Table of Contents

Assignment 1:

Q:1 Describe the followings:

Process?

Process are execution environment provided by operating system that has its own set of resources such as memory

Thread? Provide five examples, Advantages?

Threads are processes that live within the process and shares their resources.
Threads are also called as lightweight processes.

Examples

1) Microsoft word or any other text editors where we are writing and another thread is used to check the spelling
2) Integrated development environment (IDE such as IntelliJ or Android Studio) which runs the auto complete or suggest syntax feature in separate thread.
3) Web Browser where we can work simultaneously with multiple tabs and we can download multiple things concurrently.
4) Video Players which process the multiple frame and audio script using multiple threads

5) Games are another example of multithreading where one thread using for background music another is used for visuals and other stuff.

Advantages:

Improve development and Maintenance cost

Improve performance of complex application

Using Multithreading we can increase the throughput on single process system.

Threads are very useful for GUI application to improve the responsiveness of the user interface.

Task?

Task is set of work that we can perform using single or multiple thread

Yield?

Yield allow us to temporarily pause the execution of current thread and allow other thread to proceed.

States of Java Threads

1) NEW: A thread that has not yet started is in this state.

2) RUNNABLE: A thread executing in the Java virtual machine is in this state.
3) BLOCKED: A thread that is blocked waiting for a monitor lock is in this state.
4) WAITING: A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
5) TIMED_WAITING: A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.
6) TERMINATED: A thread that has exited is in this state.

Q:1.2 What is the difference between Concurrency and Parallelism?

Concurrency :

- Concurrency means running multiple task at the same time and share same resources

- For example if we have different task to perform such as Task1, Task2 and so on we can perform these task concurrently by assigning each task to different thread.

Parallelism

- Splits up task in subtask and execute all task parallelly in different core of CPU.

- Parallelism requires hardware with multiple processing units.

- In single core CPU you might get concurrency not parallelism.

- For example we want to perform big task assuming that we can not perform that in single core of CPU then we can splits up this task into subtask and let run each individual task in different core of the CPU.

Q:1.3 Why Multi-Threading is important?

- Multithreading **increase the Responsiveness** of application. Single application will use the multiple thread to accomplish various task.

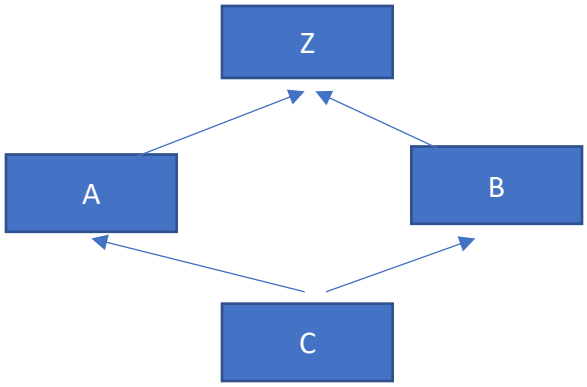
- Multithreading **improve the throughput** of the applications .

- **Minimized system resource usage.** Threads impose minimal impact on system resources. Threads require less overhead to create, maintain, and manage than a traditional process.

Q:1.4 What are similarities and differences between Java and C++

- Both the languages support **Object oriented programming.**
- Java **compiler compile code in bytecode and JVM convert that code to native machine code using interpreter**.
- While C++ compiler **compile and convert the source code into machine code**.

- C++ allows **multiple inheritance using concrete class using virtual inheritance while java support multiple inheritance using interface**.
- Java supports **interfaces and packages while C++ doesn't**.
- C++ is platform dependent while java is platform independent.



Diamond problem in C++

Q:1.5 What is diamond problem in C++. JVM, JRE, explain what they do?

- C++ supports the multiple inheritance using concrete classes.

-Consider the following diagram Let's assume that class A and B inherits the method of Class Z and class C inherits the class A and B.
- Class **A and B override the same method of class Z with different Implementation**. Now when class C uses the same method compiler did not understand which implementation it should use. This problem is known as classic diamond problem.

JVM, JRE, explain what they do?

- JRE stands for **Java runtime Environment**
- Minimum Environment requires to run java application.
- JRE Contains **java virtual machine and deployment tools**

- JVM stands for **Java virtual Environment**
- JVM interpret the **compile code into native machine language which can be understood by OS**. and underlying hardware platform.
- It resides **inside the RAM**

Q:1.6 What is a Class Loader?

- Class loader helps to **bring the class file on to RAM also known as Dynamic class loading** functionality.
- It's **load, link and initialize the class file when it refers to class for the for the first time at runtime.**
- There are three class loaders available in loading task

→ Bootstrap class loader
→ Extension Class loader
→ Application Class Loader

*Apart from **these three major class loaders we can also defined user defined class loader as well.**

Q:1.7 Java Object class.

- Java object class is the parent class of all the classes (Root of the class hierarchy) in java. Aka Topmost class in java.
- It's helpful when you don't know type of the object.

Q:2 Write a Java program to define a Student class with instance variables name, id, homework, midterm, and final. Name is a string whereas others are all integers. Add a static variable nextId, which is an integer and statically initialized to 1. Have some overLoaded constructors. In each constructor, id should be assigned to the next available id given by nextId. The default constructor should set the name of the student object to "Student_X" where X is the next id. Add calculateGrade() method which returns a string for the letter grade of the student, like "A", "B", "C", "D" or "F", based on the overall score. Overall score should be calculated as (50% homework, 30% midterm + 20% final).

Write TestDriver to test your program. It should create 20 student objects with default constructor and invoke the setter methods for homework, midterm, and final with random numbers ranging from 70 to 100 inclusive. Then it should print the student information via the toString() method. Student information provided by toString() should include name, homework, midterm, and final and the letter grade given by the calculateGrade() method.

3. Create file data.txt and add this line: "Welcome to class CSYE712 Parallelism, Concurrency, MultiThreading, Fall 2020". Write a program that reads text line from file using BufferedReader and FileReader:

a) print it on the console

b) add it to a list

c) read it from the list and change it to upper case
d) count the number of strings
e) count the number of letters
e) add it to hashMap with key/value (string, frequency)
Note: first use HashSet to get frequencies,
f) sort the map using TreeMap
g) iterate through the map and print and provide Catch clause
for FileNotFoundException (file not found), and IOException (invalid file).

Approach :

- **Create file data.txt**
  File file = new File(name);

  FileWriter x = new FileWriter(file);
  BufferedWriter bw = new BufferedWriter(x);

  bw.write(content)

- **Read the file's each line and split it with space**
  Add each string to array list

```
List<String> al;
    al = Arrays.asList(m.readFile(fileName).trim().split(" "));
```

- **Read it from list and change cases of string to upper case**

```
al.forEach(st -> System.out.println((st.toUpperCase())));
```

- **Count the number of letters**

```
int noOfLetters =
al.stream().mapToInt(String::length).sum();
```

- **Count the string frequency using hashmap**

```
Set<String> st = new HashSet<>(al);
HashMap<String, Integer> hm = new HashMap<>();
        al.forEach(a -> {
                if (hm.containsKey(a)) {
                   hm.put(a, hm.get(a) + 1);
                } else {
                   hm.put(a, 1);
                }
            }
        );
```

- **Sort the map using tree map**

```
SortedMap<String, Integer> tm = new TreeMap<>(new
Comparator<String>() {
        @Override
        public int compare(String o1, String o2) {
           return o1.compareToIgnoreCase(o2);
```

```
    }
});
tm.putAll(hm);
```

4. Create interface I that extends interface I1 with m1() and m2() methods, and interface I2 with m3() and m4() methods. Create class A that implements interface I. The implementation of each method must print the name of method. Create a Test class that instantiates an object of class A but uses interface I as its type. The Test class must execute all the methods of class A. Compile and run the code.

5. Compile and Run this code. Explain what and how it does it Step-by-Step?

```
public class Input {
   int index;
   int[] input = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};

   public Input(){
      index = 0;
   }

   public void print(int index){
      System.out.println(input[index]);
   }
```

Synchronized method block make sure that at a time only one thread can execute this method

```
   synchronized public int getIndex(){
      if(index == 15)
         return -1;
      return index++;
   }
}
```

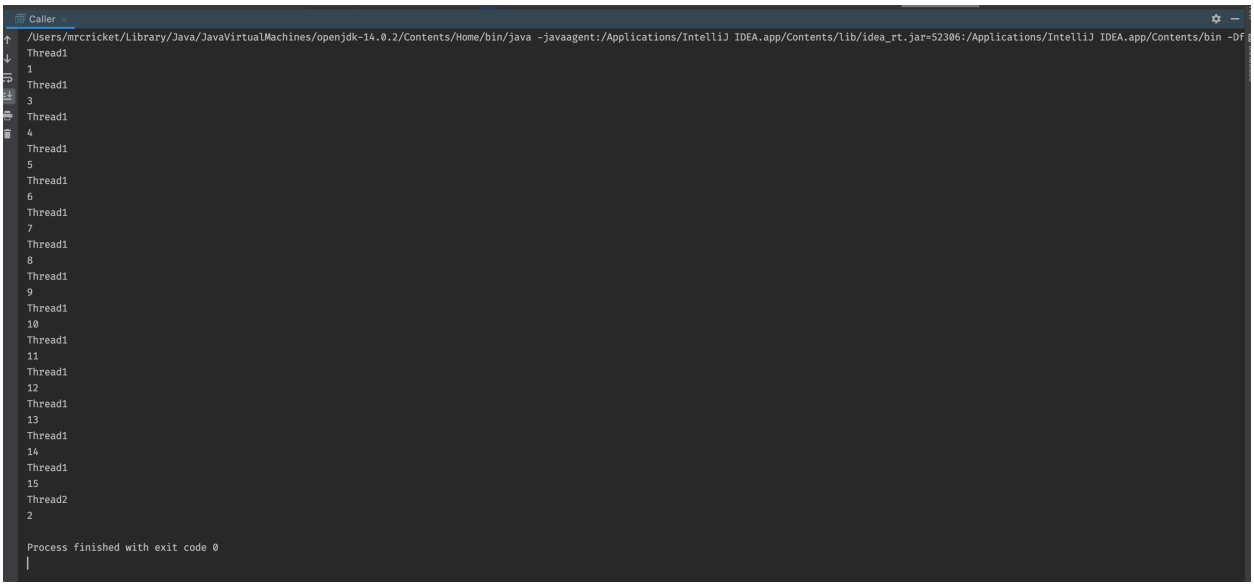
```
public class MyThread implements Runnable{

   Input ip;
   Object lock;

   public MyThread(Input ip, Object lock){
      this.ip = ip;
      this.lock = lock;
   }

   @Override
   public void run() {
      int index = -1;
      while((index=ip.getIndex())!=-1){
```

used instance lock approach to makes sure that single instance of an object can access this block

```
         synchronized(lock) {
             System.out.println(
                                Thread.currentThread().getName());

                            ip.print(index);
         }
      }
   }
}

public class Caller {
```

```
    public static void main(String[] args)
                throws InterruptedException {
        Input ip = new Input();
        Object lock = new Object();
        Thread t1 = new Thread(new MyThread(ip, lock),
"Thread1");
        Thread t2 = new Thread(new MyThread(ip, lock),
"Thread2");
        t1.start();
        t2.start();
        t1.join();
        t2.join();
    }
}
```

Output



Explanation:

- In caller class, we have created the object of the Input class and lock instance of the object class. We have passed the same lock object to thread t1 and t2 to make sure that our synchronized code block(inside the MyThread class run method) execute by on one thread at the same time.
- similar way we have passed the same input object to thread t1 and t2. We used that object to call the getIndex method which is again synchronized method so it makes sure that it is executed by a single thread at a given time.
- We can confirm this by checking the output screenshot which clearly shows that at a given time each thread gets the different value form the getIndex method.
- At the end of the main method, we have used the thread.Join() method which makes sure that the main thread not get terminated before thread t1 and t2.

6. What is the result of this Lambda function?

```
Runnable task = () -> {
   String threadName = Thread.currentThread().getName();
   System.out.println("Hello " + threadName);
};

task.run();

Thread thread = new Thread(task);
thread.start();
```

System.out.println(“Done!");

Output :

Explanation

- The given program creates the runnable task using the lambda function inside the main thread and call the run method using the main thread and also create a separate thread and pass the runnable task to that thread object using the constructor and starts that thread.
- We can observe from the output screenshot that the last print statement “Done” executes before the Thread2.start() it shows the asynchronous behavior of multiple concurrent threads. The output may vary it depends on OS how it allocates the resources to each thread

7. Consider the following programs Example1 and Example2. You have no control over the order of running Threads, the operating system manages scheduling of running threads. However the code in example1 and example2 have differences that are observable if you look closely when running the programs.

a) Compile example1 and example2
b) Run each program
d) Read the code for both examples
b) Run each program
e) Identify the different behaviors between the two programs and Explain.

Example-1:

```
class Job implements Runnable {
  private static Thread [] jobs = new Thread[4];
  private int threadID;
```

```
        public Job(int ID) {
                threadID = ID;
        }
        public void run() { do something }
        public static void main(String [] args) {
                for(int i=0; i<jobs.length; i++) {
                   jobs[i] = new Thread(new Job(i));
                   jobs[i].start();
                }
                try {
                   for(int i=0; i<jobs.length; i++) {
                      jobs[i].join();
                   }
                } catch(InterruptedException e) {
System.out.println(e); }
        }
}
```

Output:

Explanation:

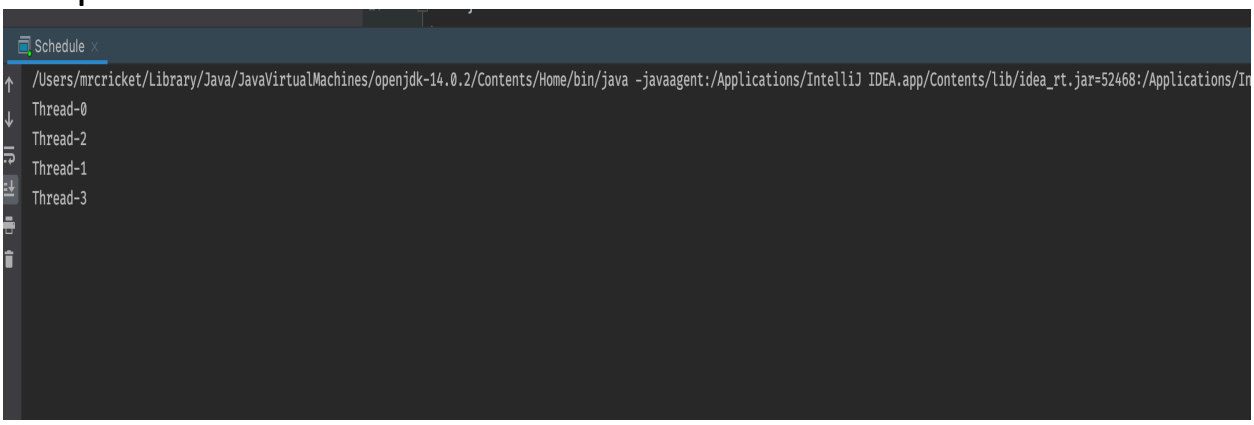
- In given program inside the main method using for loop we have create 4 different threads and assign job to each thread and start each thread using thread.start() method.
- After that we used another loop and use thread.join() method to makes sure that main thread is running until all the threads finish their jobs. Here the order of thread execution may vary depend on how OS allocate the resources

Example-2:

```
class Schedule implements Runnable {
        private static Thread [] jobs = new Thread[4];
        private int threadID;
        public Schedule(int ID) {
                threadID = ID;
        }
        public void run() { do something }
        public static void  main(String [] args) {
                int nextThread = 0;
                setPriority(Thread.MAX_PRIORITY);
                for(int i=0; i<jobs.length; i++) {
                   jobs[i] = new Thread(new Job(i));
```

```
                jobs[i].setPriority(Thread.MIN_PRIORITY);
                jobs[i].start();
            }
            try {
                for(;;) {

jobs[nextThread].setPriority(Thread.NORM_PRIORITY);
                    Thread.sleep(1000);

jobs[nextThread].setPriority(Thread.MIN_PRIORITY);
                    nextThread = (nextThread + 1) % jobs.length;
                }
            } catch(InterruptedException e) {
System.out.println(e); }
        }
}
```

Output:



Explanation:

- In given program we have created 4 different threads and set the priority of the each thread to Min_Priority (1), so now all threads have same priority so when we run the program order of threads execution are depend on OS because we have set the same priority to each thread.
- After that that we have use infinite for loop and in each iteration we are changing the priority of thread to NORM_PRIORITY(5) in order of threadID like 0,1,2,3 then again 0,1,2,3 for 1 second and after that we again change the priority of that thread to 1. We have used the infinite loops so we need to terminate the program manually.

So in conclusion if we compare the behavior of the both example we can clearly see that in **first example1 order of thread execution is depend upon how OS allocate** the resources to our program **while in second example we tried to control the order of thread execution using setPriority method.**

Assignment 2:

Q: 1 Explain

Monitor

- A monitor is mechanism to control concurrent access to an object.

- **mutual exclusion** – only one thread can execute the method at a certain point in time, using *locks*
- ***cooperation*** – the ability to make threads wait for certain conditions to be met, using *wait-set*

Lock

- A lock is kind of data which is logically part of an object's header on the heap memory.
- Each object in a JVM has this lock (or mutex) that any program can use to coordinate multi-threaded access to the object.
- If any thread want to access instance variables of that object; then thread must "own" the object's lock.
- All other threads that attempt to access the object's variables have to wait until the owning thread releases the object's lock (unset the flag).

What are Java Object Monitors?

Java associates a monitor with each object. The monitor enforces mutual exclusive access to synchronized methods invoked on the associated object. When a thread calls a

synchronized method on an object, the JVM checks the monitor for that object

- If the monitor is *unowned*, ownership is assigned to the calling thread, which is then allowed to proceed with the method call
- if the monitor is *owned* by another thread, the calling thread will be put on hold until the monitor becomes available

What is the difference between Lock and Monitor? Synchronization

- A lock is kind of data **which is logically part of an object's** header on the heap memory on the other hand A monitor is **mechanism to control concurrent acc**ess to an object using mutual exclusion (**using lock**) and cooperation
- Mutual exclusive lock we can achieve using synchronized block while cooperation can be done using wait and notify methods.

Object Level Lock

- Every object in java has a unique lock. Whenever we are using synchronized keyword, then only lock concept will come in the picture.
- If a thread wants to execute synchronized method on the given object. First, it has to get lock of that object. Once thread got the lock then it is allowed to execute any synchronized method on that object. Once method execution completes automatically thread releases the lock.

Class Level Lock

- Every class in java has a unique lock which is nothing but class level lock.
- If a thread wants to execute a static synchronized method, then thread requires class level lock. Once a thread got the class level lock, then it is allowed to execute any static synchronized method of that class.
- Once method execution completes automatically thread releases the lock

StackThread

- Thread **stack is memory area which can be accessible by only that particular thread that owns the thread stack**.
- We cannot share the data between two thread via thread stack.

- Every time we are creating new thread. Each threads create the new thread stack which store the local variable and the pointer to reference variable apart from this it contains the frame of method execution in LIFO manner.
- Thread stack is available inside the run time memory area of JVM.

2. Your program Hello references five objects, Class1, Class2, Class3, Class4, Class5. What happens when you start your program? Describe your answers in terms of Process, JVM, JRE, ClassLoader, Stack Thread, how does your program gets started?

Suppose I have following class Hello and it has reference of 5 classes class1 to class5

```
Public class Hello{

Class1 c1= new Class1();
Class2 c2= new Class2();
Class3 c3= new Class3();
Class4 c4= new Class4();
Class5 c5= new Class5();

}
```

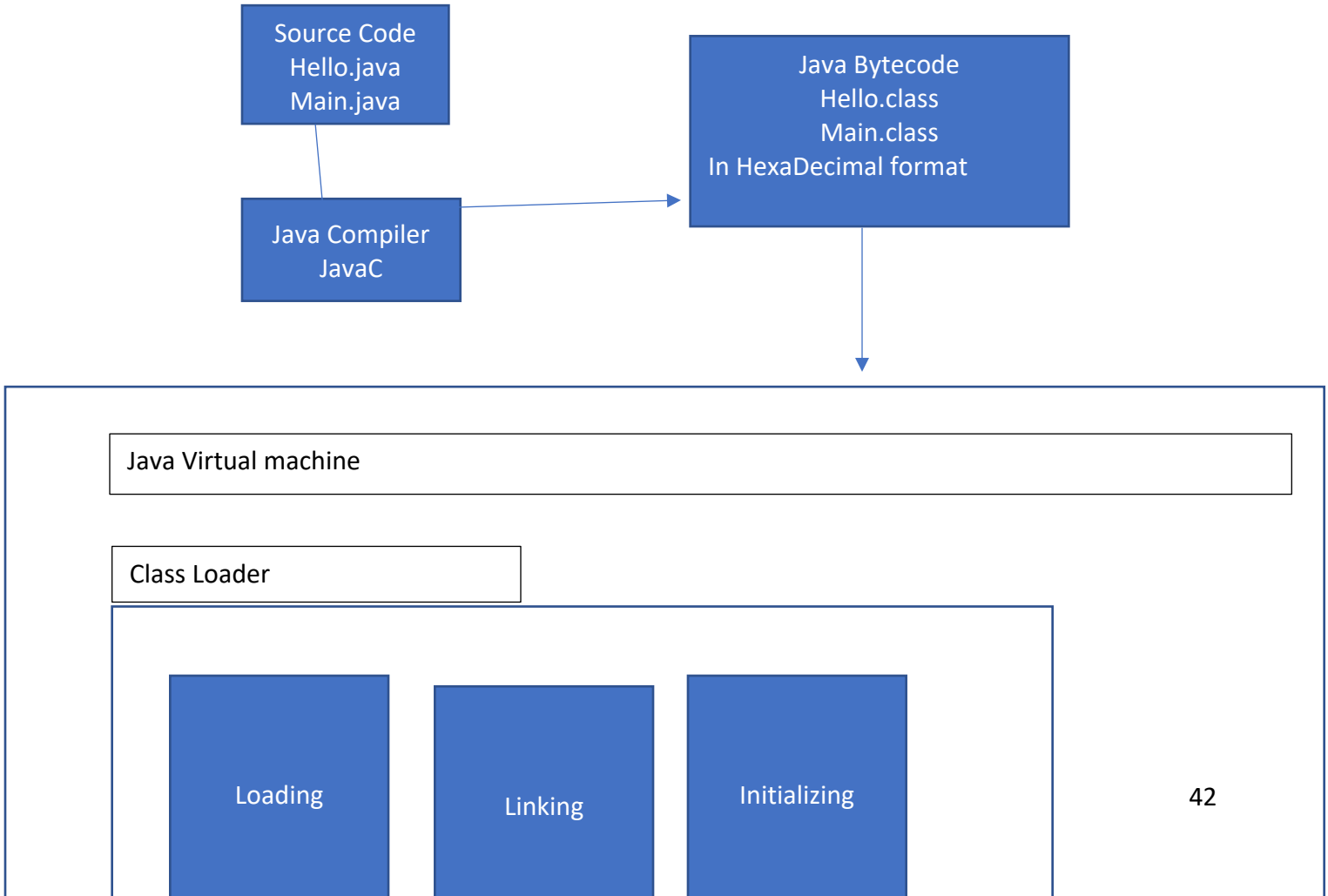
```
Public class main(){

Public static void main(String args[])
{
        Hello h = new Hello();
}

}
```

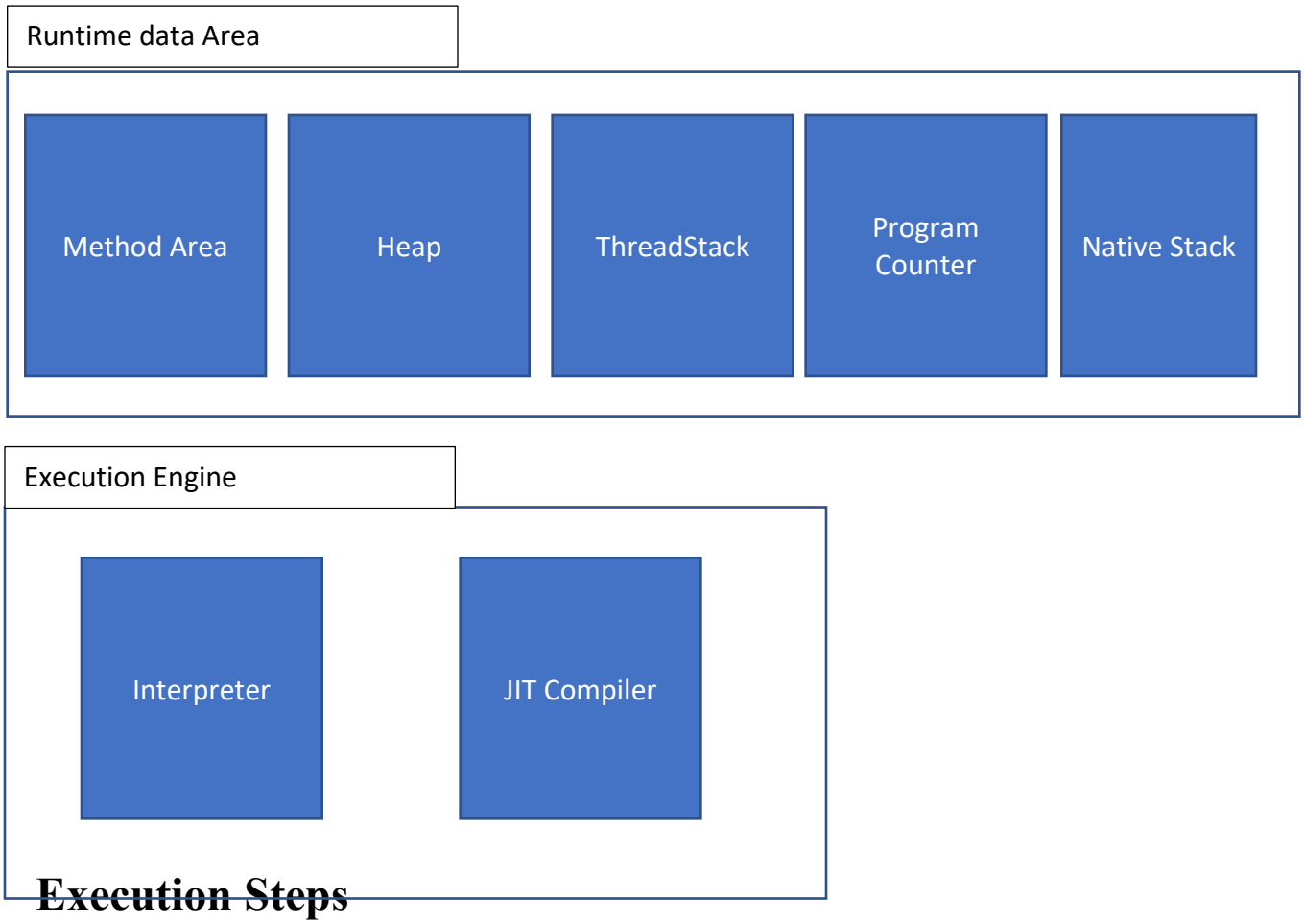
When we first compile the program java compiler convert this code to **bytecode** and create **class file** for example **Hello.class and main.class.** this bytecode acts as a platform-independent intermediary state which is portable among any JVM regardless of underlying OS and hardware architecture.

