3 is the final answer):

//Printf statements in order:

System.out.println("Final Values:");

 System.out.println("A1: " + A1);

System.out.println("A2: " + A2);

System.out.println("A3: " + A3);

System.out.println("B1: " + B1);

System.out.println("B2: " + B2);

System.out.println("B3: " + B3);

//Printf statements in the order of the FIGURE 2.1

System.out.println("\nFinal Values as shown in the Figure 2.1:");

System.out.println("A1: " + A1);

System.out.println("B1: " + B1);

System.out.println("B2: " + B2);

System.out.println("A2: " + A2);

System.out.println("B3: " + B3);

System.out.println("A3: " + A3);

A RAW CODE IS ALSO COPIED INTO THE OTHER WORD FILE INCASE THE MAIN JAVA FILE DOES NOT WORK…