So once the code is compiled by java compiler it will go into Java runtime environment. JRE contains the Java virtual machine which has different components like

- Class Loader → Dynamic class loading Functionality
- Runtime Data area→ Memory Area where class loader generates the corresponding binary data and save the method and class metadata
- Execution Engine →Execute the instruction in byte code line by line.

Please find the following figure for better understanding


Source Code
Hello.java
Main.java
Java Compiler
JavaC
Java Bytecode
Hello.class
Main.class
In HexaDecimal format
Java Virtual machine
Class Loader
Loading
Linking
Initializing



Execution Steps

- Source code compiled into byte code in .class file in hexadecimal format by java compiler
- Then this class file goes into Java runtime environment →java virtual machine
- This class file will load into main memory by class loader which is known as dynamic class loading

- Runtime Data Areas are the memory areas assigned when the JVM program runs on the OS. In addition to reading .class files, the Class Loader subsystem generates corresponding binary data and save the following information in the Method area for each class separately.
    - Fully qualified name of the loaded class and its immediate parent class
    - Whether .class file is related to a Class/Interface/Enum
    - Modifiers, static variables, and method information etc.
- Then, for every loaded .class file, it creates exactly one object of Class to represent the file in the Heap memory as defined in java.lang package. This Class object can be used to read class level information (class name, parent name, methods, variable information, static variables etc.) later in our code.
- At the end the actual execution of the bytecode occurs. Execution Engine executes the instructions in the bytecode line-by-line by reading the data assigned to above runtime data areas using interpreter and JIT.

Q:3 Program Of student

3. Define a Student class with instance variables name, id, homework, midterm, and final. Name is a string whereas others are all integers. Also add a static variable nextId which is an integer and statically initialized to 1. In each of them, the id should be assigned to the next available id given by nextId. The default constructor should set the name of the student object to "StudentX" where X is the next id.

a) Your program is to create 25 Student Threads each to be identified with name-<Thread-nextId>. The default constructor for each thread calls a method to randomly generate grades for homework, midterm, and final-exam ranging between 70 to 100 inclusive. You need to consider 1 second wait-time between each score generation for homework, midterm, and final. Each student thread writes the grade scores to "Grades" file in this format: name, nextId, ThreadId, homework, midterm, and final. All student threads share this file and you need to protect it.

b) Create GraderThread that checks "Grades" file periodically up to 30 seconds to retrieve submitted grades by all student threads. How do you protect the file? The GraderThread reads the file (format described above) and validates name, id, threadId, scores for all 25 student threads submitted scores. For any missing grade, the student will receives zero score. The GraderThread does calculateGrade() (50% homework + 30% midterm+ 20% final) and returns a letter grade like "A", "B", "C", "D" or "F", based on the overall score.

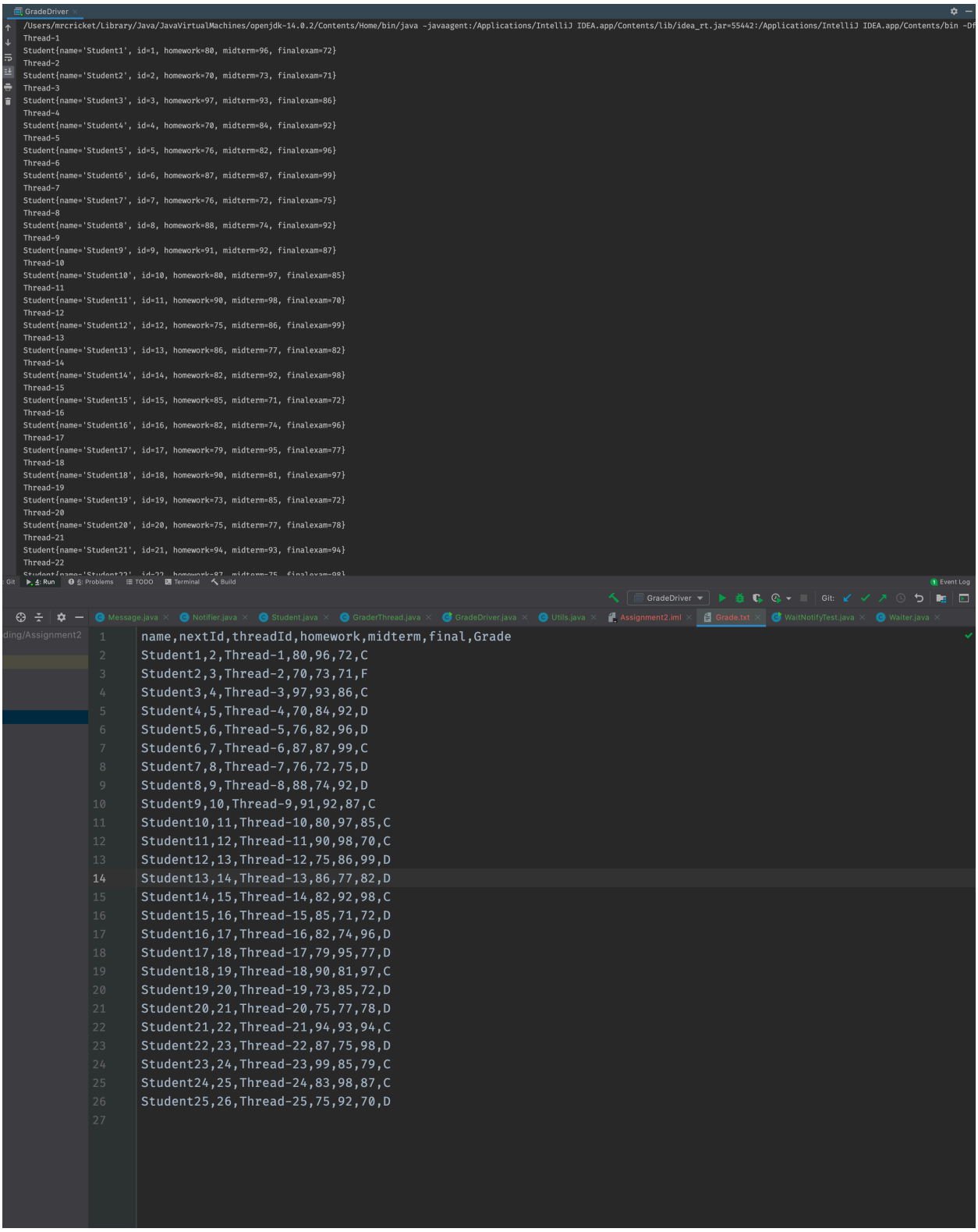
c) In your StudentThread and GraderThread, you wrote several methods to handle processing of various functions in your program. Provide JVM model for your program with details as how to handle stack threads with their methods, local variables, array, heap objects.

Note: you need to show only example data for one StudentThread. There are other single threads.

d) Create GradesDriver class to create 25 StudentThreads and GraderThread to test
your program. Compile and Run hour program.

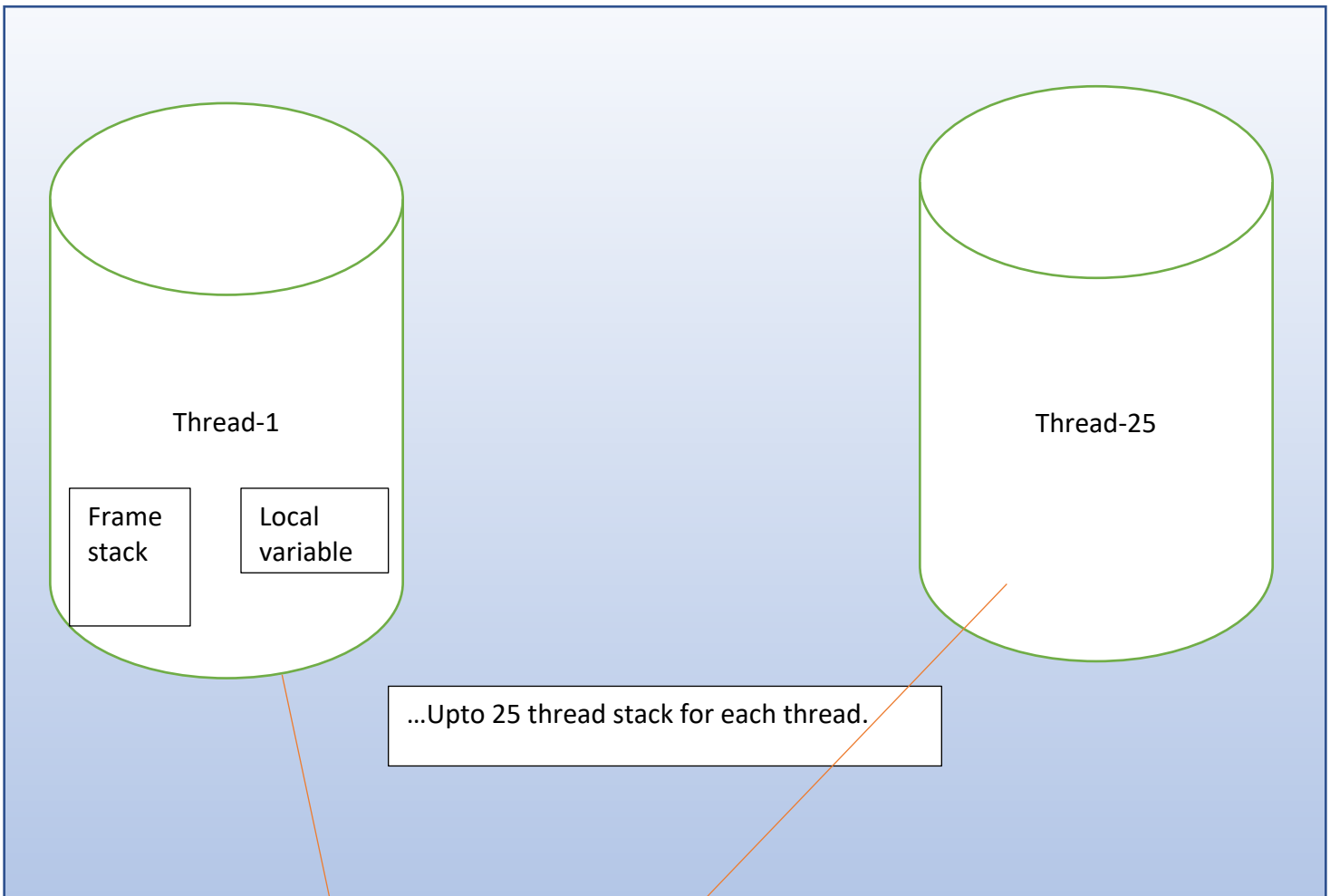
Notes: You need to consider a number of protection mechanisms to protect the "Grades" file to avoid threads over-stepping each other. You need to think about what data structures to use to hold the FinalGrade (ie: use HashMap for key/value as threadId/FinalGrade). You need to think about how to protect HashMap. You need to think about how GraderThread to notify each student with finalGrade. Do you want to update the file with new column "FinalGrade" and have student threads get the final grade by reading the file? How does that going to work? You need to think how this mechanism can work. OR do you want to do another Wait/Notify where the GraderThread will be the notifier to each student thread? All in all, you need to be creative to solve this problem.

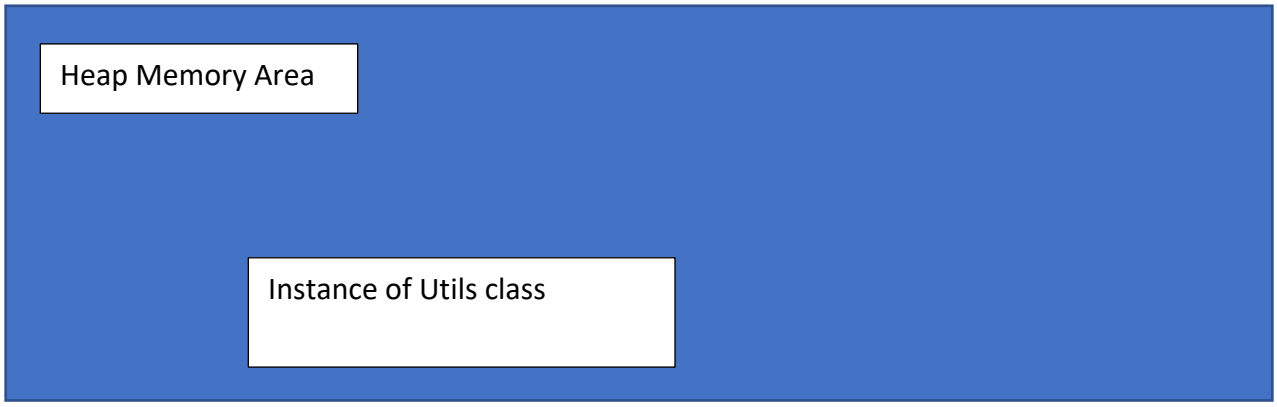
Output:


GradeDriver
/Users/mrcricket/Library/Java/JavaVirtualMachines/openjdk-14.0.2/Contents/Home/bin/java -javaagent:/Applications/IntelliJ IDEA.app/Contents/lib/idea_rt.jar=55442:/Applications/IntelliJ IDEA.app/Contents/bin -Df
Thread-1
Student{name='Student1', id=1, homework=80, midterm=96, finalexam=72}
Thread-2
Student{name='Student2', id=2, homework=70, midterm=73, finalexam=71}
Thread-3
Student{name='Student3', id=3, homework=97, midterm=93, finalexam=86}
Thread-4
Student{name='Student4', id=4, homework=70, midterm=84, finalexam=92}
Thread-5
Student{name='Student5', id=5, homework=76, midterm=82, finalexam=96}
Thread-6
Student{name='Student6', id=6, homework=87, midterm=87, finalexam=99}
Thread-7
Student{name='Student7', id=7, homework=76, midterm=72, finalexam=75}
Thread-8
Student{name='Student8', id=8, homework=88, midterm=74, finalexam=92}
Thread-9
Student{name='Student9', id=9, homework=91, midterm=92, finalexam=87}
Thread-10
Student{name='Student10', id=10, homework=80, midterm=97, finalexam=85}
Thread-11
Student{name='Student11', id=11, homework=90, midterm=98, finalexam=70}
Thread-12
Student{name='Student12', id=12, homework=75, midterm=86, finalexam=99}
Thread-13
Student{name='Student13', id=13, homework=86, midterm=77, finalexam=82}
Thread-14
Student{name='Student14', id=14, homework=82, midterm=92, finalexam=98}
Thread-15
Student{name='Student15', id=15, homework=85, midterm=71, finalexam=72}
Thread-16
Student{name='Student16', id=16, homework=82, midterm=74, finalexam=96}
Thread-17
Student{name='Student17', id=17, homework=79, midterm=95, finalexam=77}
Thread-18
Student{name='Student18', id=18, homework=90, midterm=81, finalexam=97}
Thread-19
Student{name='Student19', id=19, homework=73, midterm=85, finalexam=72}
Thread-20
Student{name='Student20', id=20, homework=75, midterm=77, finalexam=78}
Thread-21
Student{name='Student21', id=21, homework=94, midterm=93, finalexam=94}
Thread-22
Git
4: Run
6: Problems
TODO
Terminal
Build
Event Log
GradeDriver
Git:
Message.java
Notifier.java
Student.java
GraderThread.java
GradeDriver.java
Utils.java
Assignment2.iml
Grade.txt
WaitNotifyTest.java
Waiter.java
ding/Assignment2
name,nextId,threadId,homework,midterm,final,Grade
Student1,2,Thread-1,80,96,72,C
Student2,3,Thread-2,70,73,71,F
Student3,4,Thread-3,97,93,86,C
Student4,5,Thread-4,70,84,92,D
Student5,6,Thread-5,76,82,96,D
Student6,7,Thread-6,87,87,99,C
Student7,8,Thread-7,76,72,75,D
Student8,9,Thread-8,88,74,92,D
Student9,10,Thread-9,91,92,87,C
Student10,11,Thread-10,80,97,85,C
Student11,12,Thread-11,90,98,70,C
Student12,13,Thread-12,75,86,99,D
Student13,14,Thread-13,86,77,82,D
Student14,15,Thread-14,82,92,98,C
Student15,16,Thread-15,85,71,72,D
Student16,17,Thread-16,82,74,96,D
Student17,18,Thread-17,79,95,77,D
Student18,19,Thread-18,90,81,97,C
Student19,20,Thread-19,73,85,72,D
Student20,21,Thread-20,75,77,78,D
Student21,22,Thread-21,94,93,94,C
Student22,23,Thread-22,87,75,98,D
Student23,24,Thread-23,99,85,79,C
Student24,25,Thread-24,83,98,87,C
Student25,26,Thread-25,75,92,70,D

JVM Model of Student Program

- For given program I have created two class which implements the runnable interface 1) StudentThread and 2)Grader Thread. Apart from this to perform the reading , writing and updating the Grade.txt file I have created one Utils classs.
- So I am creating the 25 different threads for each student and then assign the id , name, homework, midterm and final score randomly between 70 to 100.
- So Inside the JVM run time memory area it will create the 25 different thread stack for each thread such as thread-1 to thread-25.





- Each thread stack contains the all local variables and reference to class variable which is pointing to heap memory.
- Apart from this it will also contains the frame stack of method execution so every time the local method return something the frame will popped out from that stack in LIFO manner.
- In my solution I have used the Utils class which is performing data reading and writing task. During the

creation of each StudentThread I passed the instance of utils class.

- Inside the util class I have created two three synchronized method
    - writeData → to write data in Grade.txt file
    - readFile→ Read the data from Grade.txt file
    - updateFile→ Update the letter grade in grade.txt file
- I have used the object level lock mechanism so I am acquiring the lock on instance of utils class.
- So Inside the main method I am creating new thread named as GraderThread and start that thread. So after every 30 seconds ,this thread will read the Grade.txt file and calculate the letter grade.
- Initially this thread will wait so StudentThread can write into Grade.txt file
- After the 30 seconds this thread start reading the grade.txt file and calculate the grade and update the grade in grade.txt file.
- To synchronized the letter grade with each thread I have created ConcurrentHashMap object which store the student thread id as key and letter grade as value.

4. Does Thread synchronization works correctly with the following code? Why or Why not? If Not, how do you fix it?

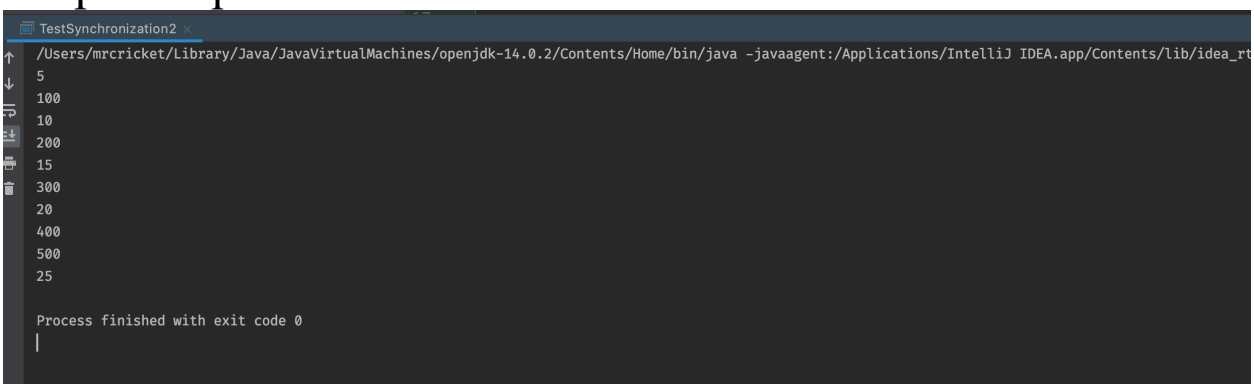
```
//example of java synchronized method
class Table{
 synchronized void printTable(int n){//synchronized method
   for(int i=1;i<=5;i++){
     System.out.println(n*i);
     try{
      Thread.sleep(400);
     }catch(Exception e){System.out.println(e);}
   }
 }
}

class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
}
}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
```

```
    this.t=t;
    }
    public void run(){
    t.printTable(100);  } }
public class TestSynchronization2{
public static void main(String args[]){
Table obj1 = new Table();//only one object
// Obj2 initialization is missing

MyThread1 t1=new MyThread1(obj1);
MyThread2 t2=new MyThread2(obj2);
t1.start();  t2.start();  }}
```

Answer:

In above program synchronization will not work properly because we are passing two different object of Table class so here the object level lock will not work.

Output snap:



Now we have two option to fix this problem

1) we can pass the same table object to each thread so we can utilize the object level lock functionality

   MyThread1 t1=new MyThread1(obj1);
   MyThread2 t2=new MyThread2(obj1); <------ ***this allow us to use object level lock***

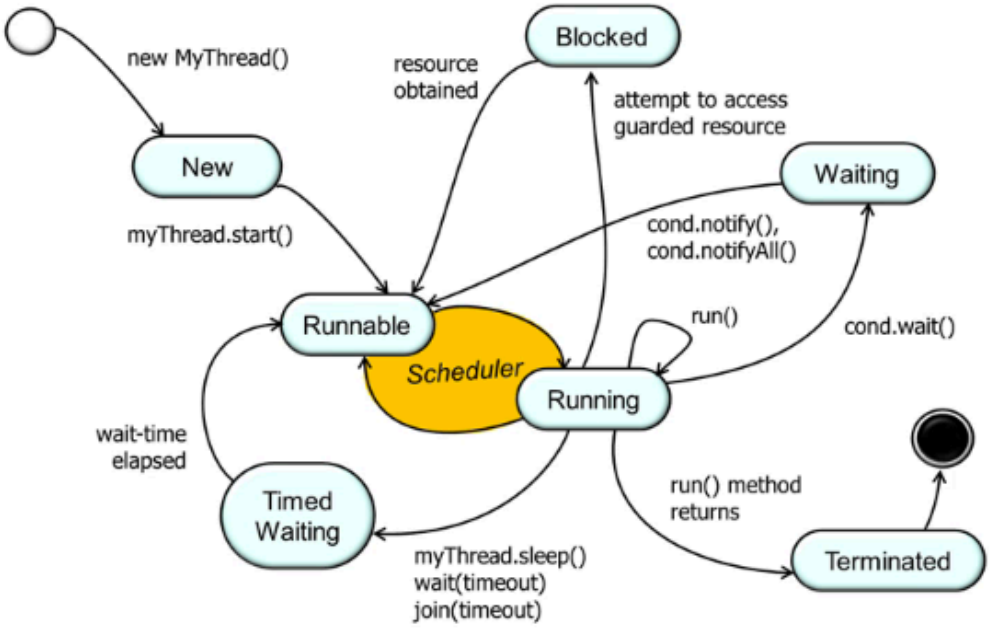
2) we can make the **printable** method as static method so we can acquire lock on class object itself.

```
synchronized static void printTable(int n){//synchronized method
    for(int i=1;i<=5;i++){
      System.out.println(n*i);
      try{
       Thread.sleep(400);
      }catch(Exception e){System.out.println(e);}
    }
  }
```

In above case we are making printable as static method so we can make sure that multiple thread with different table instance execute this method in synchronized manner.

   MyThread1 t1=new MyThread1(obj1);
   MyThread2 t2=new MyThread2(obj2); <------ ***it will work as we have used class level lock.***

5. Explain Thread State Diagram



- when we creates a new thread and use the **thread.start()** method thread will go to in Runnable state.
- After that scheduler if scheduler pick that thread and run it then it will be in running state.
- If the Running thread use the any sleep method such as sleep(), wait() or join() then it will go into time waiting state for the **particular time period which mentioned in sleep methods**.
- If the thread in Running state and use the **thread.wait()** method then thread will go into waiting state until other thread notify the current thread using **notify() or notifyAll**() method.
- If the current thread trying to access the guarded resources such as synchronized block and that resources is already in

use by other threads, then current thread will go into Blocked state until it obtained the resource.

- At the end, once the run method returns the value current thread gets terminated.

6. Consider the following code segments:

A) Compile and Run the following code segments

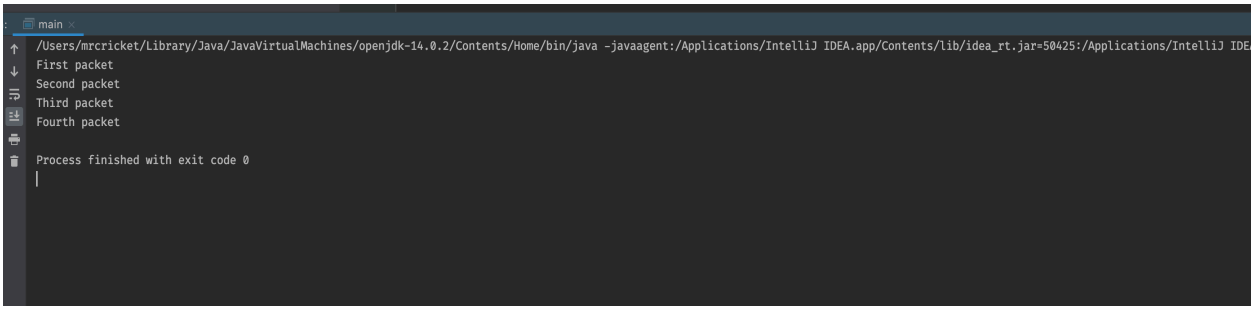
B) Explain the code and how Wait/Notify synchronization works with

transfer flag? Send Receive packet example

```
public class Data {
    private String packet;

    // True if receiver should wait
    // False if sender should wait
    private boolean transfer = true;

    public synchronized void
send(String packet) {
        while (!transfer) { // Tansfer
is false means receiver yet not received
the packet so wait for that
            try {
                wait();
            } catch
(InterruptedException e)  {
                Thread.currentThread().
interrupt();
                Log.error("Thread
interrupted", e);
            }
        }
        transfer = false;
```

```
            this.packet = packet;
            notifyAll();
        }

        public synchronized String
receive() {
            while (transfer) { // True that
means Sender is sending a packet
                try {
                    wait();
                } catch
(InterruptedException e)   {
                    Thread.currentThread().
interrupt();
                    Log.error("Thread
interrupted", e);
                }
            }
            transfer = true;

            notifyAll();
            return packet;
        } }
```

Output :



How Programs works?

- Packet variable denotes as the data that is being transferred over the network.

- We have used the transfer flag to make sure that sender will send the packet only after receiver received previous packet otherwise sender should wait vice versa receiver should wait while sender is sending a new packet.
- The Sender will use the send() method to send the data to receiver.
    - If transfer = false means sender should wait
    - If transfer = true then sender should notify all other thread to wake up using notifyAll() method.
- Receiver will use Receiver Method to receive data
    - If transfer = false thread will proceed the data and notify all other threads using notifyAll() method
    - If transfer = true then thread will wait

7. Consider the following Thread example:

A) Compile and Run the following code segments

B) Explain the code, What are object monitors?

```
public class Input {
    int index;
    int[] input = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};

    public Input(){
        index = 0;
    }

    public void print(int index){
        System.out.println(input[index]);
    }

  Synchronized method block make sure that
at a time only one thread can execute this
method
    synchronized public int getIndex(){
        if(index == 15)
            return -1;
        return index++;
    }
}

public class MyThread implements Runnable{

    Input ip;
```

```
        Object lock;

        public MyThread(Input ip, Object lock){
            this.ip = ip;
            this.lock = lock;
        }

        @Override
        public void run() {
            int index = -1;
            while((index=ip.getIndex())!=-1){
                synchronized(lock) {
```

used object lock approach to makes sure that single instance of an object can access this block

```
System.out.println(Thread.currentThread().getName());
                    ip.print(index);
                }
            }
        }
    }
    public class Caller {

        public static void main(String[] args) throws
InterruptedException {
            Input ip = new Input();
            Object lock = new Object();
            Thread t1 = new Thread(new MyThread(ip, lock),
"Thread1");
            Thread t2 = new Thread(new MyThread(ip, lock),
"Thread2");
            t1.start();
            t2.start();
            t1.join();
            t2.join();
        }
    }
```

Output:

Explanation:

- In caller class, we have created the **object of the Input class** and **lock instance of the object class which will monitor the lock on current instance method**. We have passed the same lock object to thread t1 and t2 to make sure that our synchronized code block(inside the MyThread class run method) execute by on one thread at the same time.
- similar way we have passed the **same input object to thread t1 and t2**. We used that object to call **the getIndex method** which is again **synchronized method** so it makes sure that **it is executed by a single thread at a given time**.
- We can confirm this by checking the output screenshot which clearly shows that at a given time each thread gets the different value form the **getIndex()** method.

- At the end of the main method, we have used the **thread.Join()** method which makes sure that **the main thread not get terminated before thread t1 and t2**.

What are object monitors?

Object monitor is a **mechanism** that **allow thread to have mutual exclusive lock** means only **one thread** can **execute** the method at certain **point in time** using **lock** and **cooperation** means **ability to make threads wait** for certain condition to be met using **wait and set**

8. Consider the following code segments: Wait notify example

A) Compile and Run the following code segments

B) Explain the code and how the synchronization of code works?

```
package com.journaldev.concurrency;

public class Message {
    private String msg;

    public Message(String str){
        this.msg=str;
```

```
    }

    public String getMsg() {
        return msg;
    }

    public void setMsg(String str) {
        this.msg=str;
    }
}

package com.journaldev.concurrency;

public class Notifier implements Runnable {
    private Message msg;
    public Notifier(Message msg) {
        this.msg = msg;
    }

    @Override
    public void run() {
        String name = Thread.currentThread().getName();
        System.out.println(name+" started");
        try {
            Thread.sleep(1000);
            synchronized (msg) {
                msg.setMsg(name+" Notifier work done");
                msg.notify();
                // msg.notifyAll();
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
```

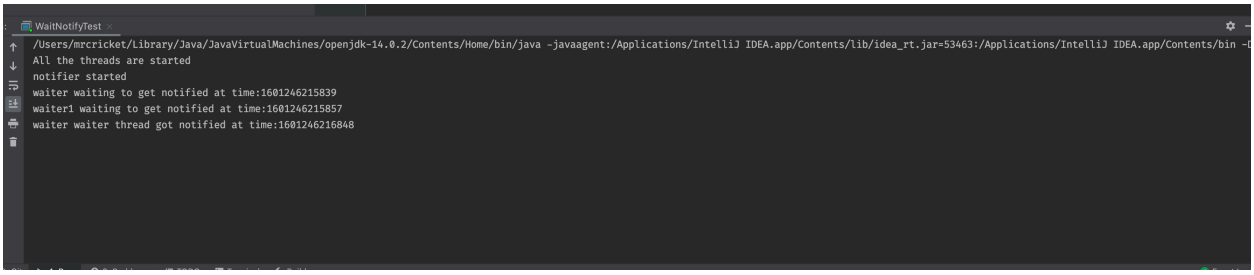
```
        }
    }
}}
package com.journaldev.concurrency;

public class WaitNotifyTest {

    public static void main(String[] args) {
        Message msg = new Message("process it");
        Waiter waiter = new Waiter(msg);
        new Thread(waiter,"waiter").start();

        Waiter waiter1 = new Waiter(msg);
        new Thread(waiter1, "waiter1").start();

        Notifier notifier = new Notifier(msg);
        new Thread(notifier, "notifier").start();
        System.out.println("All the threads are started");
    }

}
```

Explanation

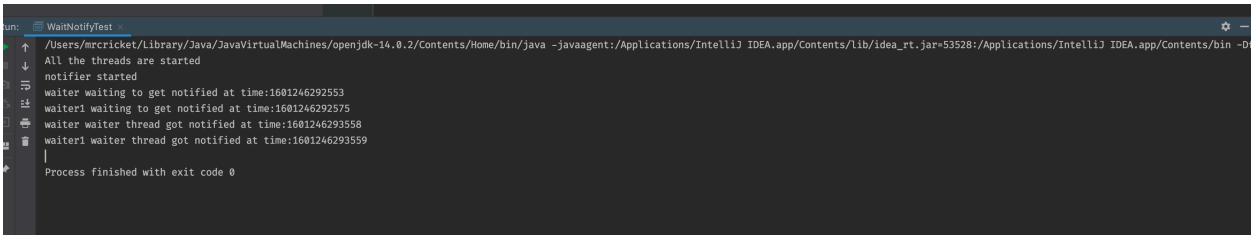
- In given program we have created waiter class and notifier class which implements the runnable interface and we have message class which is simple contains message.

- In main class we are creating multiple objects of the waiter class waiter and wiater1 and passing message objects and start that threads using thread.start() method.
- In Run method of waiter class we are printing the waiting message and hold the execution of that thread using thread.wait() method.
- Meanwhile we are also creating instance of notifier class and start the thread.In Notifier class run method we are printing the thread started message and after that inside the synchronized block we are setting the message value and then we notify the other thread (waiter / waiter1) using thread.notify() method.
- As we have used the notify method on of the waiter thread start their execution while other will still in waiting state.
- If we use the thread.notifyAll() method then both the waiter thread start their execution.
- We can verify the behavior of notify and notifyAll method by output screenshots

Output with notify() method



Output with notifyAll() method



Assignment 3:

1. Explain:

Basic components of computer architecture ? Memory Heap?

Memory heap is an area of pre-reserved computer **main storage (memory)** that a program process can use to store data in some variable amount that won't be known until the program is running.

When you compile and execute your program (eg: Hello.java), what happens? Provide details of execution environment. JVM memory architecture components?

Let’s consider the following simple hello.java file

```
Public class hello{

    Public static void main(String args[]){
```

```
        // code
    }

}
```

Execution steps

1) java compiler compile this file and convert into hello.class file and send to JVM.
2) Once the JVM receive this file it perform following steps
    a. It will load the class file in main memory which is know as dynamic class loading using following class loader
        i. Bootstrap classloader
            - Responsible for load the class from bootstrap classpath
        ii. Extension classloader
            - Responsible for load the classes from ext folder
        iii. Application classloader
            - Responsible for load application level classpath
    b. Inside the linking phase JVM verify the weather the generated bytecode is proper or not then allocate the memory for static variables and resolve the symbolic memory reference with original reference from memory area.
3) After all this process Run time data area comes into picture. It is divided into five major parts .

a. **Method area** : All the class-level data will be stored here, including static variables. There is only one method area per JVM, and it is a shared resourc
b. **Heap Area** – All the Objects and their corresponding instance variables and arrays will be stored here. There is also one Heap Area per JVM. Since the Method and Heap areas share memory for multiple threads, the data stored is not thread-safe
c. **Stack Area** – For every thread, a separate runtime stack will be created. For every method call, one entry will be made in the stack memory which is called Stack Frame. All local variables will be created in the stack memory
d. **PC Registers** – Each thread will have separate PC Registers, to hold the address of current executing instruction once the instruction is executed the PC register will be updated with the next instruction.
e. **Native Method stack**s – Native Method Stack holds native method information. For every thread, a separate native method stack will be created.

4) Once the bytecode assigned to run time data area , execution engine start executing bytecodes. JVM execution engine contains following components
   a. **Interpreter** – The interpreter interprets the bytecode faster but executes slowly. The disadvantage of the interpreter is that when one method is called multiple times, every time a new interpretation is required.
   b. **JIT Compiler** – The JIT Compiler neutralizes the disadvantage of the interpreter. The Execution Engine

will be using the help of the interpreter in converting byte code, but when it finds repeated code it uses the JIT compiler, which compiles the entire bytecode and changes it to native code. This native code will be used directly for repeated method calls, which improve the performance of the system.

c. **Garbage Collector**: Collects and removes unreferenced objects. Garbage Collection can be triggered by calling System.gc(), but the execution is not guaranteed. Garbage collection of the JVM collects the objects that are created.

5) JVM also contains JNI and native method library. JNI will interact with native method library and provides the native library required for the execution.

JVM Heap?

- All the Objects and their corresponding instance variables and arrays will be stored here. There is also one Heap Area per JVM. Since the Method and Heap areas share memory for multiple threads, the data stored is not thread-safe
- JVM heap is divided into 2 parts
    1) Young generation
    2) Old Generation

1) Young generation

- This is reserved for newly created object. Young Gen includes three parts Eden Memory and two Survivor Memory spaces (S0, S1)

- Most of the newly-created objects goes Eden space. When Eden space is filled with objects, Minor GC (a.k.a. Young Collection) is performed and all the survivor objects are moved to one of the survivor spaces.
- Objects that are survived after many cycles of GC, are moved to the Old generation memory space. Usually it's done by setting a threshold for the age of the young generation objects before they become eligible to promote to Old generation

2) Old generation

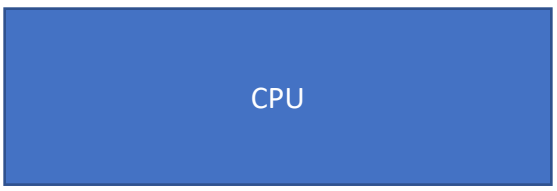
- This is reserved for containing long lived objects that could survive after many rounds of Minor GC
- When Old Gen space is full, Major GC is performed

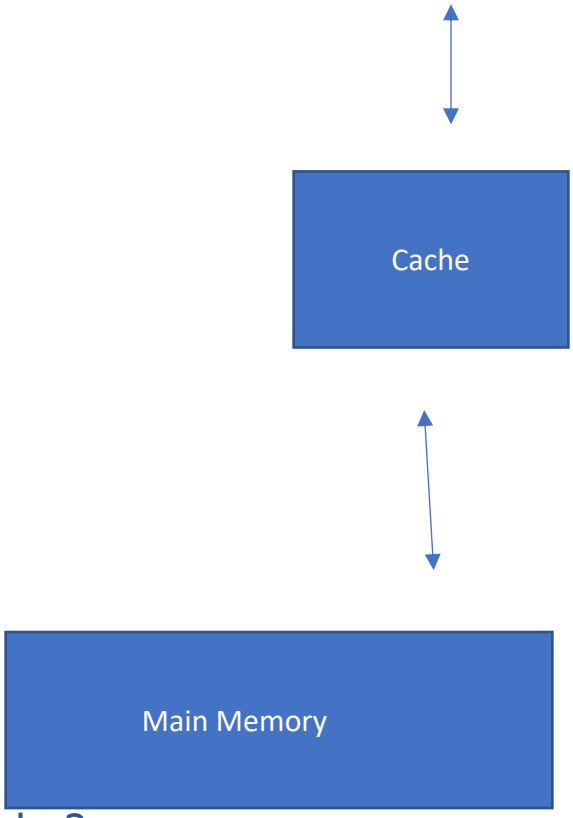
JVM Non-Heap?

JVM non -heap include the permanent generation. It stores the per class structure such as runtime constant pool, field and method data and interned string.

Cache?

Cache is a special type of very high-speed memory. It is used to speed up and synchronizing with high-speed CPU. Cache memory is an extremely fast memory type that acts as a buffer between RAM and the CPU. It holds frequently requested data and instructions so that they are immediately available to the CPU when needed.





Native Cache?

The Just-In-Time (JIT) compiler stores the compiled code in an area called code cache. It is a special heap that holds the compiled code. This area is flushed if its size exceeds a threshold and these objects are not relocated by the GC

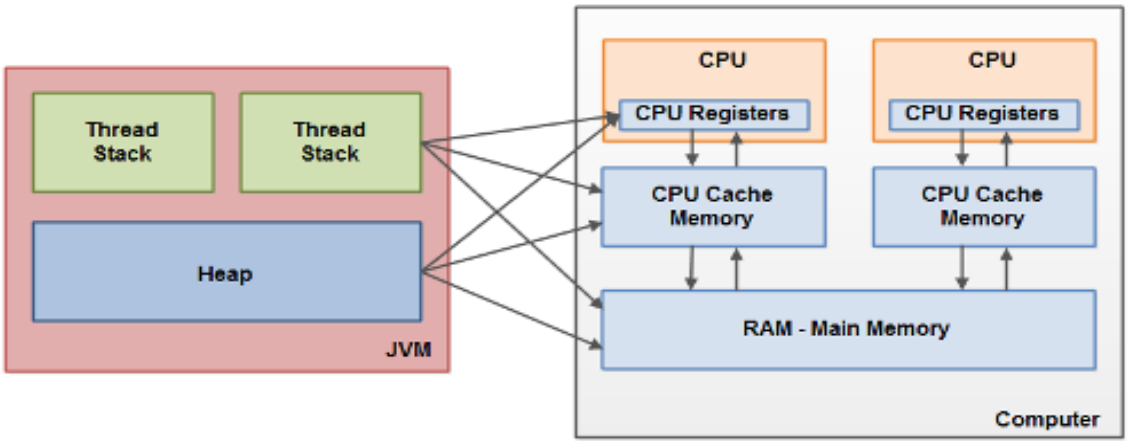
Java Native Interface (JNI)?

JNI will be interacting with the Native Method Libraries and provides the Native Libraries required for the Execution Engine

Why JVM is threadSafe?

In JVM for every thread, a separate **runtime stack will be created**. For every method call, one entry will be made in the stack memory which is called Stack Frame. All local variables will be created in the stack memory. The stack area is **thread-safe since it is not a shared resource.**

2. How does Java Memory Model and Hardware Memory Architecture are different and how they work together?



- Java memory model is different than the Hardware memory architecture.

As we know that inside the JVM memory **divide s into thread stack and the heap while in hardware memory thread stack and heap both located in main memory.**

- Each thread running in the Java virtual machine has its own thread stack. The thread stack contains information about what methods the thread has called to reach the current point of execution
- While hardware memory **model has different layers of memory such as CPU register, CPU cache memory level 1 and level 2 and main memory**. part of the heap and thread stack is residing in each layer of memory.

3. When objects and variables can be stored in various different memory areas in the computer, certain problems may occur. The two main problems are:

a) Visibility of thread updates (writes) to shared variables, provide an example? One way to solve visibility issue is to use Volatile keyword, how does it work? Provide example.

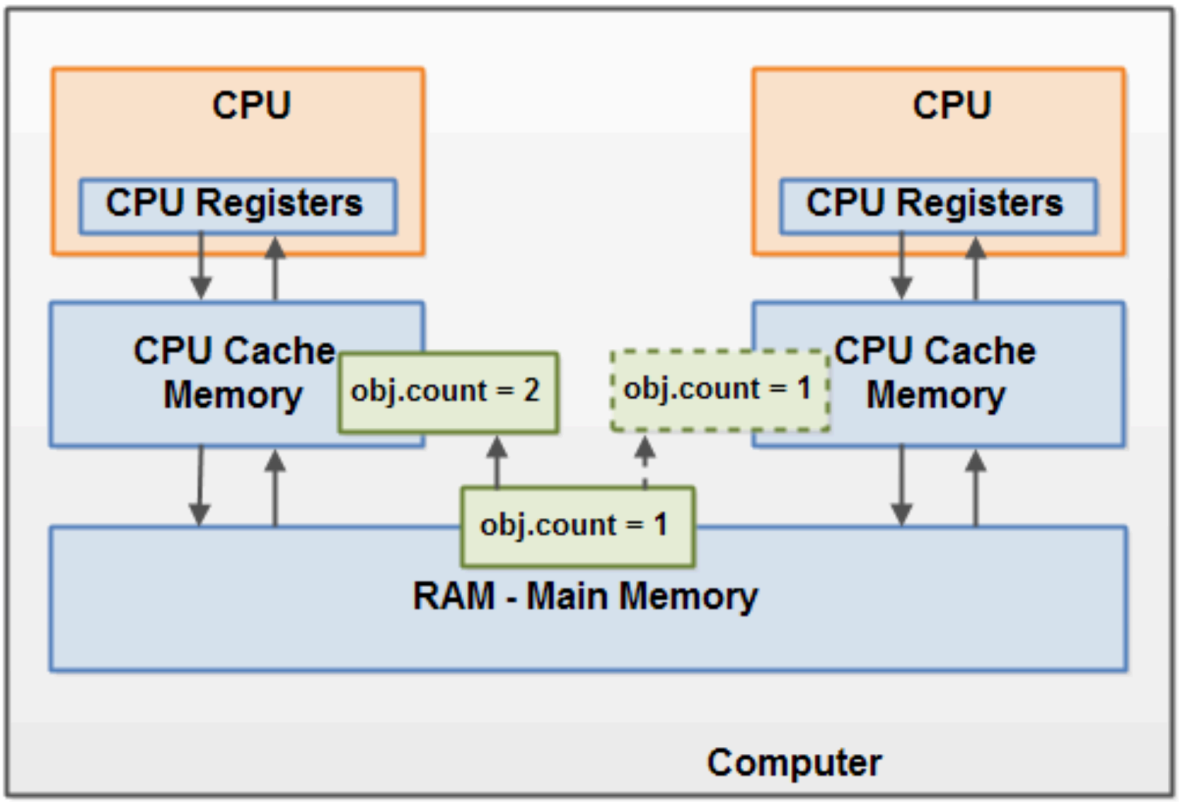
b) Race conditions when reading, checking and writing shared variables. Provide example of each, Explain as how these two problems can occur.

Visibility of thread update

- If two or more threads are sharing an object, without the proper use of either volatile declarations or synchronization, updates to shared object made by one thread may not be visible to other threads.
- Imagine that shared object is initially stored in main memory. A thread running on CPU one then reads the shared object into its CPU cache. This makes a change to shared object.
- As long as the CPU cache has not been flushed back to main memory, the changed version of the shared object is not visible to threads running on other CPUs. This way

each thread may end up with its own copy of the shared object, each copy sitting in a different CPU cache.

- The following diagram illustrates the sketched situation. One thread running on the left CPU copies the shared object into its CPU cache, and changes its count variable to 2.
- This change is not visible to other threads running on the right CPU, because the update to count has not been flushed back to main memory yet.
- To solve this problem you can use Java's volatile keyword. The volatile keyword can make sure that a given variable is read directly from main memory, and always written back to main memory when updated.



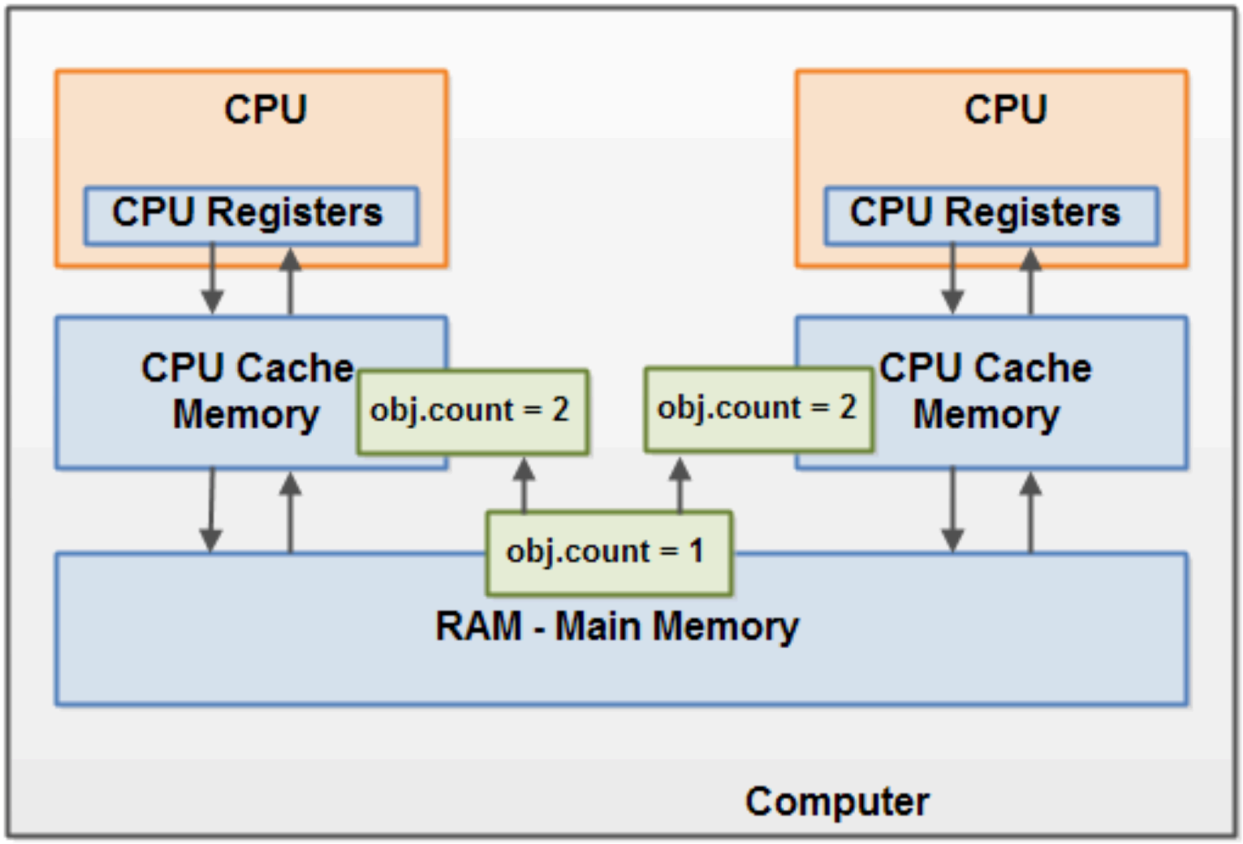
```
Public class TestVisibility implements Runnable{

    Int count =0; // we can solve the issue by making this volatile
    Public void run(){
        Count++;
    }
}

Public class main{
    Public static void main(String args[]){
    TestVisibility ts = new TestVisibility();
    Thread t1 = new Thread(ts);
    Thread t2 = new Thread(ts);
    t1.start();
    t2.start();
    }
}
```

Race conditions

- if two or more threads share an object, and more than one thread updates variables in that shared object, race conditions may occur.
- Thread A reads variable count shared object into its CPU cache, and Thread B does the same but into a different CPU cache. Now thread A adds one to count, and thread B does the same. Now var1 has been incremented two times, once in each CPU cache.

- If these increments had been carried out sequentially, the variable count would be incremented twice and had the original value + 2 written back to main memory.
- However, the two increments have been carried out concurrently without proper synchronization. Regardless of which thread A and B that writes its updated version of count back to main memory, the updated value will only be 1 higher than the original value, despite the two increments.

This diagram illustrates an occurrence of the problem with race conditions:



- To solve race condition problem we can use a Java **synchronized block**. A synchronized block guarantees that **only one thread can enter a given critical section of the code at any given time**. Synchronized blocks also guarantee that all variables accessed inside the synchronized block will be **read in from main memory, and when the thread exits the synchronized block, all updated variables will be flushed back to main memory again**, regardless of whether the variable is declared volatile or not.

Program :

```
Public class TestVisibility implements Runnable{

    Int count =0; // we can solve the issue by making this
volatile
    Public void run(){
        synchronized(this)
            {
            Count++;
            }
    }
}

Public class main{
    Public static void main(String args[]){
    TestVisibility ts = new TestVisibility();
    Thread t1 = new Thread(ts);
```

```
        Thread t2 = new Thread(ts);
        t1.start();
        t2.start();
        }
}
```

Q:4 Student Program

4. In Homework2, you defined Student class, and created 25 student threads and created one GraderThread. Change previous homework, and have GraderThread to write final grade to file “FinalGrades”. This is how things going to work:
A) Create 40 student threads, each thread records 3 scores (homework, midterm-exam, final-exam), and each score is randomly generated between 70 to 100 with incremental 1second interval. Use a List to store scores and is set to zeros initially. Use Map with key/value, (List, threadId).

B) Each student thread first generates homework score within 1 second, stores score in List [0], and then writes List to map. Next, it generates midterm-exam score within the next second, stores score to List[1] and writes List to map. Next, it generates final-exam score within the next second, stores score in List[2] and writes List to map.

C) GraderThread randomly reads scores from Map, calculates final grade for Letter
A, B, C, D, F, and writes the final grade to file “FinalGrades”.

D) Each student thread, Reads final grade from file “FinalGrades” and reports its grade,

Notes: Things to think about: 1) How to protect Map and File? 2) student Threads writing to Map and Reading from File, 3) GraderThread is reading Map and writing to File. 4) student

threads write List to Map with incremental List update for each score. Grader thread reads Map for each student scores, the Grader can move to calculate letter grade calculation and write the grade to the file?, the Grader thread does not need to wait for all student threads, once each student completes scores, grader thread can move to calculate grade. 5) Student threads often read file to see if final grade is submitted by the grader thread, think thread safety for all reads and writes. Carefully read steps described above.
Note: Because there are many read and write operations to the Map and File, between student threads and grader thread, there is possibility for Race Condition, How does that is possible?, Explain.

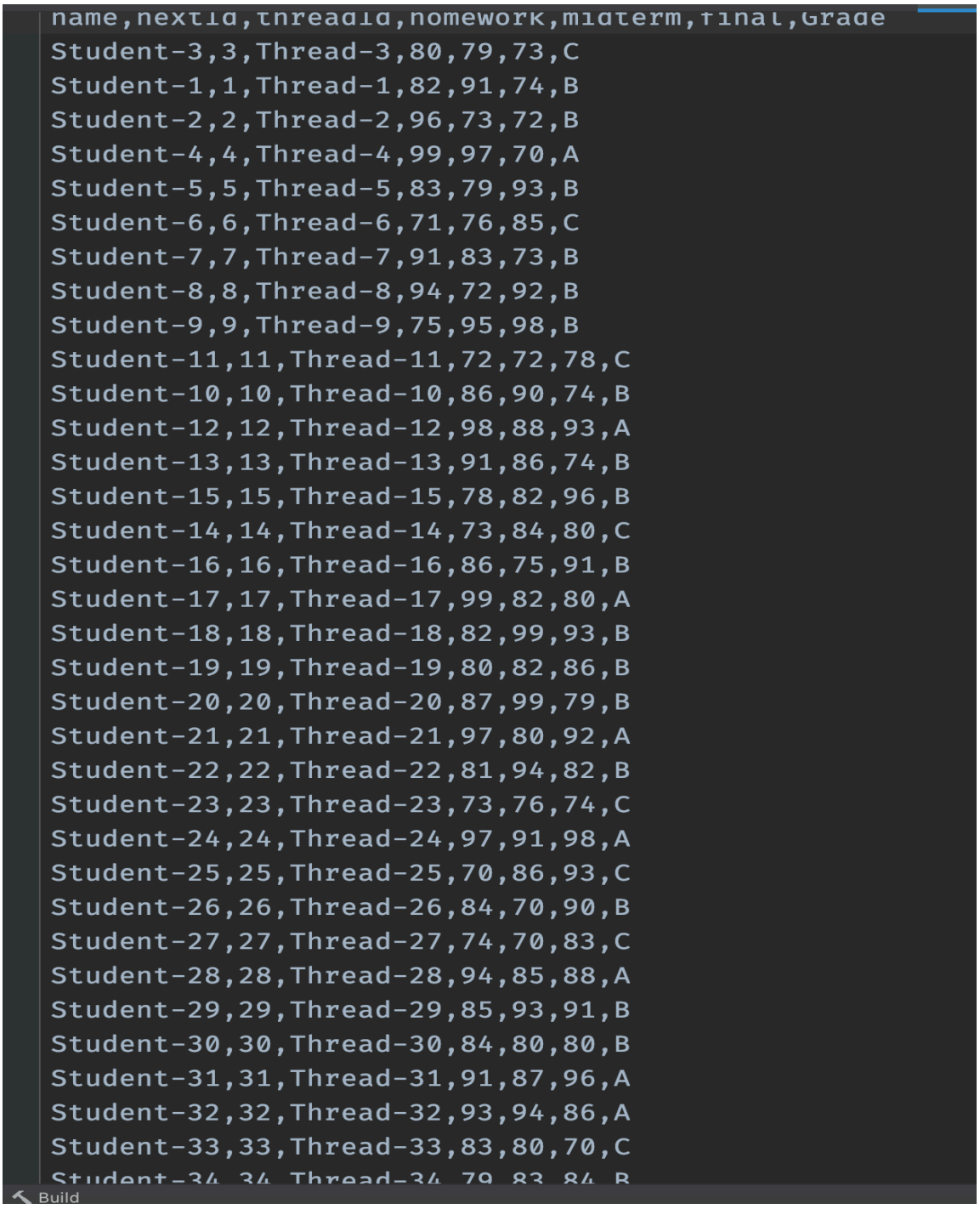
Possible race condition Explanation:

- In a given problem we are creating 40 different threads using student class.
- Here each thread will generate different score for homework, midterm and final exam in difference of 1 second and store that score inside the List and store that list inside the Map collection.
- Concurrently grader thread will try to access that Map object and list of all the grades to generate Letter grade. In

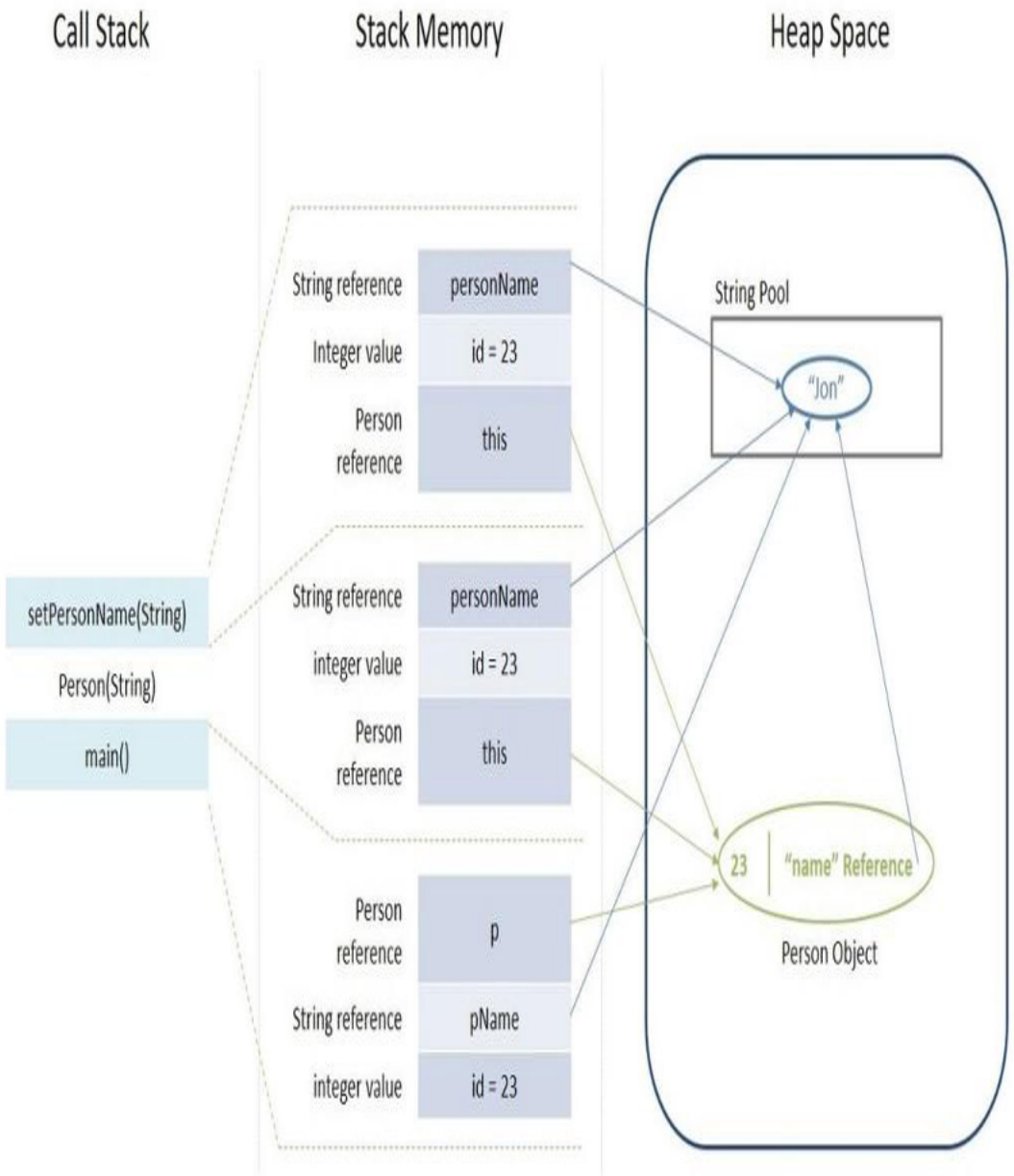
this case race condition wilk occur All the student thread try to write data in map object and we need to make sure that grader thread will access the data once the student thread complete their write operation.

- We have to writes program in such way that the grader thread should wait until student thread complete their write operation. We can achieve this using synchronized block and wait() and notifyall() **functions**
- **Second** part of the problem is once the Grader thread calculate the letter grade it will write that data in grade.txt file and all the student thread will try to read that file and get the updated grade. As this two operations occurs concurrently race condition may occur to get access the grade.txt file
- We can also handle this situation using synchronized block , wait() and notifyall() method
- For implementation detail you can check the code files.

```
Thread-1 [82, 97, 97]
Thread-2 [76, 88, 82]
Thread-3 is reading this line
Student{name='Student-3', id=3, homework=72, midterm=92, finalexam=95, Grade=B}
Thread-4
GraderThread is processing map
Thread-3 [72, 92, 95]
Thread-4 [99, 94, 74]
Student{name='Student-2', id=2, homework=76, midterm=88, finalexam=82, Grade=B}
Thread-5
Thread-1 [82, 97, 97]
Thread-2 [76, 88, 82]
Student{name='Student-4', id=4, homework=99, midterm=94, finalexam=74, Grade=A}
Thread-6
GraderThread is processing map
Thread-3 [72, 92, 95]
Thread-4 [99, 94, 74]
Thread-5 [86, 70, 90]
Thread-6 [84, 75, 98]
Thread-1 [82, 97, 97]
Thread-2 [76, 88, 82]
Thread-6 is reading this line
Student{name='Student-6', id=6, homework=84, midterm=75, finalexam=98, Grade=B}
Thread-7
GraderThread is processing map
Thread-3 [72, 92, 95]
Thread-4 [99, 94, 74]
Thread-5 [86, 70, 90]
Thread-6 [84, 75, 98]
Thread-7 [85, 70, 74]
Thread-1 [82, 97, 97]
Thread-2 [76, 88, 82]
Thread-7 is reading this line
Student{name='Student-7', id=7, homework=85, midterm=70, finalexam=74, Grade=C}
```


name,nextId,threadId,homework,midterm,final,Grade
Student-3,3,Thread-3,80,79,73,C
Student-1,1,Thread-1,82,91,74,B
Student-2,2,Thread-2,96,73,72,B
Student-4,4,Thread-4,99,97,70,A
Student-5,5,Thread-5,83,79,93,B
Student-6,6,Thread-6,71,76,85,C
Student-7,7,Thread-7,91,83,73,B
Student-8,8,Thread-8,94,72,92,B
Student-9,9,Thread-9,75,95,98,B
Student-11,11,Thread-11,72,72,78,C
Student-10,10,Thread-10,86,90,74,B
Student-12,12,Thread-12,98,88,93,A
Student-13,13,Thread-13,91,86,74,B
Student-15,15,Thread-15,78,82,96,B
Student-14,14,Thread-14,73,84,80,C
Student-16,16,Thread-16,86,75,91,B
Student-17,17,Thread-17,99,82,80,A
Student-18,18,Thread-18,82,99,93,B
Student-19,19,Thread-19,80,82,86,B
Student-20,20,Thread-20,87,99,79,B
Student-21,21,Thread-21,97,80,92,A
Student-22,22,Thread-22,81,94,82,B
Student-23,23,Thread-23,73,76,74,C
Student-24,24,Thread-24,97,91,98,A
Student-25,25,Thread-25,70,86,93,C
Student-26,26,Thread-26,84,70,90,B
Student-27,27,Thread-27,74,70,83,C
Student-28,28,Thread-28,94,85,88,A
Student-29,29,Thread-29,85,93,91,B
Student-30,30,Thread-30,84,80,80,B
Student-31,31,Thread-31,91,87,96,A
Student-32,32,Thread-32,93,94,86,A
Student-33,33,Thread-33,83,80,70,C
Build

5. Reverse engineer the following JVM memory model explain what object it represents in detail?



Aforementioned Java model represents the following code.

```
class Person {
   int pid;
   String personName;
// constructor, setters/getters

       Void setPersonName(String name)
       {
          this. personName = name ;
       }
}
public class Driver {
   public static void main(String[] args) {
      int id = 23;
      String pName = "Jon";
      Person p = null;
      p = new Person(id, pName);
   }
}
```

<u>Explanation:</u>

- Inside the call stack we can see that when we start the program it will call the main() function then Person() and at last setPersonName() function.

- Stack memory shows the how the object are created. In first block we can **see that Pname string is directly pointing to string constant pool** while int is **primitive type it store it's value inside the memory stack and person object refers to Null object.**
- After that we are calling person() constructor method and assign string value using setPerName() methed

6. Provide Call Stack, Stack Memory, and Heap Space for the following code:

```
class MyThread implements Runnable {

String name;

Thread t;
```

```
MyThread String thread){

        name = thread;

        t = new Thread(this, name);

       System.out.println("New thread: " + t);

       t.start();

}


public void run() {

 try {

    for(int i = 5; i > 0; i--) {

    System.out.println(name + ": " + i);

     Thread.sleep(1000);
threadname
}

}catch (InterruptedException e) {
```

```
    System.out.println(name + "Interrupted");
}
    System.out.println(name + " exiting.");
}
}


class MultiThread {
public static void main(String args[]) {
    new MyThread("One");
    new MyThread("Two");
    new NewThread("Three");
try {
    Thread.sleep(10000);
} catch (InterruptedException e) {
     System.out.println("Main thread Interrupted");
```

```
}

    System.out.println("Main thread exiting.");

    }
```

Above program creates the three instance of the MyThread class
And it also contain the thread object which start in object initialization

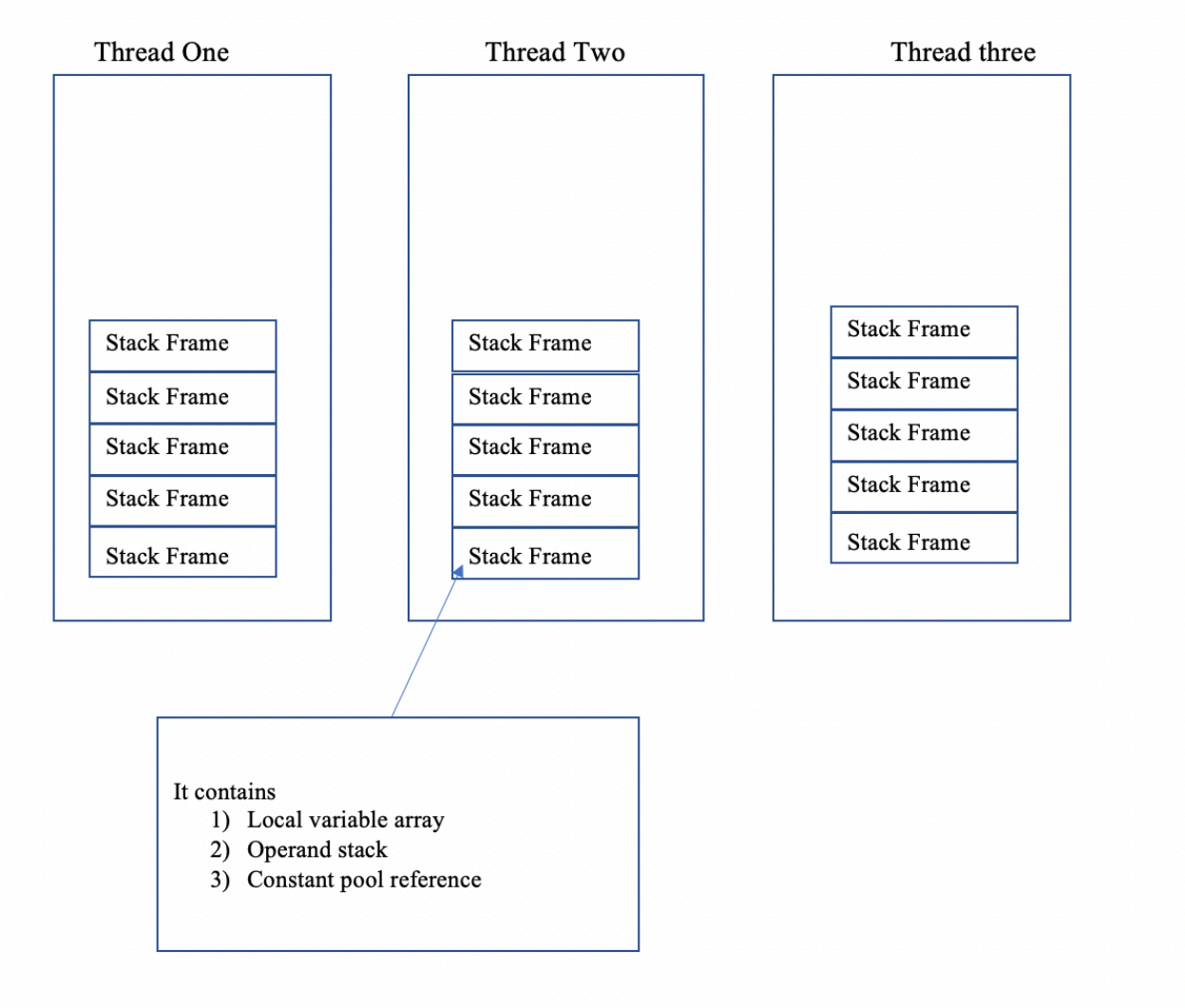
Call Stack :

Run()
Run()
Run()
Start()
new Thread()
MyThread(“three”)
Start()
new Thread()
MyThread(“two”)
Start()
new Thread()
MyThread(“One”)
Main()

// Run method are depended on how operating system assign the memory to particular thread to execute that method

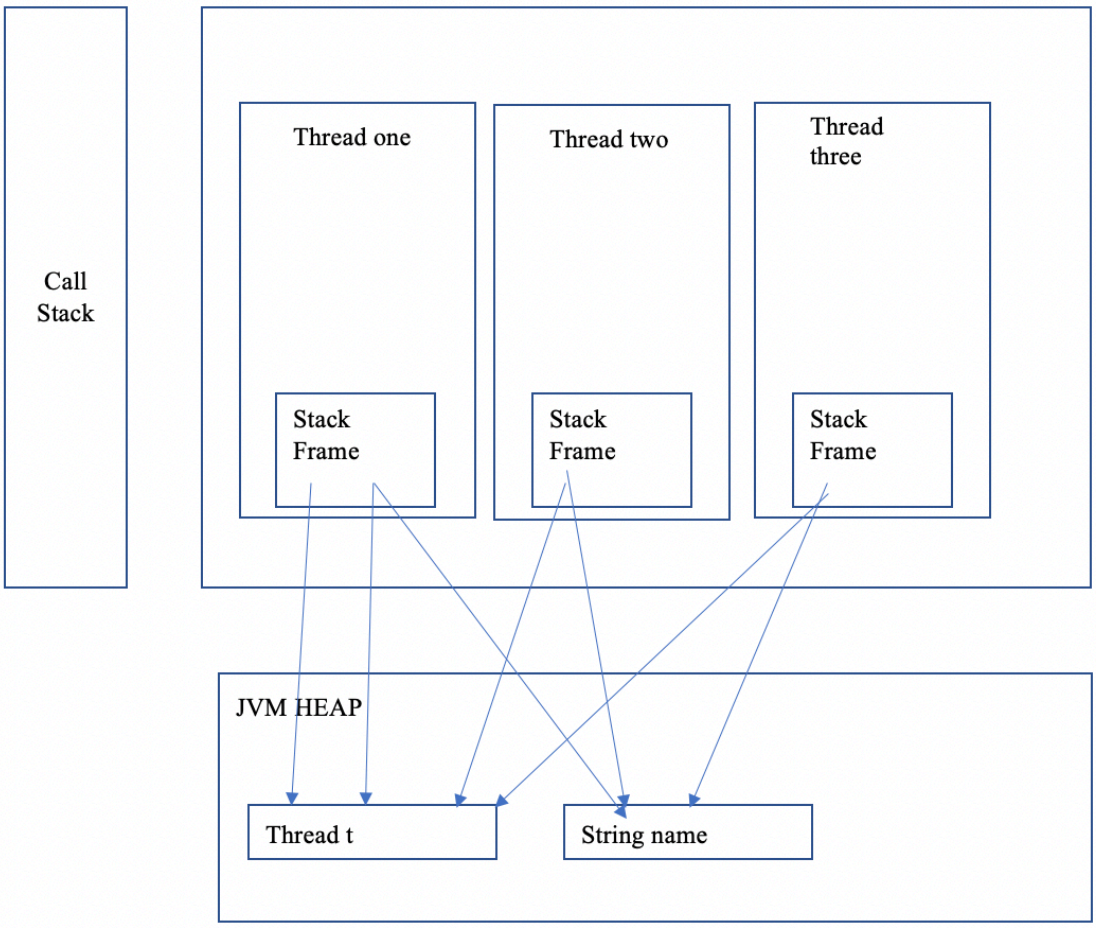
- Call stack store the current point of execution it perform the operation in **LIFO (Last in First out)** manner

As we are creating separate three threads we JVM creates **three different Stack for each thread**



In Above figure you can find the thread different stack thread for each thread and each thread stack contains the stack frame.

So MyThread class have two reference variable one is String name and another is thread t so stackframe will store the reference of both. The actual address of String and Thread instance is inside the JVM heap.


Call
Stack
Thread one
Thread two
Thread
three
Stack
Frame
Stack
Frame
Stack
Frame
JVM HEAP
Thread t
String name

7. Consider the following code segments:

A) Compile the following code segments

B) Use Javap to disassemble code, report

Javap -l -v -c WaitNotifyTest.class

C) Explain the report

```
package com.journaldev.concurrency;

public class Message {
   private String msg;

   public Message(String str){
      this.msg=str;
   }

   public String getMsg() {
      return msg;
   }

   public void setMsg(String str) {
      this.msg=str;
   }
}
```

```
package com.journaldev.concurrency;

public class Notifier implements Runnable {
   private Message msg;
```

```
    public Notifier(Message msg) {
        this.msg = msg;
    }

    @Override
    public void run() {
        String name = Thread.currentThread().getName();
        System.out.println(name+" started");
        try {
            Thread.sleep(1000);
            synchronized (msg) {
                msg.setMsg(name+" Notifier work done");
                msg.notify();
                // msg.notifyAll();
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}}


package com.journaldev.concurrency;

public class WaitNotifyTest {

    public static void main(String[] args) {
        Message msg = new Message("process it");
        Waiter waiter = new Waiter(msg);
```

```
        new Thread(waiter,"waiter").start();

        Waiter waiter1 = new Waiter(msg);
        new Thread(waiter1, "waiter1").start();

        Notifier notifier = new Notifier(msg);
        new Thread(notifier, "notifier").start();
        System.out.println("All the threads are started");
    }

}
```

Javap command disassembles the one more classfile.

Please find the attach screenshot of Javap -l -v -c WaitNotifyTest.class

```
Classfile /Users/mrcricket/Desktop/Fall2020/Multithreading/Assignments/Assignment3/out/production/Assignment3/com/multithreding/Question7/WaitNotifyTest.class
  Last modified Oct 4, 2020; size 1229 bytes
  SHA-256 checksum e4a4e5591e31119c57d6c1243f12d6f1d630c7f3e4a030d0c5fcadcdbd0ed937
  Compiled from "WaitNotifyTest.java"
public class com.multithreding.Question7.WaitNotifyTest
  minor version: 0
  major version: 58
  flags: (0x0021) ACC_PUBLIC, ACC_SUPER
  this_class: #49                         // com/multithreding/Question7/WaitNotifyTest
  super_class: #2                         // java/lang/Object
  interfaces: 0, fields: 0, methods: 2, attributes: 1
Constant pool:
   #1 = Methodref          #2.#3          // java/lang/Object."<init>":()V
   #2 = Class              #4             // java/lang/Object
   #3 = NameAndType        #5:#6          // "<init>":()V
   #4 = Utf8               java/lang/Object
   #5 = Utf8               <init>
   #6 = Utf8               ()V
   #7 = Class              #8             // com/multithreding/Question7/Message
   #8 = Utf8               com/multithreding/Question7/Message
   #9 = String             #10            // process it
  #10 = Utf8               process it
  #11 = Methodref          #7.#12         // com/multithreding/Question7/Message."<init>":(Ljava/lang/String;)V
  #12 = NameAndType        #5:#13         // "<init>":(Ljava/lang/String;)V
  #13 = Utf8               (Ljava/lang/String;)V
  #14 = Class              #15            // com/multithreding/Question7/Waiter
  #15 = Utf8               com/multithreding/Question7/Waiter
  #16 = Methodref          #14.#17        // com/multithreding/Question7/Waiter."<init>":(Lcom/multithreding/Question7/Message;)V
  #17 = NameAndType        #5:#18         // "<init>":(Lcom/multithreding/Question7/Message;)V
  #18 = Utf8               (Lcom/multithreding/Question7/Message;)V
  #19 = Class              #20            // java/lang/Thread
  #20 = Utf8               java/lang/Thread
  #21 = String             #22            // waiter
  #22 = Utf8               waiter
  #23 = Methodref          #19.#24        // java/lang/Thread."<init>":(Ljava/lang/Runnable;Ljava/lang/String;)V
  #24 = NameAndType        #5:#25         // "<init>":(Ljava/lang/Runnable;Ljava/lang/String;)V
  #25 = Utf8               (Ljava/lang/Runnable;Ljava/lang/String;)V
  #26 = Methodref          #19.#27        // java/lang/Thread.start:()V
  #27 = NameAndType        #28:#6         // start:()V
  #28 = Utf8               start
  #29 = String             #30            // waiter1
  #30 = Utf8               waiter1
  #31 = Class              #32            // com/multithreding/Question7/Notifier
  #32 = Utf8               com/multithreding/Question7/Notifier
  #33 = Methodref          #31.#17        // com/multithreding/Question7/Notifier."<init>":(Lcom/multithreding/Question7/Message;)V
  #34 = String             #35            // notifier
  #35 = Utf8               notifier
  #36 = Fieldref           #37.#38        // java/lang/System.out:Ljava/io/PrintStream;
  #37 = Class              #39            // java/lang/System
  #38 = NameAndType        #40:#41        // out:Ljava/io/PrintStream;
  #39 = Utf8               java/lang/System
  #40 = Utf8               out
  #41 = Utf8               Ljava/io/PrintStream;
  #42 = String             #43            // All the threads are started
  #43 = Utf8               All the threads are started
  #44 = Methodref          #45.#46        // java/io/PrintStream.println:(Ljava/lang/String;)V
  #45 = Class              #47            // java/io/PrintStream
  #46 = NameAndType        #48:#13        // println:(Ljava/lang/String;)V
  #47 = Utf8               java/io/PrintStream
  #48 = Utf8               println
  #49 = Class              #50            // com/multithreding/Question7/WaitNotifyTest
  #50 = Utf8               com/multithreding/Question7/WaitNotifyTest
  #51 = Utf8               Code
```

```
  #64 = Utf8               SourceFile
  #65 = Utf8               WaitNotifyTest.java
{
  public com.multithreding.Question7.WaitNotifyTest();
    descriptor: ()V
    flags: (0x0001) ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method java/lang/Object."<init>":()V
         4: return
      LineNumberTable:
        line 3: 0
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0       5     0  this   Lcom/multithreding/Question7/WaitNotifyTest;

  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
    flags: (0x0009) ACC_PUBLIC, ACC_STATIC
    Code:
      stack=4, locals=5, args_size=1
         0: new           #7                  // class com/multithreding/Question7/Message
         3: dup
         4: ldc           #9                  // String process it
         6: invokespecial #11                 // Method com/multithreding/Question7/Message."<init>":(Ljava/lang/String;)V
         9: astore_1
        10: new           #14                 // class com/multithreding/Question7/Waiter
        13: dup
        14: aload_1
        15: invokespecial #16                 // Method com/multithreding/Question7/Waiter."<init>":(Lcom/multithreding/Question7/Message;)V
        18: astore_2
        19: new           #19                 // class java/lang/Thread
        22: dup
        23: aload_2
        24: ldc           #21                 // String waiter
        26: invokespecial #23                 // Method java/lang/Thread."<init>":(Ljava/lang/Runnable;Ljava/lang/String;)V
        29: invokevirtual #26                 // Method java/lang/Thread.start:()V
        32: new           #14                 // class com/multithreding/Question7/Waiter
        35: dup
        36: aload_1
        37: invokespecial #16                 // Method com/multithreding/Question7/Waiter."<init>":(Lcom/multithreding/Question7/Message;)V
        40: astore_3
        41: new           #19                 // class java/lang/Thread
        44: dup
        45: aload_3
        46: ldc           #29                 // String waiter1
        48: invokespecial #23                 // Method java/lang/Thread."<init>":(Ljava/lang/Runnable;Ljava/lang/String;)V
        51: invokevirtual #26                 // Method java/lang/Thread.start:()V
        54: new           #31                 // class com/multithreding/Question7/Notifier
        57: dup
        58: aload_1
        59: invokespecial #33                 // Method com/multithreding/Question7/Notifier."<init>":(Lcom/multithreding/Question7/Message;)V
        62: astore        4
        64: new           #19                 // class java/lang/Thread
        67: dup
        68: aload         4
        70: ldc           #34                 // String notifier
        72: invokespecial #23                 // Method java/lang/Thread."<init>":(Ljava/lang/Runnable;Ljava/lang/String;)V
        75: invokevirtual #26                 // Method java/lang/Thread.start:()V
        78: getstatic     #36                 // Field java/lang/System.out:Ljava/io/PrintStream;
        81: ldc           #42                 // String All the threads are started
        83: invokevirtual #44                 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
        44: dup
        45: aload_3
        46: ldc           #29                 // String waiter1
        48: invokespecial #23                 // Method java/lang/Thread."<init>":(Ljava/lang/Runnable;Ljava/lang/String;)V
        51: invokevirtual #26                 // Method java/lang/Thread.start:()V
        54: new           #31                 // class com/multithreding/Question7/Notifier
        57: dup
        58: aload_1
        59: invokespecial #33                 // Method com/multithreding/Question7/Notifier."<init>":(Lcom/multithreding/Question7/Message;)V
        62: astore        4
        64: new           #19                 // class java/lang/Thread
        67: dup
        68: aload         4
        70: ldc           #34                 // String notifier
        72: invokespecial #23                 // Method java/lang/Thread."<init>":(Ljava/lang/Runnable;Ljava/lang/String;)V
        75: invokevirtual #26                 // Method java/lang/Thread.start:()V
        78: getstatic     #36                 // Field java/lang/System.out:Ljava/io/PrintStream;
        81: ldc           #42                 // String All the threads are started
        83: invokevirtual #44                 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
        86: return
      LineNumberTable:
        line 6: 0
        line 7: 10
        line 8: 19
        line 10: 32
        line 11: 41
        line 13: 54
        line 14: 64
        line 15: 78
        line 16: 86
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0      87     0  args   [Ljava/lang/String;
           10      77     1   msg   Lcom/multithreding/Question7/Message;
           19      68     2 waiter   Lcom/multithreding/Question7/Waiter;
           41      46     3 waiter1   Lcom/multithreding/Question7/Waiter;
           64      23     4 notifier   Lcom/multithreding/Question7/Notifier;

ourceFile: "WaitNotifyTest.java"
```

Java compile file consist following structure

Screenshot 1 shows all this details.

magic,minor_version,major_version → specifies information about the version of the class and the version of the JDK this class was compiled for.

constant_pool → The JVM maintains a per-type constant pool, a run time data structure that **is** similar to a symbol table although it contains more data. Byte codes in Java require data, often this data is too large to store directly in the byte codes, instead it is stored in the constant pool and the byte code contains a reference to the constant pool.

access_flags → provides the list of modifiers for this class.

this_class→ index into the constant_pool providing the fully qualified name of this class

super_class→ index into the constant_pool providing a symbolic reference to the super class

interfaces→ array of indexes into the constant_pool providing a symbolic references to all interfaces that have been implemented.

Fields→ array of indexes into the constant_pool giving a complete description of each field.

Attributes → array of different value that provide additional information about the class including any annotations with RetentionPolicy.CLASS or RetentionPolicy.RUNTIME

Methods → array of indexes into the constant_pool giving a complete description of each method signature, if the method is not abstract or native then the bytecode is also present. Each method contains 4 areas

- signature and access flags
- byte code
- LineNumberTable – this provides information to a debugger to indicate which line corresponds to which byte code instruction, for example line 6 in the Java code corresponds to byte code 0 in the sayHello method and line 7 corresponds to byte code 8.
- LocalVariableTable – this lists all local variables provided in the frame, in both examples the only local variable is this.

Please find following screenshot for reference.

```
public static void main(java.lang.String[]);
  descriptor: ([Ljava/lang/String;)V
  flags: (0x0009) ACC_PUBLIC, ACC_STATIC
  Code:
    stack=4, locals=5, args_size=1
       0: new           #7                  // class com/multithreding/Question7/Message
       3: dup
       4: ldc           #9                  // String process it
       6: invokespecial #11                 // Method com/multithreding/Question7/Message."<init>":(Ljava/lang/String;)V
       9: astore_1
      10: new           #14                 // class com/multithreding/Question7/Waiter
      13: dup
      14: aload_1
      15: invokespecial #16                 // Method com/multithreding/Question7/Waiter."<init>":(Lcom/multithreding/Question7/Message;)V
      18: astore_2
      19: new           #19                 // class java/lang/Thread
      22: dup
      23: aload_2
      24: ldc           #21                 // String waiter
      26: invokespecial #23                 // Method java/lang/Thread."<init>":(Ljava/lang/Runnable;Ljava/lang/String;)V
      29: invokevirtual #26                 // Method java/lang/Thread.start:()V
      32: new           #14                 // class com/multithreding/Question7/Waiter
      35: dup
      36: aload_1
      37: invokespecial #16                 // Method com/multithreding/Question7/Waiter."<init>":(Lcom/multithreding/Question7/Message;)V
      40: astore_3
      41: new           #19                 // class java/lang/Thread
      44: dup
      45: aload_3
      46: ldc           #29                 // String waiter1
      48: invokespecial #23                 // Method java/lang/Thread."<init>":(Ljava/lang/Runnable;Ljava/lang/String;)V
      51: invokevirtual #26                 // Method java/lang/Thread.start:()V
      54: new           #31                 // class com/multithreding/Question7/Notifier
      57: dup
      58: aload_1
      59: invokespecial #33                 // Method com/multithreding/Question7/Notifier."<init>":(Lcom/multithreding/Question7/Message;)V
      62: astore        4
      64: new           #19                 // class java/lang/Thread
      67: dup
      68: aload         4
      70: ldc           #34                 // String notifier
      72: invokespecial #23                 // Method java/lang/Thread."<init>":(Ljava/lang/Runnable;Ljava/lang/String;)V
      75: invokevirtual #26                 // Method java/lang/Thread.start:()V
      78: getstatic     #36                 // Field java/lang/System.out:Ljava/io/PrintStream;
      81: ldc           #42                 // String All the threads are started
      83: invokevirtual #44                 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
      86: return
    LineNumberTable:
```

Following operand code used in this report

new → Create new object

dup. → Duplicate the top operand stack value

ldc → push item from run-time constant pool

invokespecial → Invoke instance method; special handling for superclass, private, and instance initialization method invocations

astore_1. → Store reference into local variable

aload_1-→ Load reference from local variable

invokevirtual → Invoke a class (static) method

getstatic → Get static field from class

8. Write a Java Test program that creates a Thread and has this "calculate" method.

```
public int calculate (int i, int j, int k) {
        int a = i * j * k + k * i + 10 ;
        return a;
}
```

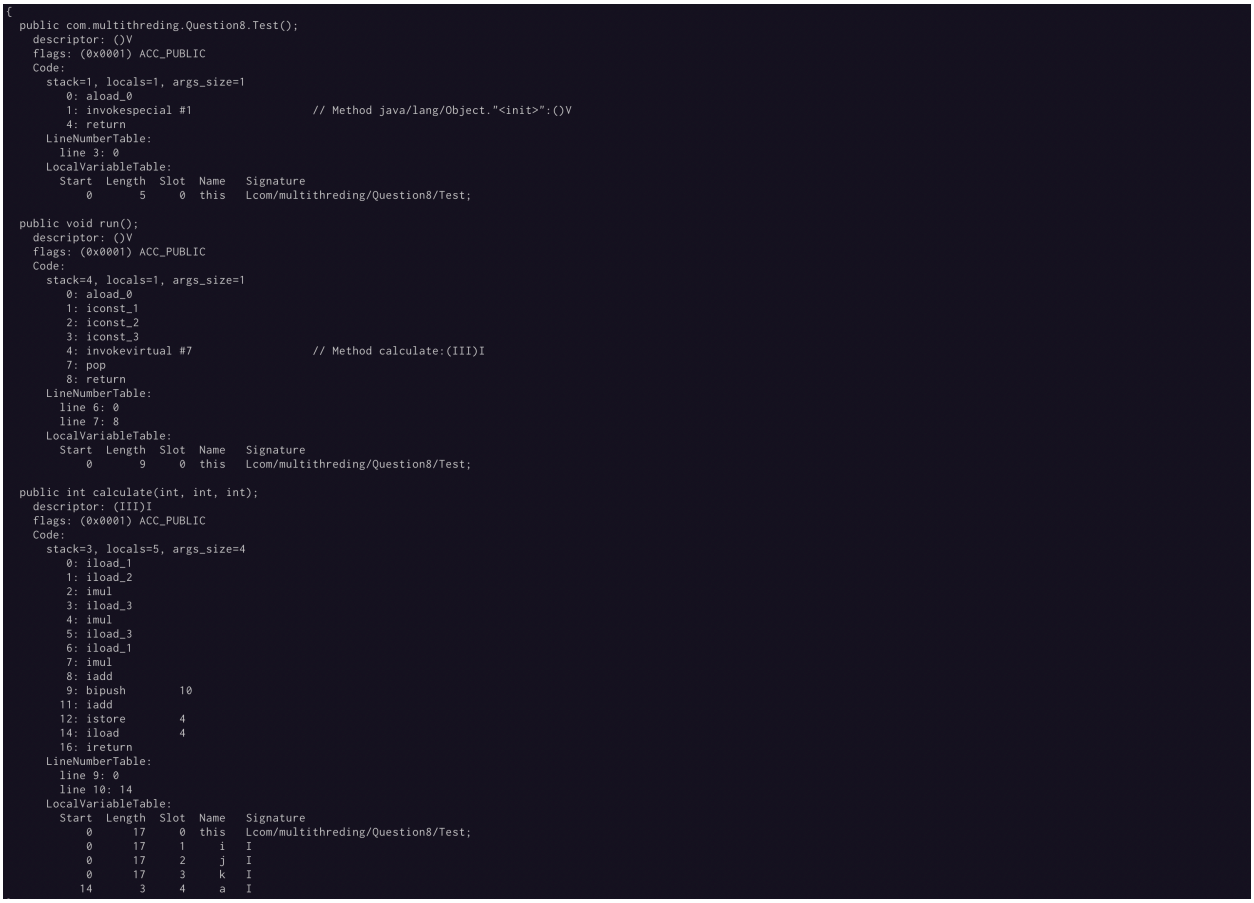
A) Compile Java code

B) Run "Javap -l -v -c Test.class"

C) Discuss the report, the output, Local Variable Array and Operand Stack

D) Provide a drawing to show JVM step-by-step Execution of Local Variable Array (LVM) and Operand Stack

Report screenshot



```
{
  public com.multithreding.Question8.Test();
    descriptor: ()V
    flags: (0x0001) ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method java/lang/Object."<init>":()V
         4: return
      LineNumberTable:
        line 3: 0
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0       5     0  this   Lcom/multithreding/Question8/Test;

  public void run();
    descriptor: ()V
    flags: (0x0001) ACC_PUBLIC
    Code:
      stack=4, locals=1, args_size=1
         0: aload_0
         1: iconst_1
         2: iconst_2
         3: iconst_3
         4: invokevirtual #7                  // Method calculate:(III)I
         7: pop
         8: return
      LineNumberTable:
        line 6: 0
        line 7: 8
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0       9     0  this   Lcom/multithreding/Question8/Test;

  public int calculate(int, int, int);
    descriptor: (III)I
    flags: (0x0001) ACC_PUBLIC
    Code:
      stack=3, locals=5, args_size=4
         0: iload_1
         1: iload_2
         2: imul
         3: iload_3
         4: imul
         5: iload_3
         6: iload_1
         7: imul
         8: iadd
         9: bipush        10
        11: iadd
        12: istore        4
        14: iload         4
        16: ireturn
      LineNumberTable:
        line 9: 0
        line 10: 14
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0      17     0  this   Lcom/multithreding/Question8/Test;
            0      17     1     i   I
            0      17     2     j   I
            0      17     3     k   I
           14       3     4     a   I
```

According to give report we can see that there are 5 local variables mentioned in local variable table

```
      line 5: 0
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0       5     0  this   Lcom/multithreding/Question8/Test;
```

LOCAL VARIABLE ARRAYA

Index	Variable
0	this
1	Int i
2	Int j
3	Int k
4	Int a

Operand Stack

0: iload_1. → Load the value of index 1 which is **i**
1: iload_2. → // Load the value of index 2 which is **j**
2: imul → perform multiplication
3: iload_3 → Load the value of index 3 which is **k**
4: imul. → perform multiplication
5: iload_3 → Load the value of index 3 which is **k**
6: iload_1 → Load the value of index 1 which is **i**
7: imul → perform multiplication
8: iadd → perform addition
9: bipush 10 → push the 10 into operand stack
11: iadd → perform addition

12: istore 4 → store the result value at index 4
14: iload 4 → Load the value of index 4 which is **j**
16: ireturn → return that value

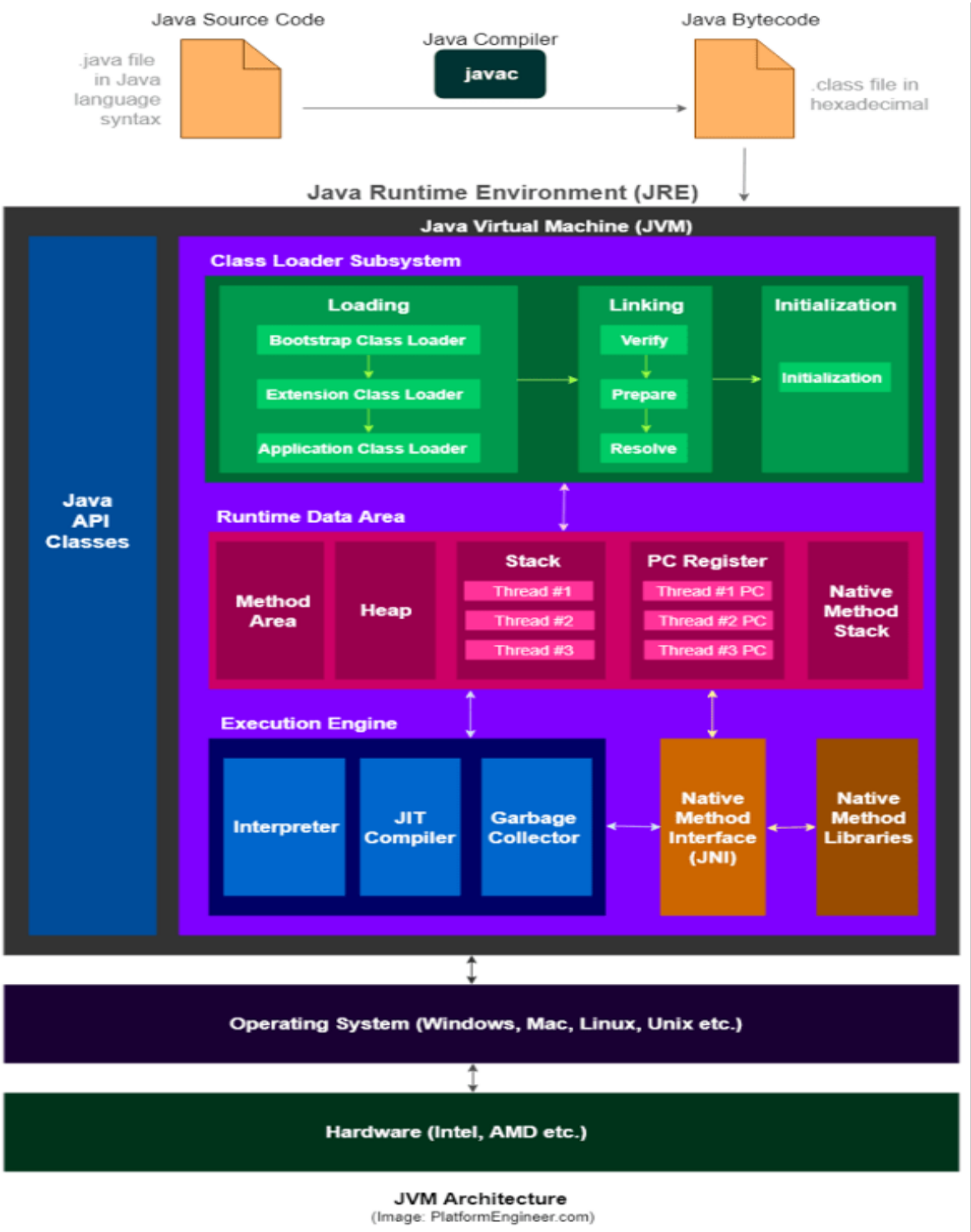
Operand stack operation :

Before Loading	iload_1	iload_2	imul	iload_3	imul	iload_3	iload_1	imul	iadd	bi
	i	i	I*j	k	I*j*k	k	k	K*i	I*j*k + K*i	10
		j					i			

iadd	istore	ireturn
i * j * k + k * i + 10	Store a in LVA at 4th index	a

Assignment:4

1. Explain every element in this JVM architecture, and how element layers are tied to each other?



JVM Architecture
(Image: PlatformEngineer.com)

1) Firstly Java compiler will compile the source code and generate the .class file which is further send to JVM
2) Once the JVM receive this file it perform following steps
3) It will load the class file in main memory which is know as dynamic class loading using following class loader
    - Bootstrap classloader
        - Responsible for load the class from bootstrap classpath
    - Extension classloader
        - Responsible for load the classes from ext folder
    - Application classloader
        - Responsible for load application level classpath
4) Inside the linking phase JVM verify the weather the generated bytecode is proper or not then allocate the memory for static variables and resolve the symbolic memory reference with original reference from memory area.

- After all this process Run time data area comes into picture. It is divided into five major parts .
    1) **Method area** :
        - All the class-level data will be stored here, including static variables.
        - There is only one method area per JVM, and it is a shared resourcw

2) **Heap Area** –
   - All the Objects and their corresponding instance variables and arrays will be stored here.
   - There is also one Heap Area per JVM. Since the Method and Heap areas share memory for multiple threads, the data stored is not thread-safe

3) **Stack Area** –
   - For every thread, a separate runtime stack will be created. For every method call, one entry will be made in the stack memory which is called Stack Frame.
   - All local variables will be created in the stack memory

4) **PC Registers** –
   - Each thread will have separate PC Registers, to hold the address of current executing instruction
   - Once the instruction is executed the PC register will be updated with the next instruction.

5) **Native Method stack**s – Native Method Stack holds native method information. For every thread, a separate native method stack will be created.

- Once the bytecode assigned to run time data area , execution engine start executing bytecodes. JVM execution engine contains following components

1) **Interpreter** –
   - The interpreter interprets the bytecode faster but executes slowly.

- The disadvantage of the interpreter is that when one method is called multiple times, every time a new interpretation is required.

2) **JIT Compiler** –

- The JIT Compiler neutralizes the disadvantage of the interpreter.
- The Execution Engine will be using the help of the interpreter in converting byte code, but when it finds repeated code it uses the JIT compiler, which compiles the entire bytecode and changes it to native code.
- This native code will be used directly for repeated method calls, which improve the performance of the system.
-

3) **Garbage Collector**:

- Collects and removes unreferenced objects. Garbage Collection can be triggered by calling System.gc(), but the execution is not guaranteed.
- Garbage collection of the JVM collects the objects that are created.

- JVM **also contains JNI and native method library**. **JNI will interact with native method library and provides the native library required for the** execution.

2. Explain:
Implicit lock versus Explicit lock

-> Java provides a built-in locking mechanism for enforcing atomicity: the synchronized block.

-> Every Java object can implicitly act as a lock for purposes of synchronization; these built-in locks are called intrinsic locks or monitor locks.

-> The lock is automatically acquired by the executing thread before entering a synchronized block and automatically released when control exits the synchronized block, whether by the normal control path or by throwing an exception out of the block.
-> The only way to acquire an intrinsic lock is to enter a synchronized block or method guarded by that lock

```
synchronized (someObject) {
  // do some stuff
}

someObject.lock.acquire();
try {
  // do some stuff
} finally {
  someObject.lock.release();
}
```

-> On the other hand Java has Lock interface which allows to implement abstract locking operations.
->Unlike Implicit locking, Lock offers a choice of unconditional, polled, timed, and interruptible lock acquisition, and all lock and unlock operations are explicit. Lock implementations must provide the same memory

```
Lock lock = new ReentrantLock();
...
lock.lock(); try {
   // update object state
   // catch exceptions and restore invariants if necessary
} finally {
lock.unlock(); }
```

➔ main difference between implicit lock and explicit lock

that it is not possible to interrupt a thread waiting to acquire a lock, or to attempt to acquire a lock without being willing to wait for it forever

➔ Having a timeout trying to get access to a synchronized block is not possible. Using

Lock.tryLock(long timeout, TimeUnit timeUnit), it is possible.

➔ The synchronized block must be fully contained within a single method. A Lock can have it's calls to lock() and unlock() in separate methods

Class lock versus Object lock

Class level Lock:

➔ Class level lock prevents multiple threads to enter in synchronized block in any of all available instances of the class on runtime.

➔ Class level locking should always be done to make static data thread safe. As we know that static keyword associate data of methods to class level, so use locking at static fields or methods to make it on class level

```
//Acquire lock on .class reference
      synchronized (DemoClass.class)
      {
         //other thread safe code
      }

//Lock object is static
  private final static Object lock = new Object();
```

```
synchronized (lock)
{
   //other thread safe code
}
```

Object level Lock:

➔ Object level lock is mechanism when we want to synchronize a non-static method or non-static code block such that only one thread will be able to execute the code block on given instance of the class. This should always be done to make instance level data thread safe.

```
// Using this
synchronized (this)
      {
         //other thread safe code
      }
```

OR

```
// using private object instance
private final Object lock = new Object();
   public void test(){
      synchronized (lock)
      {
         //other thread safe code
```

```
    }
}
```

Call Stack

➔ The call stack is what a program uses to keep track of method calls. The call stack is made up of stack frames—one for each method call

➔ Each thread has it's own call stack. Let's consider the following example of main thread

```
Public class example {
Public static void main(String args[]){

Method1();
Method2();
Method3();
Method4();
}
}
```

So call stack for aforementioned program would be like this

Method4()
Method3()
Method2()

Method1()
Main()

Stack Memory

-> Java Stack memory is used for execution of a thread and it contains method specific values and references to other objects in Heap.

-> For every thread, a separate runtime stack will be created. For every method call, one entry will be made in the stack memory which is called Stack Frame. All local variables will be created in the stack memory.

-> The stack area is thread-safe since it is not a shared resource.

-> The Stack Frame is divided into three subentities:

Local Variable Array – Related to the method how many local variables are involved and the corresponding values will be stored here.

Operand stack – If any intermediate operation is required to perform, operand stack acts as runtime workspace to perform the operation.

Frame data – All symbols corresponding to the method is stored here. In the case of any exception, the catch block information will be maintained in the frame data.

Heap Space

➔ All the Objects and their corresponding instance variables and arrays will be stored in heap space.

➔ There is one Heap Area per JVM. Since the Method and Heap areas share memory for multiple threads, the data stored is not thread-safe

➔ JVM heap is divided into 2 parts

Young generation - This is reserved for containing newly allocated objects. Young Gen includes three parts — **Eden Memory and two Survivor Memory spaces (S0, S1).** Most of the **newly created** objects **goes Eden space**. When Eden space is filled with objects, **Minor GC** is performed, and all the survivor objects are moved to one of the survivor spaces. Minor GC also checks the survivor objects and move them to the other survivor space. So, at a time, one of the survivor spaces is always empty.

Old Generation- This is reserved for containing long lived objects that could survive after many rounds of Minor GC. When Old Gen space is full, Major GC (a.k.a. Old Collection) is performed (usually takes longer time)

String Pool, give example

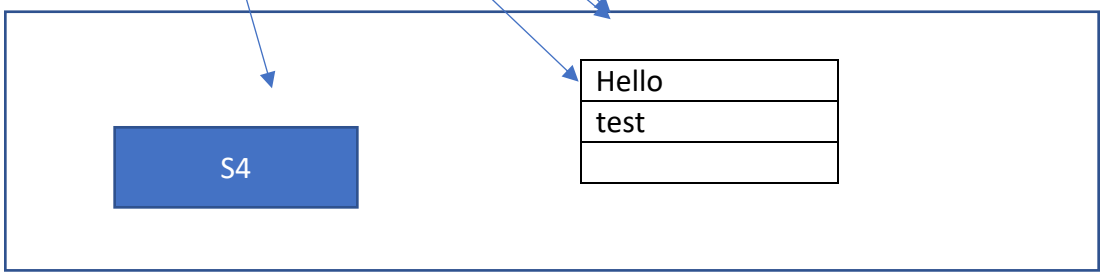
➔ String Pool is a storage area in Java heap. The JVM performs some steps while initializing string literals to increase performance and decrease memory overhead. To decrease the number of String objects created in the JVM, the String class keeps a pool of strings.

➔ Each time a string literal is created, the JVM checks the string literal pool first. If the string already exists in the string pool, a reference to the pooled instance returns. If the string does not exist in the pool, a new String object initializes and is placed in the pool

Example

String s1 = “Hello”
String s2 = “Hello”
String s3 = “Test”
String s4 = new String(“test2”)

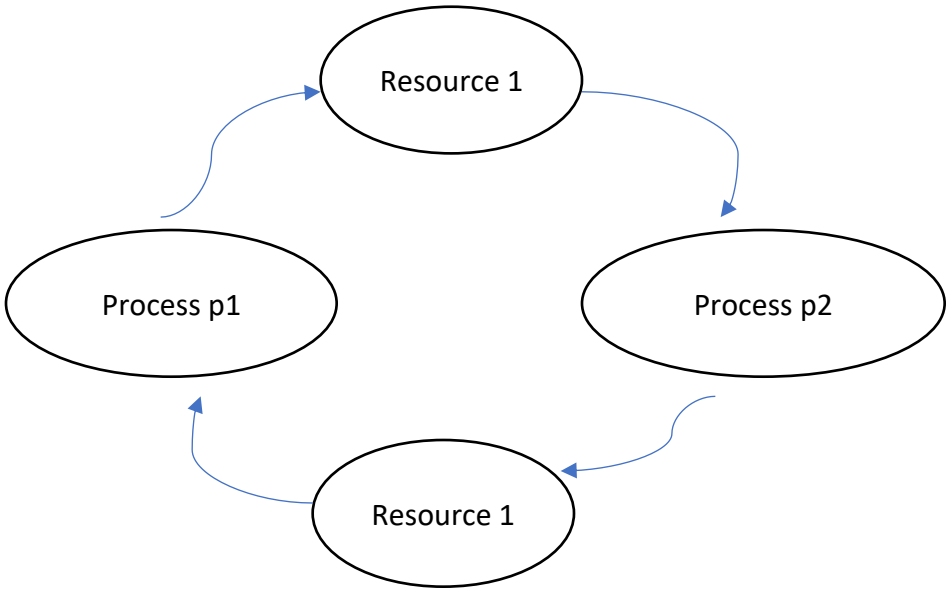
Heap Area



Deadlock, Starvation, Race condition, provide examples

Deadlock - Deadlock describes a situation where two or more
threads are blocked forever, waiting for each other

Example



➔ In Above example process p1 try to acquire lock on resource 1 but resource 1 is already locked by process

p2 and process p2 try to acquire lock on resource r2 but it already locked by process p1. Now both the processes p1 and p2 are interdependent. This kind of situation known as Deadlock.

Starvation – Starvation describes a situation where a **thread is unable to gain regular access to shared resources and** is unable to make progress. This happens when shared resources are made **unavailable for long periods by "greedy" threads**

➔ For example, 5 thread running concurrently and for each thread we are setting some priority using **setPriority().** Method. If we are setting very high priority for three threads and low priority for 2 threads, then execution of low priority thread barely occurs and they hardly get chance to access the shared resources.

Race Condition - A race condition occurs when two or more threads can access shared data and they try to change it at the same time. Execution order of threads are dependent on Thread scheduling algorithm, you don't know the order in which the threads will attempt to access the shared data. Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e. both threads are "racing" to access/change the data

For Example Thread A reads variable count shared object into its CPU cache, and Thread B does the same but into a different CPU cache. Now thread A adds one to count, and

thread B does the same. Now var1 has been incremented two times, once in each CPU cache. If these increments had been carried out sequentially, the variable count would be incremented twice and had the original value + 2 written back to main memory. However, the two increments have been carried out concurrently without proper synchronization. Regardless of which thread A and B that writes its updated version of count back to main memory, the updated value will only be 1 higher than the original value, despite the two increments. Please find the reference code

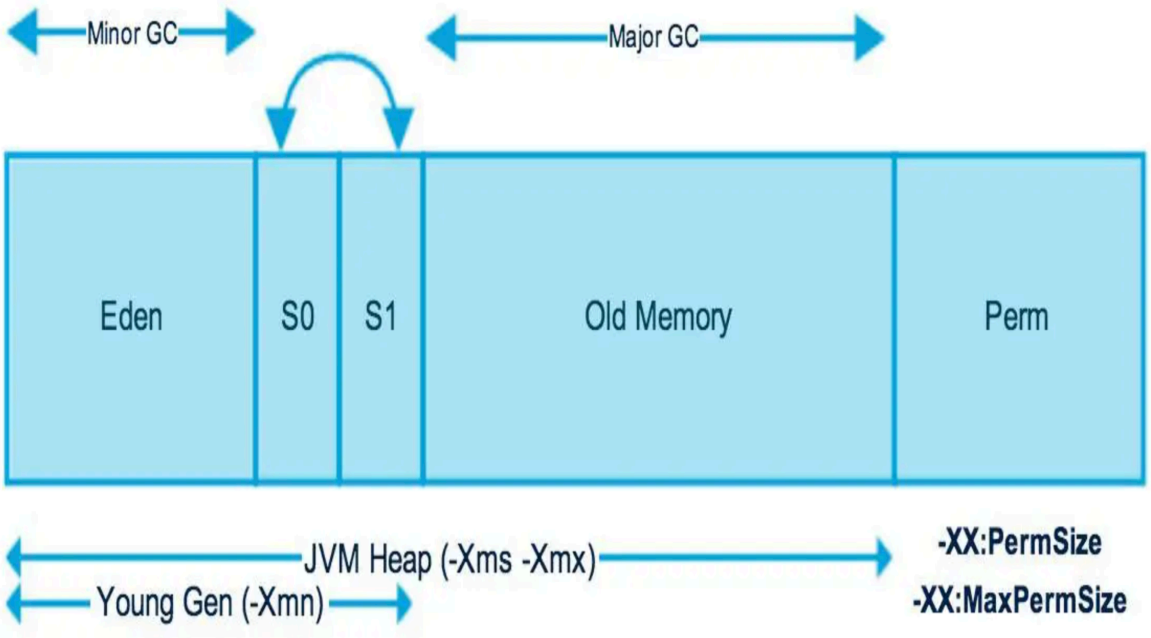
```
Public class RaceCondition implements Runnable{
    Private int count= 0;
        @override
        Public void run(){
        Count++;
        }

    }
```

How does Garbage collector works in Java?

- In Java every time we create a new object it will acquire some space in JVM heap area.Garbage collector will help us to reclaimed the unused memory
- So basically when we creates a new object it will occupy the some space in JVM heap area and after **once that object is no longer referenced by the program**, **the heap space it occupies can be recycled so that the space is made available for subsequent new objects**.
- The garbage collector **must somehow determine which objects are no longer referenced by the program and make available the heap space occupied by such unreferenced objects**.
- In the process of freeing unreferenced objects, the garbage collector must run any finalizers of objects being freed.

How does JVM manages garbage collector?

- Whenever we are creating new object JVM allocate separate space to that object inside the heap memory area.
- Now JVM heap space is divided into two parts
  1) Young Generation
  2) Old Generation



- The Young Generation is a part of the heap reserved for the allocation of new objects. When the nursery

becomes full, garbage is collected by running a special young collection, where all the objects that have lived long enough in the nursery are promoted (moved) to the old space, thus freeing up the nursery for more object allocation. This garbage collection is called Minor GC. The Young Generation is divided into three parts – Eden Memory and two Survivor Memory spaces.

- Most of the newly created objects are located in the Eden Memory space. When **Eden space is filled with objects**, **Minor GC is performed and all the survivor objects are moved to one of the survivor spaces Minor GC also checks the survivor objects and moves them to the other survivor space. So at a time, one of the survivor space is always empty Objects that have survived many cycles of GC, are moved to the old generation memory space.**
- **When the old generation becomes full, garbage is collected there and the process is called as old collection. Old generation memory contains the objects that are long-lived and survived after many rounds of Minor GC. Usually, garbage collection is performed in Old generation memory when it's full. Old generation garbage collection is called as "Major GC" and usually takes longer.**
- **The reasoning behind a Young Generation is that most objects are temporary and short-lived. A young collection is designed to be swift at finding newly allocated objects that are still alive and moving them away from the nursery. Typically, a young collection**

frees a given amount of memory much faster than an old collection or a garbage collection of a single-generational heap (a heap without a nursery).

3. Search internet to find all Threadsafe and NotThreadsafe Java HashMap,
HashTable, collections. For each collection:

a) List three methods in each collection describing what is the intent of collection,
when it is useful to use this collection, and describe the selected methods.

Collection are useful when we are working with the same type of objects

Thread Safe classes in java collection frameworks

- Stack
- Vector
- HashTable
- Blocking Queue
- ConcurrentMap
- ConcurrentNavigableMap

Stack

push(E item)→ Pushes an item onto the top of this stack.
pop()→Removes the object at the top of this stack and returns that object as the value of this function.
peek()→ Looks at the object at the top of this stack without removing it from the stack.

Vector

add(E e)→Appends the specified element to the end of this Vector.
firstElement()→Returns the first component (the item at index 0) of this vector.
size()→Returns the number of components in this vector.

HashTable

clear()→Clears this hashtable so that it contains no keys.
get(Object key)→Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
remove(Object key)→Removes the key (and its corresponding value) from this hashtable.

Blocking Queue

put(E e)→Inserts the specified element into this queue, waiting if necessary for space to become available.
contains(Object o)→Returns true if this queue contains the specified element.
take()→Retrieves and removes the head of this queue, waiting if necessary until an element becomes available.

ConcurrentHashMap

elements()→Returns an enumeration of the values in this table.

keySet()→Returns a Set view of the keys contained in this map.

size()→Returns the number of key-value mappings in this map.

Not Thread safe

- ArrayList
- HashMap
- LinkedList
- HashSet
- Linked hashset
- Tree Hashset

ArrayList

isEmpty()→Returns true if this list contains no elements.

contains(Object o)→Returns true if this list contains the specified element.

clear()→Removes all of the elements from this list.

HashMap

put(K key, V value)→Associates the specified value with the specified key in this map.

remove(Object key)→Removes the mapping for the specified key from this map if present.

containsKey(Object key)→Returns true if this map contains a mapping for the specified key.

LinkedList

getFirst()→Returns the first element in this list.

peekFirst()→Retrieves, but does not remove, the first element of this list, or returns null if this list is empty.

set(int index, E element)→Replaces the element at the specified position in this list with the specified element.

HashSet

contains(Object o)→Returns true if this set contains the specified element.

iterator()→Returns an iterator over the elements in this set.

add(E e)→Adds the specified element to this set if it is not already present.

LinkedhashSet

remove(Object o)→Removes the specified element from this set if it is present.

size()→Returns the number of elements in this set (its cardinality).

clear()→Removes all of the elements from this set.

b) What is Collections class? Most methods in this class are static, name five meth-ods.

Java collections class

- This class consists exclusively of static methods that operate on or return collections
- class contains polymorphic algorithms that operates on collection

known as "Wrapper"

Class contains three static field

List→The empty list (immutable). This list is serializable.

map → The empty map (immutable). This map is serializable.

set → The empty set (immutable). This set is serializable.

Collections class contains lots of static methods

<u>Useful methods in Collections class</u>

1) public static <T> Collection<T> synchronizedCollection(Collection<T> c)

Returns a synchronized (thread-safe) collection backed by the specified collection. In order to guarantee serial access, it is critical that all access to the backing collection is accomplished through the returned collection.

example:

```
 Collection c =
Collections.synchronizedCollection(myCollection);
    ...
 synchronized (c) {
    Iterator i = c.iterator(); // Must be in the synchronized block
    while (i.hasNext())
      foo(i.next());
 }
```

2) public static <T> Set<T> synchronizedSet(Set<T> s)

Returns a synchronized (thread-safe) set backed by the specified set.

example

```
      Set s = Collections.synchronizedSet(new HashSet());
    ...
 synchronized (s) {
    Iterator i = s.iterator(); // Must be in the synchronized block
    while (i.hasNext())
       foo(i.next());
 }
```

3) public static <T> List<T> synchronizedList(List<T> list)

Returns a synchronized (thread-safe) list backed by the specified list.

example :

```
        List list = Collections.synchronizedList(new
ArrayList());
    ...
 synchronized (list) {
    Iterator i = list.iterator(); // Must be in synchronized block
    while (i.hasNext())
       foo(i.next());
 }
```

4) public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m)

Returns a synchronized (thread-safe) map backed by the specified map.

example :

```
Map m = Collections.synchronizedMap(new HashMap());
    ...
 Set s = m.keySet();  // Needn't be in synchronized block
    ...
 synchronized (m) {  // Synchronizing on m, not s!
    Iterator i = s.iterator(); // Must be in synchronized block
```

```
    while (i.hasNext())
       foo(i.next());
 }
```

5) public static <T> void sort(List<T> list,
Comparator<? super T> c)

It is used to sort the elements present in the specified list of Collection in ascending order

example :

```
ArrayList<String> al = new ArrayList<String>();
/* Collections.sort method is sorting the
elements of ArrayList in ascending order. */
Collections.sort(al)
```

4. In Homework3, you created 50 student threads and one GraderThread. Change the program to use Explicit locking instead of implicit locking. Note: see problem descrip-tion in hw3, problem-3 (a) (b) (c) (d), and all the requirements for that problem must be implemented in this problem using explicit locking.

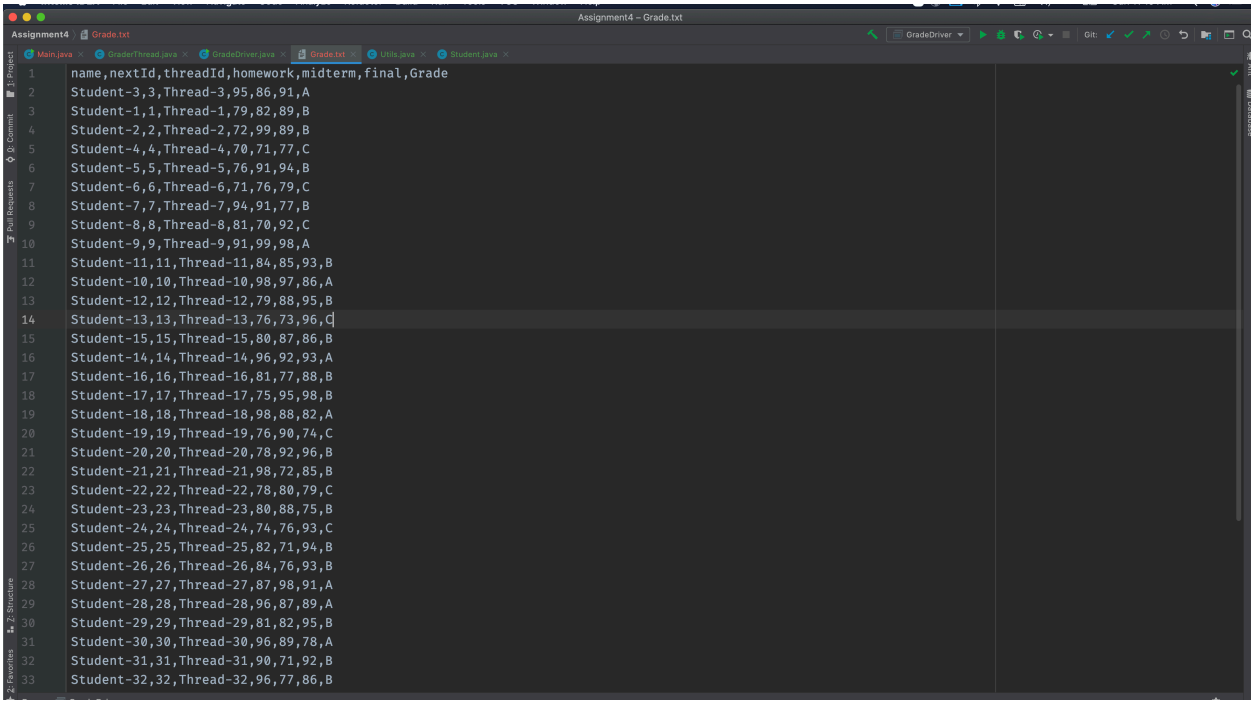
Approach :

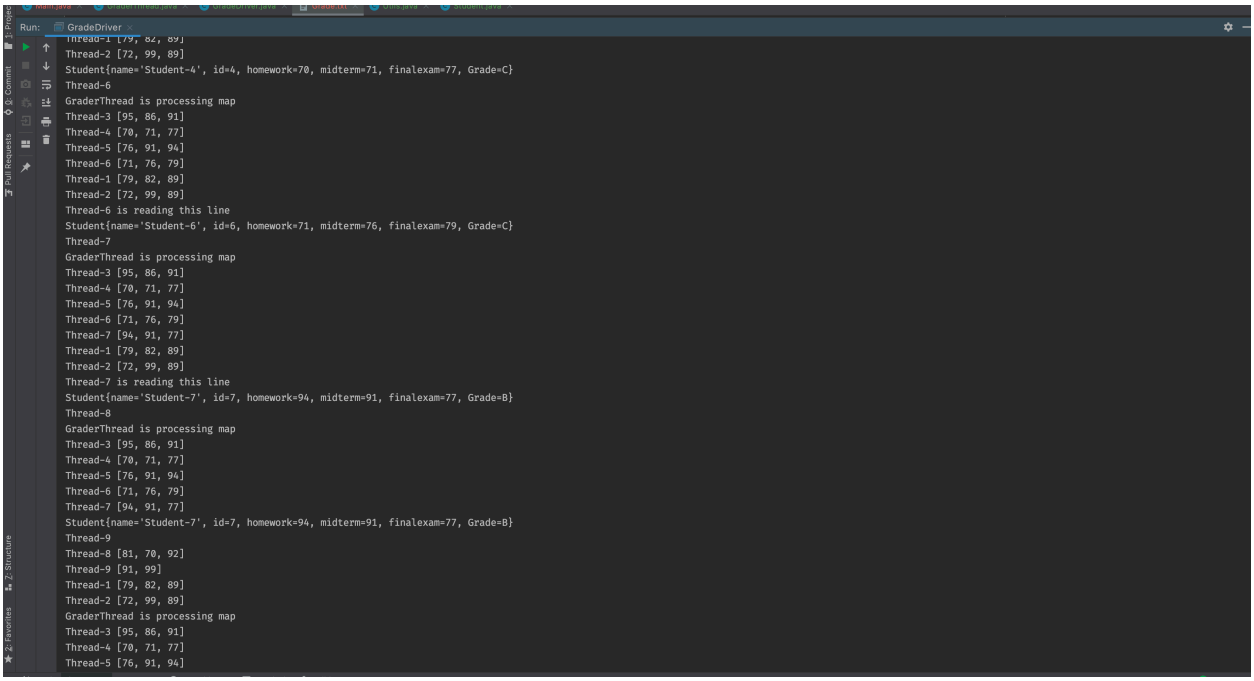
→ As we have already implemented the logic of the give program using synchronized block now I am eliminating

synchronized block (implicit locking). Instead of that **I've used the ReentrantLock lock class which implements the lock interface**

→ To replace the notifyAll and wait method I have used the condition class implementation in which we have methods such as **await()** and **signalAll()** which performs the similar task.

→ Please find the following output screenshot using Implicit locking. To review detail implementation please check attached solution file





5. Design a program that creates 10 Student objects with each student is size of 20 bytes. The JVM heap size is 240 bytes. Your program design should consider scenario where five objects become Unreferenced for garbage collection. Provide the design cri-terion for Minor GC and Major GC. You should provide the detail design for JVM managing garbage collector memory management.

- The JVM heap size determines how often and how long the VM spends collecting garbage. An acceptable rate for garbage

collection is application-specific and should be adjusted after analyzing the actual time and frequency of garbage collections.
- If **we set a large heap size, full garbage collection is slower**, but it **occurs less frequently. If** you set **your heap size in accordance with your memory needs, full garbage collection is faster, but occurs more frequently.**
->In our application we know that we need to set the Heap size to 240 bytes. So we can specify those setting using following command.

java -Xms240m -Xmx240m -XX:NewRatio=2 -XX:SurvivorRatio=12

-XmsSetting — initial Heap size
-XmxSetting — maximum Heap size
-XX:NewRatio – specify the ration between new and old generation
-XX:SurvivorRatio - ratio between eden and a survivor space

<table>
<tr><td colspan="4"><----------240 bytes----------></td></tr>
<tr><td colspan="3">----------120 bytes----------</td><td>----------120 bytes---------</td></tr>
<tr><td colspan="3">Young Generation</td><td></td></tr>
<tr><td>Eden Space
100 Bytes</td><td>S1</td><td>S2</td><td>Old generation</td></tr>
</table>

	10 Bytes	*10 Bytes*	
New Student object will resides here			old student object will resides here

Explanation

- Here I have specify the ratio between young and old generation is ½ means 120 byte will be allocated to each generation
- The ration between Eden and survivor space is 1/12 means each survivor space will get 10 bytes and Eden space will get 100 bytes.
- **We know that Eden space is the area where newly created object resides and once it will gets full Minor GC will perform and reclaimed the unused memory and move the survivor objects to survivor space so in our application student object will occupy 20 bytes so if we creates the 5 student objects then Eden space area gets full and Minor GC will perform.**
- **To makes sure that Minor GC perform more frequently I have set the Eden space size to 100 bytes for robust performance**
- **If I will increase the younger generation size then the minor GC will perform in less frequency and when**

Older Generation gets full Major GC need to do more work.

Sample Code:

```
Public class student {// ….};
Public class testGC {

Student s1 = new Student();
Student s2 = new Student();
Student s3 = new Student();
s3 = null // unreferenced the s3
Student s4 = new Student();
Student s5 = new Student();
Student s6 = new Student();
// eden space gets full and Mino GC performed and remove reclaimed s3 memory

}
```

6. Synchronized blocks in Java are Reentrant. That is if a Java thread enters a syn-chronized block code, and thereby takes the lock on the monitor object the block is synchronized on, the thread can enter other Java code blocks synchronized on the same monitor object?

a) what is object monitor in this code?

b) Explain how the fol-lowing code works?

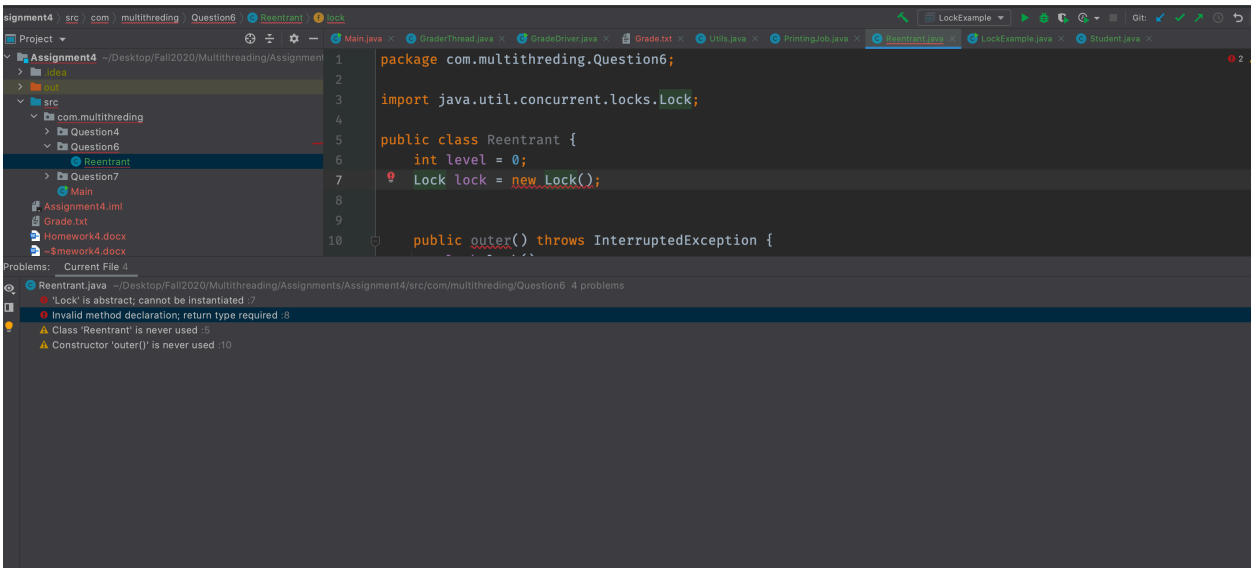
c) Provide different Test scenarios that can successfully execute this code.

```
public class Reentrant {

  int level = 0;
  Lock lock = new Lock();
   public outer() {
     lock.lock();
     inner();
     lock.unlock();
   }

   public synchronized inner(){
       level++;
```

```
            lock.lock();
            try {
            if (level <= 3) {
              inner();
              if (level == 2) {
                      Thread.sleep(1000);
                    }
              } else {
                       Thread.sleep(1000);
             }
            } finally {
                lock.unlock();
                level--;
              }
} }
```

- If we are creating instance of Lock we will get compilation error as Lock is an interface we need to create the instance of class which implements the Lock interface

Error snap:



- To resolved the above error I have now used Reentrant class instance which implements the Lock Interface.

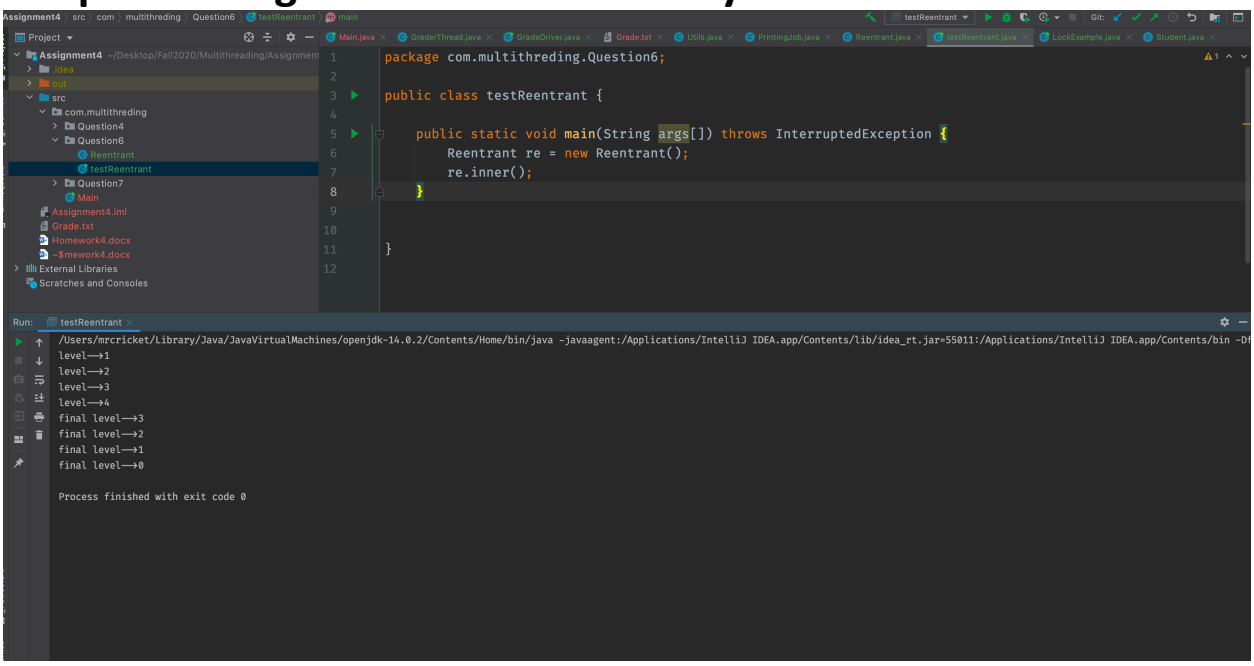
    Lock lock = new ReentrantLock();

- Now we have two different method in reentrant class one is outer() and one is inner(). Outer method acquire the explicit lock and call the inner method and then inner method will recursively call itself and increment the level value till level 4 and then decrement the value in backtracking till 0 in finally block.
- The **Instance of RentrantLock class will work as object monitor here**
- **Here we can observer the Reentrant behavior of ReentrantLock class , thread is acquiring the lock on lock**

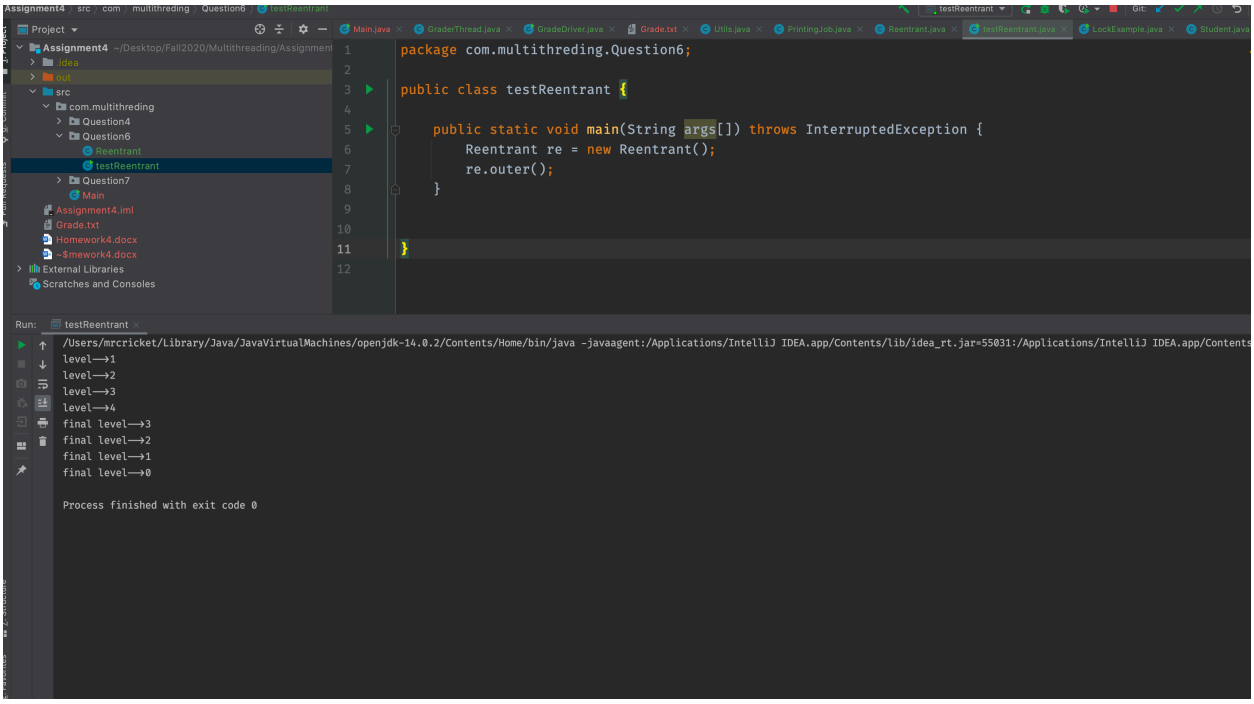
object multiple times and it allows us to hold the lock on same lock instance.

- **Even if we will call the inner method directly from main class it will same output. Inner method is recursively call itself and acquiring the lock on lock instance in each call and then release the lock during the backtracking.**

Output: calling inner method directly



Output: calling outer method directly



7. Consider the following code using Reentrance Lock. How does the Lock work in the following program. Compile and Run.

```
class PrintingJob implements Runnable
{
  private PrinterQueue printerQueue;

  public PrintingJob(PrinterQueue printerQueue)
```

```
  {
    this.printerQueue = printerQueue;
  }

  @Override
  public void run()
  {
   System.out.printf("%s: Going to print a document\n”,
 Thread.currentThread().getName());
    printerQueue.printJob(new Object());
  }
}

class PrinterQueue
{
  private final Lock queueLock = new ReentrantLock();

  public void printJob(Object document)
  {
    queueLock.lock();
    try
    {
      Long duration = (long) (Math.random() * 10000);
      System.out.println(Thread.currentThread().getName() +
": PrintQueue: Printing a Job during " + (duration / 1000) +
" seconds :: Time - " + new Date());
      Thread.sleep(duration);
    } catch (InterruptedException e)
    {
```

```
        e.printStackTrace();
      } finally
      {
        System.out.printf("%s: The document has been
printed\n”,
                Thread.currentThread().getName());
        queueLock.unlock();      }
} }

public class LockExample
{
  public static void main(String[] args)
  {
    PrinterQueue printerQueue = new PrinterQueue();
    Thread thread[] = new Thread[10];
    for (int i = 0; i < 10; i++)
    {
      thread[i] = new Thread(new PrintingJob(printerQueue),
"Thread " + i);
    }
    for (int i = 0; i < 10; i++)
    {
      thread[i].start();
    }
  }
}
```

Explanation:

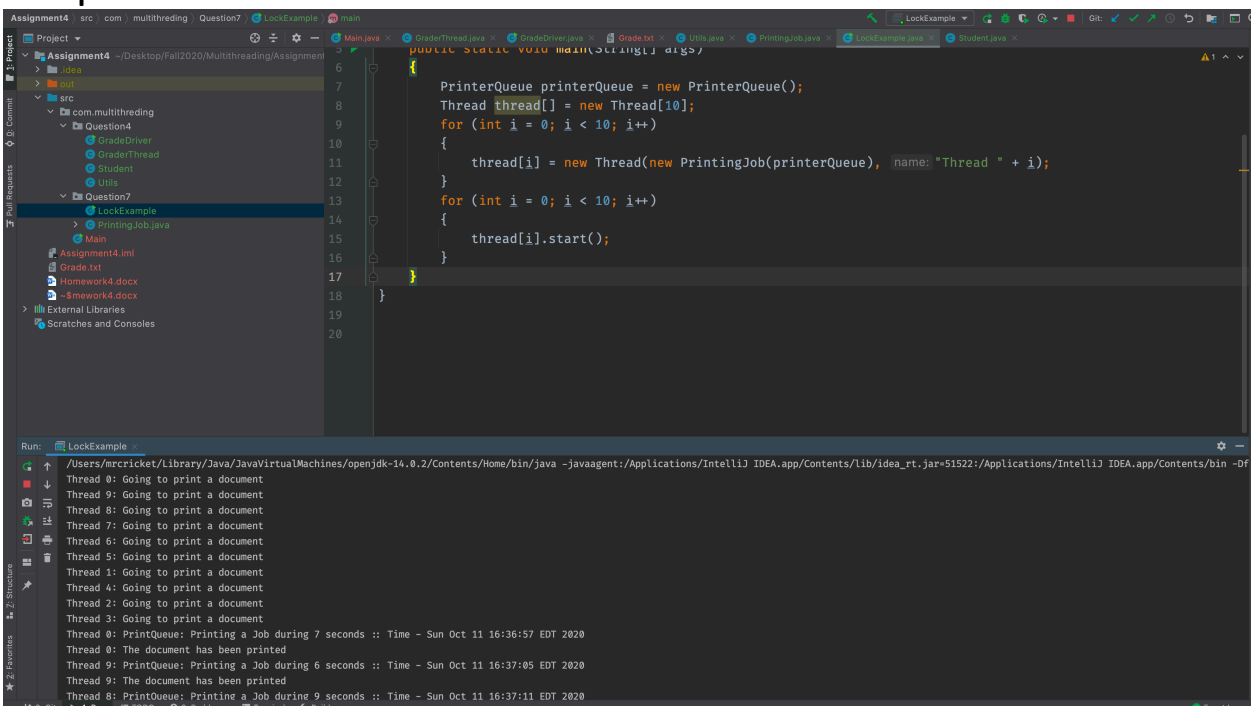
In given program we are creating two different classes 1 form **printingJob** and one is **Printerqueue**. **PrintingJob** class implements the runnable interface and call the **printJob** method of **printerqueue** class.

To handle multiple printing job we have used the explicit lock using **ReentarntLock** class (which is implements the Lock interface)

Inside the printJob method we are. acquiring explicit lock using **lock() method and releasing that lock in finally block using unlock() method.**

This locking mechanism make sure that at a time only one thread can execute the critical part of method. We can implement same thing **using synchronized block as well.**

Output :


Assignment4 › src › com › multithreding › Question7 › LockExample › main
Main.java
GraderThread.java
GradeDriver.java
Grade.txt
Utils.java
PrintingJob.java
LockExample.java
Student.java
Project
Assignment4 ~/Desktop/Fall2020/Multithreading/Assignment
idea
out
src
com.multithreding
Question4
GradeDriver
GraderThread
Student
Utils
Question7
LockExample
PrintingJob.java
Main
Assignment4.iml
Grade.txt
Homework4.docx
~$mework4.docx
External Libraries
Scratches and Consoles
public static void main(String[] args)
{
PrinterQueue printerQueue = new PrinterQueue();
Thread thread[] = new Thread[10];
for (int i = 0; i < 10; i++)
{
thread[i] = new Thread(new PrintingJob(printerQueue), name: "Thread " + i);
}
for (int i = 0; i < 10; i++)
{
thread[i].start();
}
}
}
Run: LockExample
/Users/mrcricket/Library/Java/JavaVirtualMachines/openjdk-14.0.2/Contents/Home/bin/java -javaagent:/Applications/IntelliJ IDEA.app/Contents/lib/idea_rt.jar=51522:/Applications/IntelliJ IDEA.app/Contents/bin -Df
Thread 0: Going to print a document
Thread 9: Going to print a document
Thread 8: Going to print a document
Thread 7: Going to print a document
Thread 6: Going to print a document
Thread 5: Going to print a document
Thread 1: Going to print a document
Thread 4: Going to print a document
Thread 2: Going to print a document
Thread 3: Going to print a document
Thread 0: PrintQueue: Printing a Job during 7 seconds :: Time - Sun Oct 11 16:36:57 EDT 2020
Thread 0: The document has been printed
Thread 9: PrintQueue: Printing a Job during 6 seconds :: Time - Sun Oct 11 16:37:05 EDT 2020
Thread 9: The document has been printed
Thread 8: PrintQueue: Printing a Job during 9 seconds :: Time - Sun Oct 11 16:37:11 EDT 2020

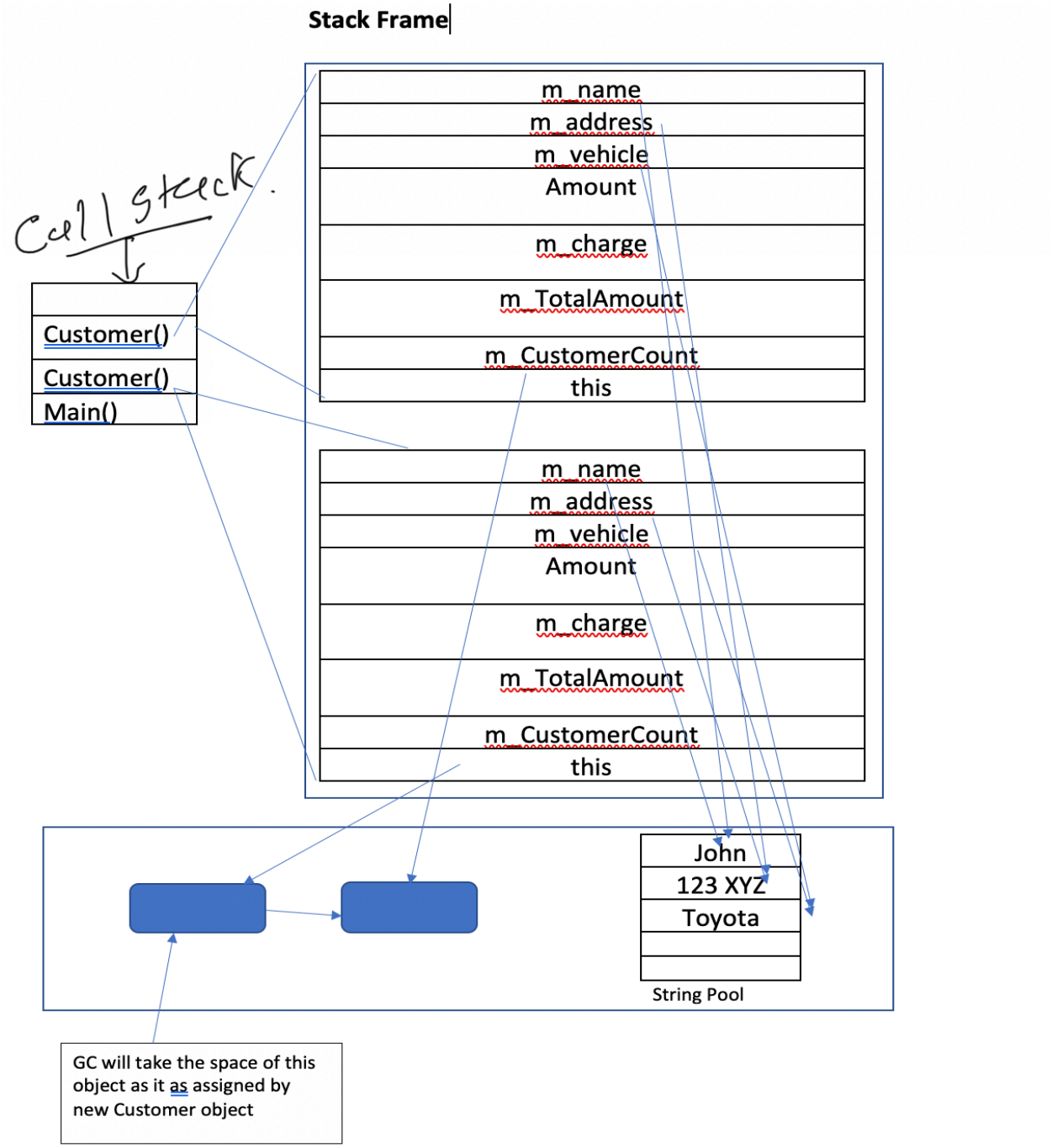
8. Consider the following class Customer class, Build a diagram to show Call Stack, Memory Stack, and Heap.

```
public class Customer{
private String name;
private String address;
private String vehicle;
private double amount;
private double charge;
private double TotalAmount;
private int CustomerCount;

public Customer() {
}

public Customer(name, address, vehicle, amount,
                        charge, TotalAmount, CustomerCount) {
        this.name=name;
        this.address=address;
        this.vehicle=vehicle;
        this.amount=amount;
        this.charge=charge;
        this.TotalAmount=TotalAmount;
        this.CustomerCount=CustomerCount;
}

public String getName()
```

```
{
return name;
}
public String getAddress()
{
return address;
}

public String getVehicle()
{
return vehicle;
}

public double getAmount()
{
return amount;
}

public double getCharge()
{
return charge;
}

public double getTotalAmount()
{
return TotalAmount;
}

public int getCustomerCount()
```

```
{
return CustomerCount;
}
}

public class TestDriver {
   public static void main(String[] args) {
            private String m_name=“John”;
       private String m_address=“123 XYZ”;
       private String m_vehicle=“Toyota”;
       private double m_amount=20;
       private double m_charge=25;
       private double m_TotalAmount=140;
       private int m_CustomerCount=10;
        Customer c = new Customer(m_name, m_address, m_;
              c = new Customer(m_name, m_address, m_vehicle,
                   m_amount, m_charge, m_TotalAmount,
m_Customer_Count);
   }
}
```

Call Stack, Stack Frame and heap space interaction



Explanation:

In Given Program call stack would me in this sequence

1) Main()
2) Customer()
3) Customer()

Call stack execute in LIFO manner means last in First out

➔ In Stack frame each frame contains the value of all the local variable such as m_name, m_address etc. Here string values are assign using literals hence all th local variable string value will point to **string pool**While we have double variable as well which will hold 2 consecutive entries in local variable arrays.

➔ Initially Customer object c assigned by new Customer object then same c object assigned by another new customer constructor hence the **previous assigned object space in heap will be reclaim by the garbage collector**

Assignment 5:

1. Explain:

What are possible ways Java objects become subject to garbage collection? give example code for each case.

JVM Garbage Collector will is process all unused/unreferenced memory automatically at runtime .

There are multiple ways to unreferenced the java object

1) nulling the reference
   example
   test e=new test();
   e=null;
2) assigning a reference to another
   Test e1=new Test ();
   Test e2=new Test ();
   e1=e2;//now the first object referred by e1 is available for garbage collection
3) anonymous object
   new test();

What is Starvation, what is the remedy for starvation?

- Starvation describes a situation where a thread is **unable to gain regular access to shared resources and is unable to make progress.** This happens when shared resources are made **unavailable for long periods by "greedy" threads.**
- For example, suppose an object provides a synchronized method that often takes a long time to return. If one

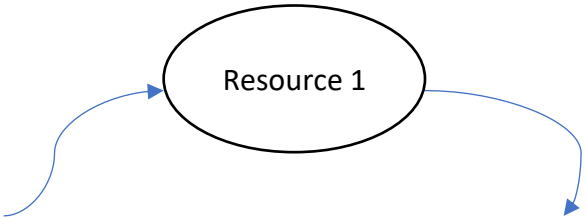
thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked.

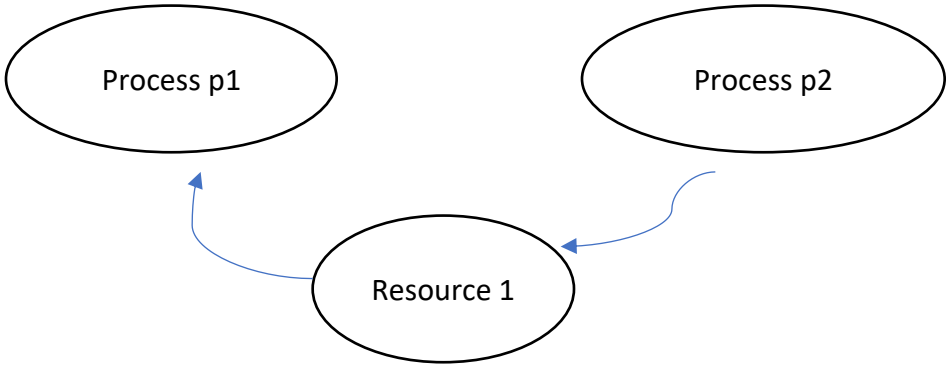
- While it is not possible to implement 100% fairness in Java we can still implement our synchronization constructs to increase fairness between threads.
- If we are using synchronized blocked make sure we use wait method for if the thread is running for long time. wait method will release the lock and allow other thread to work on critical section.
- use explicit lock and guard the critical section only

What is Deadlock and Race Condition?

Deadlock - Deadlock describes a situation where two or more
threads are blocked forever, waiting for each other

Example





- In Above example process p1 try to acquire lock on resource 1 but resource 1 is already locked by process p2 and process p2 try to acquire lock on resource r2 but it already locked by process p1. Now both the processes p1 and p2 are interdependent. This kind of situation known as Deadlock.

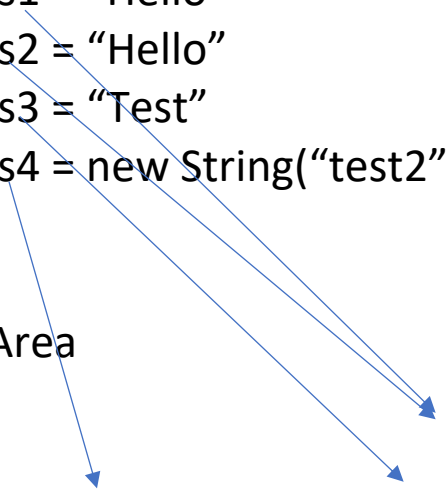
- Race Condition: A **race condition** occurs when two or more threads can access shared data and they try to change it at the same time. Execution order of threads are dependent on Thread scheduling algorithm, you don't know the order in which the threads will attempt to access the shared data. Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e. both threads are "racing" to access/change the data

What is String Pool?

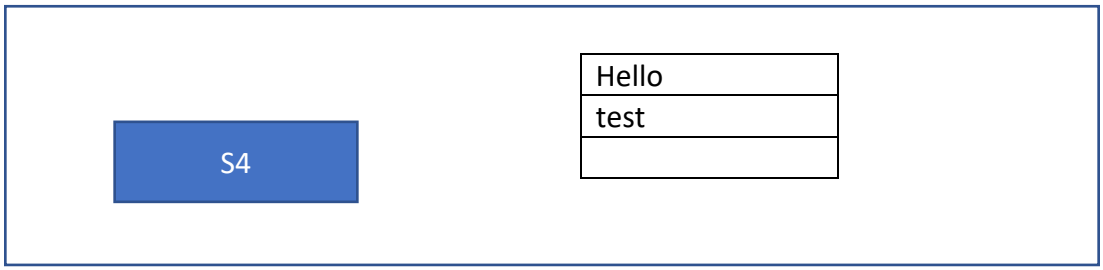
- String Pool is a storage area in Java heap. The JVM performs some steps while initializing string literals to increase performance and decrease memory overhead. To decrease the number of String objects created in the JVM, the String class keeps a pool of strings.
- Each time a string literal is created, the JVM checks the string literal pool first. If the string already exists in the string pool, a reference to the pooled instance returns. If the string does not exist in the pool, a new String object initializes and is placed in the pool

Example

String s1 = “Hello”
String s2 = “Hello”
String s3 = “Test”
String s4 = new String(“test2”)



Heap Area



What is Constant Pool?

- Constant pool is a part of .class file (and its in-memory representation) that contains constants needed to run the code of that class.
- The constant pool is organized as an array of variable-length elements
- Each constant occupies one element in the array.
- Throughout the class file, constants are referred to by the integer index that indicates their position in the array.
- The initial constant has an index of one, the second constant has an index of two, etc.
- These constants include literals specified by the programmer and symbolic references generated by compiler.

2. Provide four numbers (byte, short, int, long) examples. Show the results for Signed and Un-signed arithmetic. In each case, in binary show the base, position, and value in that position.

- Signed variables, such as signed integers will allow you to represent numbers both in the positive and negative ranges.
- if the 1st bit is 1 that means number is negative else positive
- Unsigned variables, such as unsigned integers, will only allow you to represent numbers in the positive.

Example of byte Unsigned & signed

10000001 = 1*2^7+0*2^6+……..+1*2^0 =129

Example of byte signed

First digit from left indicates the value is positive or negative

10000001 = 0*2^6+0*2^5+……+1*2^0 =1 * -1 = -1

for short unsigned & signed

1000000000000001 =
(1 × 2^15 + 0 × 2^14 + ….+ 0 × 2^1 + 1 × 2^0)(10) = 32 769(10)

for short signed
1000000000000001 =
(0 × 2^14 + … + 1 × 2^0)(10) = -1

for int unsigned & signed
10000000000000000000000000000001 =
(1× 2^31 + … + 1 × 2^0)(10) = 2147483649(10)

for int signed
10000000000000000000000000000001 =
(0 × 2^30 + … + 1 × 2^0)(10) = -1

for long unsigned & signed
1000000000000000000000000000000000000000000000000000000000000001 =
(1× 2^63 + … + 1 × 2^0)(10) = 92233720368547758 09(10)

for int signed
1000000000000000000000000000000000000000000000000000000000000001 =
(0 × 2^62 + … + 1 × 2^0)(10) = -1

*** (10) means 10 base.*

3. Consider Class File data structure, Explain each case with an Example:

a) magic, b) constant_pool, c) super_class, d) interfaces, e) fields, f) methods, g) attributes

```
ClassFile {
   u4          magic;
   u2          minor_version;
   u2          major_version;
   u2          constant_pool_count;
   cp_info     constant_pool[constant_pool_count-1];
   u2          access_flags;
   u2          this_class;
   u2          super_class;
```

```
    u2          interfaces_count;
    u2          interfaces[interfaces_count];
    u2          fields_count;
    field_info     fields[fields_count];
    u2          methods_count;
    method_info    methods[methods_count];
    u2          attributes_count;
    attribute_info attributes[attributes_count];
}
```

- The Java class file contains everything a JVM needs to know about one Java class or interface.
- In their order of appearance in the class file, the major - components are:

- magic
- version
- constant pool
- access flags
- this class

- super class
- interfaces
- fields
- methods
- attributes.

u1 u2 u4 --> unsigned 1,2,4 bytes quantity

Magic

- so first four byte in every class is always OXCAFEBABE
- This magic number makes Java class files easier to identify, because the odds are slim (means very less chance) that non-class files would start with the same initial four bytes
- second 4 bytes contains the major and minor version of class file
- JVM has maximum number it can load it number is higher than that then JVM will reject that class file.

Constant pool

- Class file stores the constants associated with class or interfaces.
- The constant pool is organized as an array of variable-length elements. Each constant occupies one element in the array.
- Throughout the class file, constants are referred to by the integer index that indicates their position in the array.

- The initial constant has an index of one, the second constant has an index of two, etc.
- Each element of the constant pool starts with a one-byte tag specifying the type of constant at that position in the array.

Super class

- The super class component, another two-byte index into the constant pool.
- Constant_pool[super_class] is a CONSTANT_Class element that points to the name of the super class from which this class descends.

Interfaces

- The interfaces component starts with a two-byte count of the **number of interfaces implemented by the class (or interface) defined in the file.**
- Immediately following is an array that contains one index into the constant pool for each interface implemented by the class. Each interface is represented by a CONSTANT_Class element in the constant pool that points to the name of the interface.

Fields

- The fields component starts with a two-byte count of the number of fields in this class or interface
- A field is an instance or class variable of the class or interface.

- Following the count is an array of variable-length structures, one for each field. Each structure reveals information about one field such as the field's name, type, and, if it is a final variable, its constant value. Some information is contained in the structure itself, and some is contained in constant pool locations pointed to by the structure
- The only fields that appear in the list are those that were declared by the class or interface defined in the file; no fields inherited from super classes or superinterfaces appear in the list

Methods

- The methods component starts with a two-byte count of the number of methods in the class or interface.
- This count includes only those methods that are explicitly defined by this class, not any methods that may be inherited from superclasses, Following the method count are the methods themselves.

Attributes

- It returns the number of attributes (instance variables) present in current class file.
- For example, one attribute is the source code attribute; it reveals the **name of the source file from which this class file was compiled**.

4. Suppose you have a two dimensional array input data;

```
int[][] arr = { { 9, 12, 6, 14, 10, 21, 13}, { 3, 5, 41, 16, 14,
10, 21},
                { 3, 15, 41, 17, 11, 10, 51}, { 3, 15, 41, 17, 11,
10, 51}
                { 4, 15, 35, 17, 11, 12, 55}, { 2, 16, 31, 18, 12,
11, 42} };
```

Write Java program:

a) Create six Threads where each thread-id corresponds to a row in array input data, for example, (tid1, row1), (tid2, row2), (tid3, row3), (tid4, row4), (tid5, row5), (tid6, row6)

b) Write code for each thread to sort its row of array data us-ing sort method in Java Collections library,

c) Write code to update array data with sorted values,

d) Sort all rows in array data using HeapSort,

e) Write code for each thread to print the sorted data.

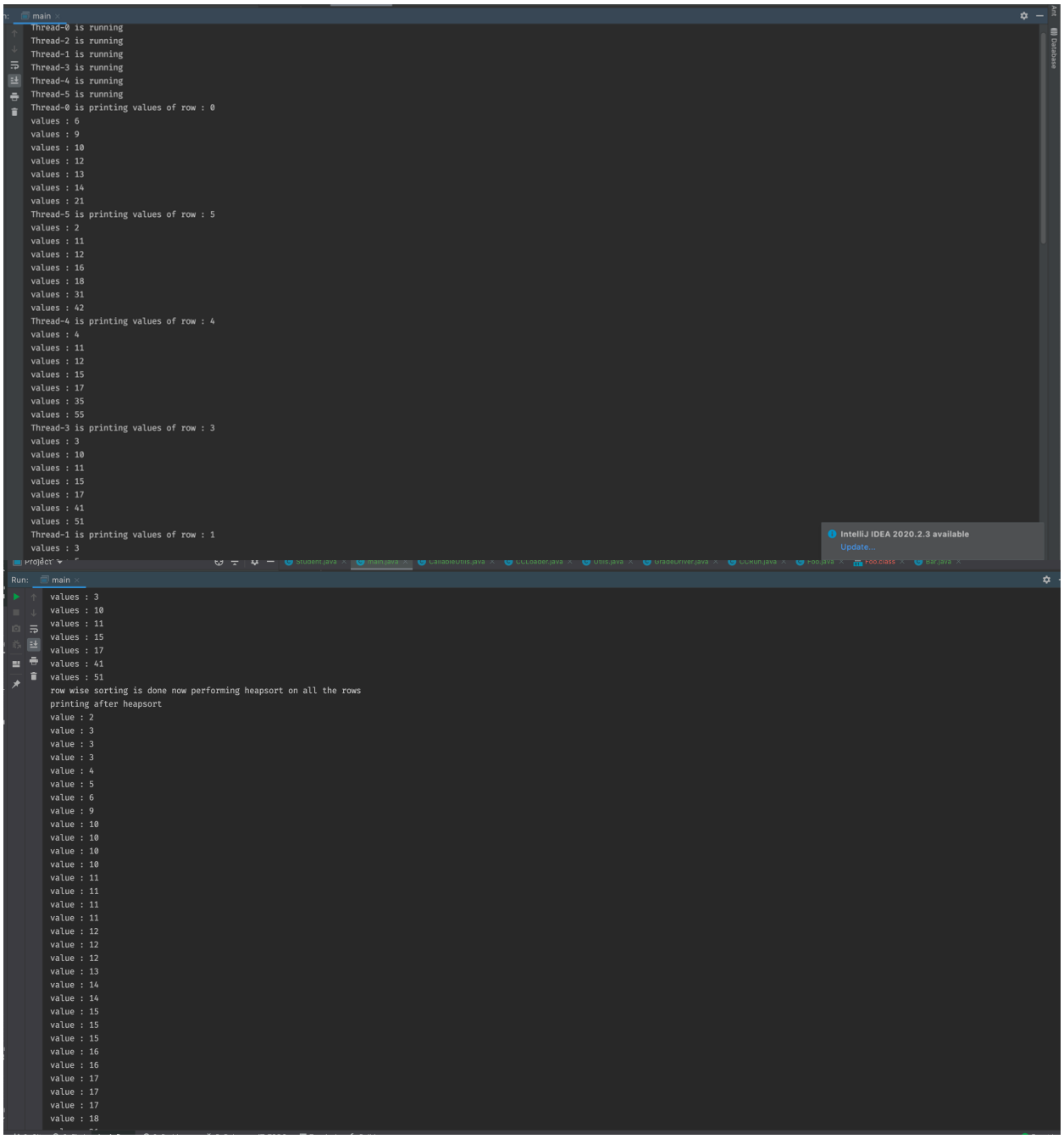
Notes: You need to protect Array data. How do you protect array data? First you sort using Collections class, and then for Heapsort. How lock mechanism works in (b) and (d)?

Approach :

- Created separate runnable class ArrayData to perform all type of sorting.

- Created six threads for each rows and perform sorting at row level
  Inside the synchronized block to protect the array
- We can create all threads using **loop** or **ThreadPoolExecutor**
- once the row level sorting gets completed perform the heapsort for entire 2d array using PriorityQueue
- To check code implementation please check source code.

Output :


main
Thread-0 is running
Thread-2 is running
Thread-1 is running
Thread-3 is running
Thread-4 is running
Thread-5 is running
Thread-0 is printing values of row : 0
values : 6
values : 9
values : 10
values : 12
values : 13
values : 14
values : 21
Thread-5 is printing values of row : 5
values : 2
values : 11
values : 12
values : 16
values : 18
values : 31
values : 42
Thread-4 is printing values of row : 4
values : 4
values : 11
values : 12
values : 15
values : 17
values : 35
values : 55
Thread-3 is printing values of row : 3
values : 3
values : 10
values : 11
values : 15
values : 17
values : 41
values : 51
Thread-1 is printing values of row : 1
values : 3
IntelliJ IDEA 2020.2.3 available
Update...
Run:
main
values : 3
values : 10
values : 11
values : 15
values : 17
values : 41
values : 51
row wise sorting is done now performing heapsort on all the rows
printing after heapsort
value : 2
value : 3
value : 3
value : 3
value : 4
value : 5
value : 6
value : 9
value : 10
value : 10
value : 10
value : 10
value : 11
value : 11
value : 11
value : 11
value : 12
value : 12
value : 12
value : 13
value : 14
value : 14
value : 15
value : 15
value : 15
value : 16
value : 16
value : 17
value : 17
value : 17
value : 18

5. The following link provides an example of user defined class loader called CCLoad-er:
https://www.journaldev.com/349/java-classloader

a) Analyze the code and Explain as how it works.
b) Compile and run CCLoader. What are the outputs?
c) Add Student class defined in Homework2. Build CCLoader, Compile and Run, Explain outputs.

a) Analyze the code and Explain as how it works.

CCLoader Class

- In Given example We are creating a custom class loader by extending ClassLoader class and override the loadClass() method and check the class name start with our local class file then load that class using custom class loader.

- We have also created the getClass() method which will determined the full class name and then call the loadClassFileData() data method which will return the bytcode from the class file.

- After that we are calling defineClass() method of classloader class and reolve the class references using resolveClass() method.

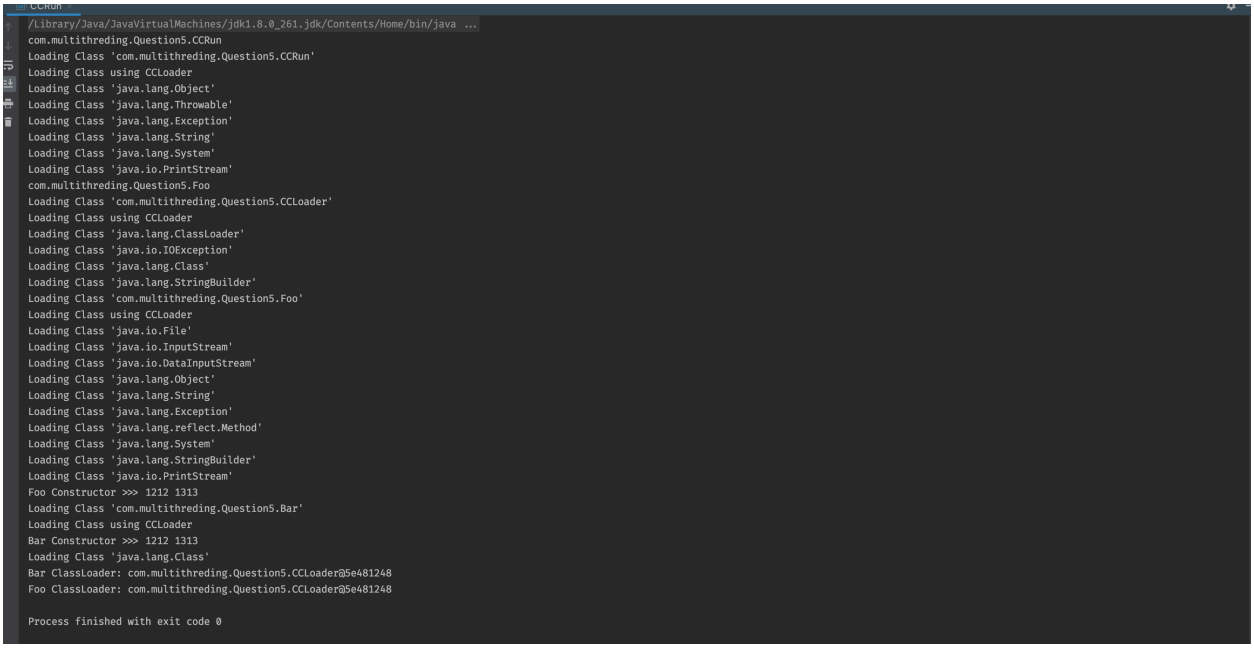
CCRun Class

- In CCRun Class we are accepting the all the command line argument and processing each class with custom class Loader
- Initailly we are creating the object of custom class loader

// This will call the constructor of super class(ClassLoader)
CCLoader ccl = new CCLoader(CCRun.class.getClassLoader());

- Then we are calling the load class method which will do following things
- Check if the class name of the file if the class is local class then use custom class loader and call the getClass method.
- getclass() method will get the binary class file then call defineClass() method which is inherited from the ClassLoader class and at last resolve the class reference.

b) Compile and run CCLoader. What are the outputs?

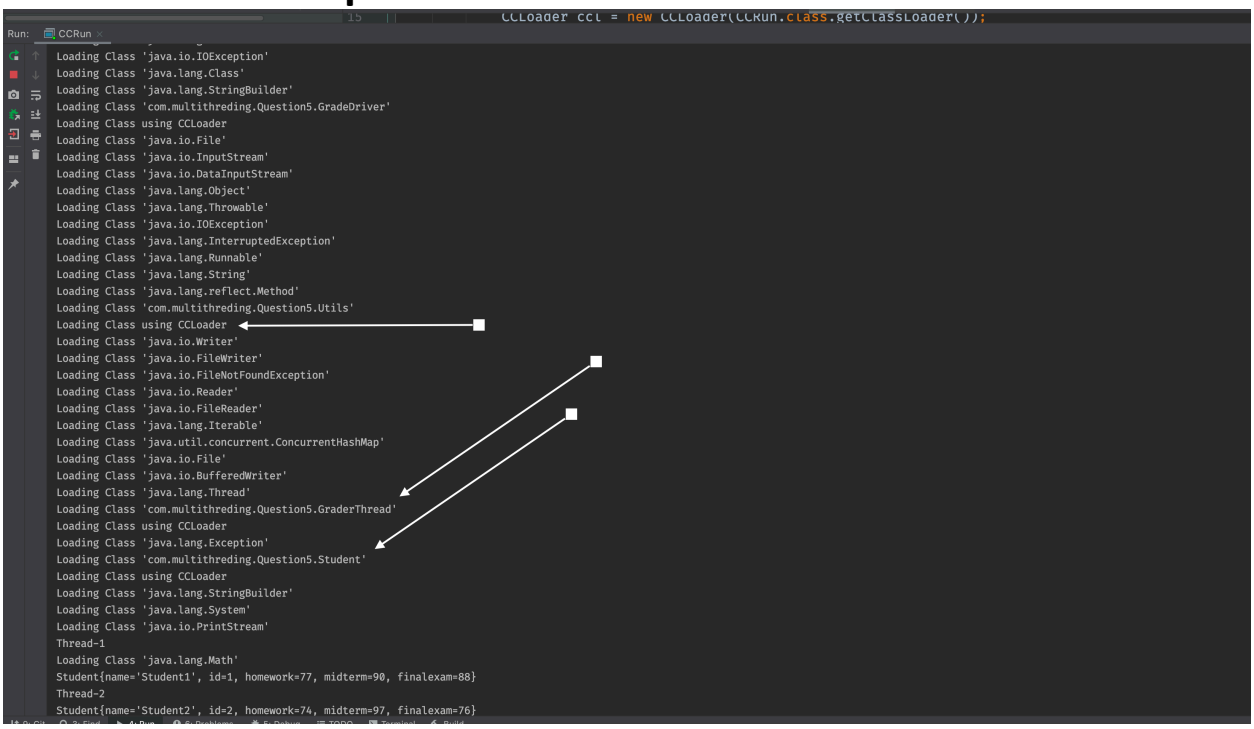


output shows that class name with their class Loader
for Example
Bar ClassLoader: CCLoader@71f6f0bf
Foo ClassLoader: CCLoader@71f6f0bf

Foo and Bar class load by our custom class loader.

c) Add Student class defined in Homework2. Build CCLoader, Compile

Student class output:



Explanation:

- Using custom class loader I have load the Grade Driver class of assignment 2 which h includes the student , util and grader class.
- In output screen we can see that custom class loader is loading all the depended classes.
- At the end of the code to verify the class is load successfully or not we have called the main method and the output shows that it is working fine.

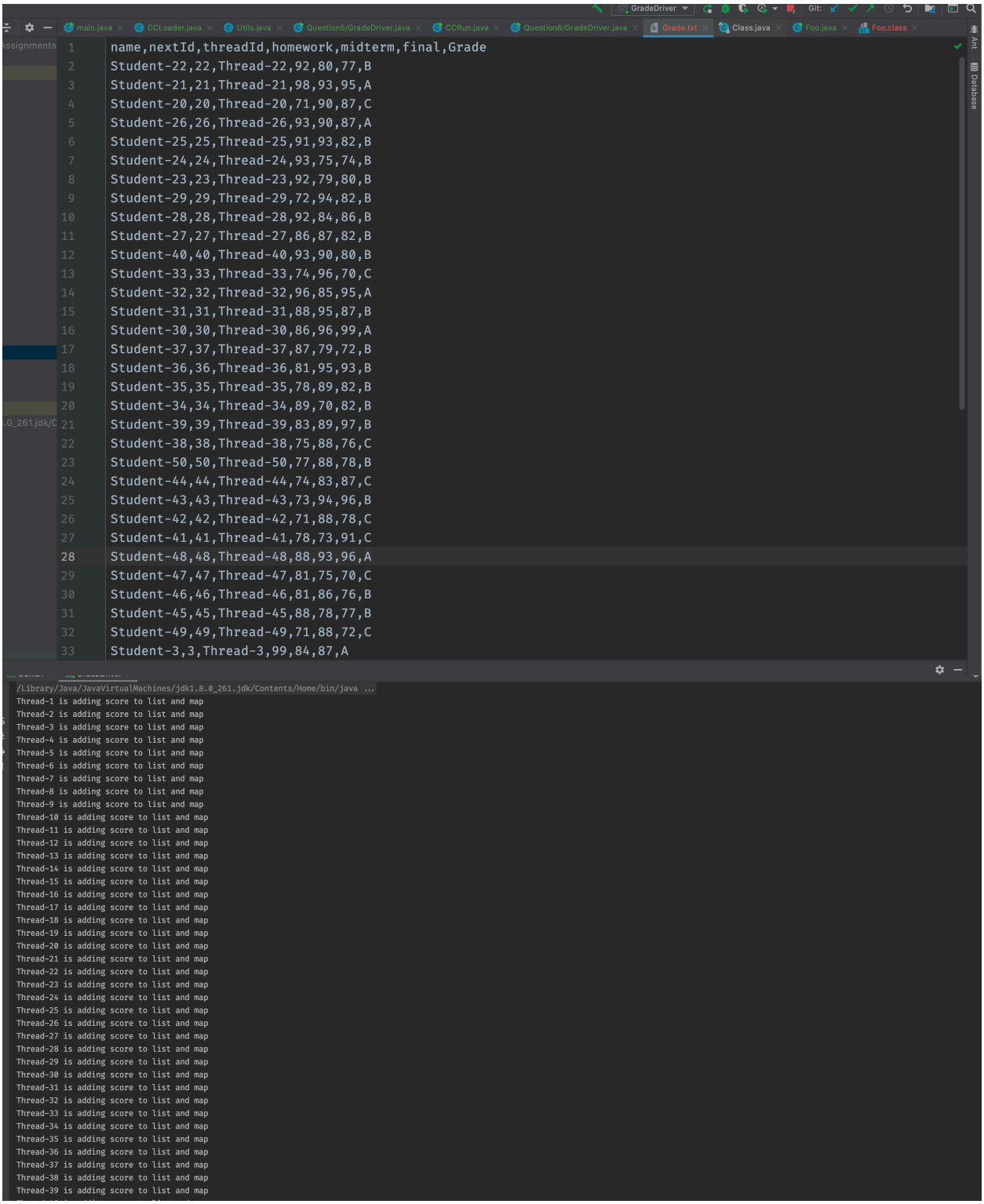
Q:6 Student program with thread executor

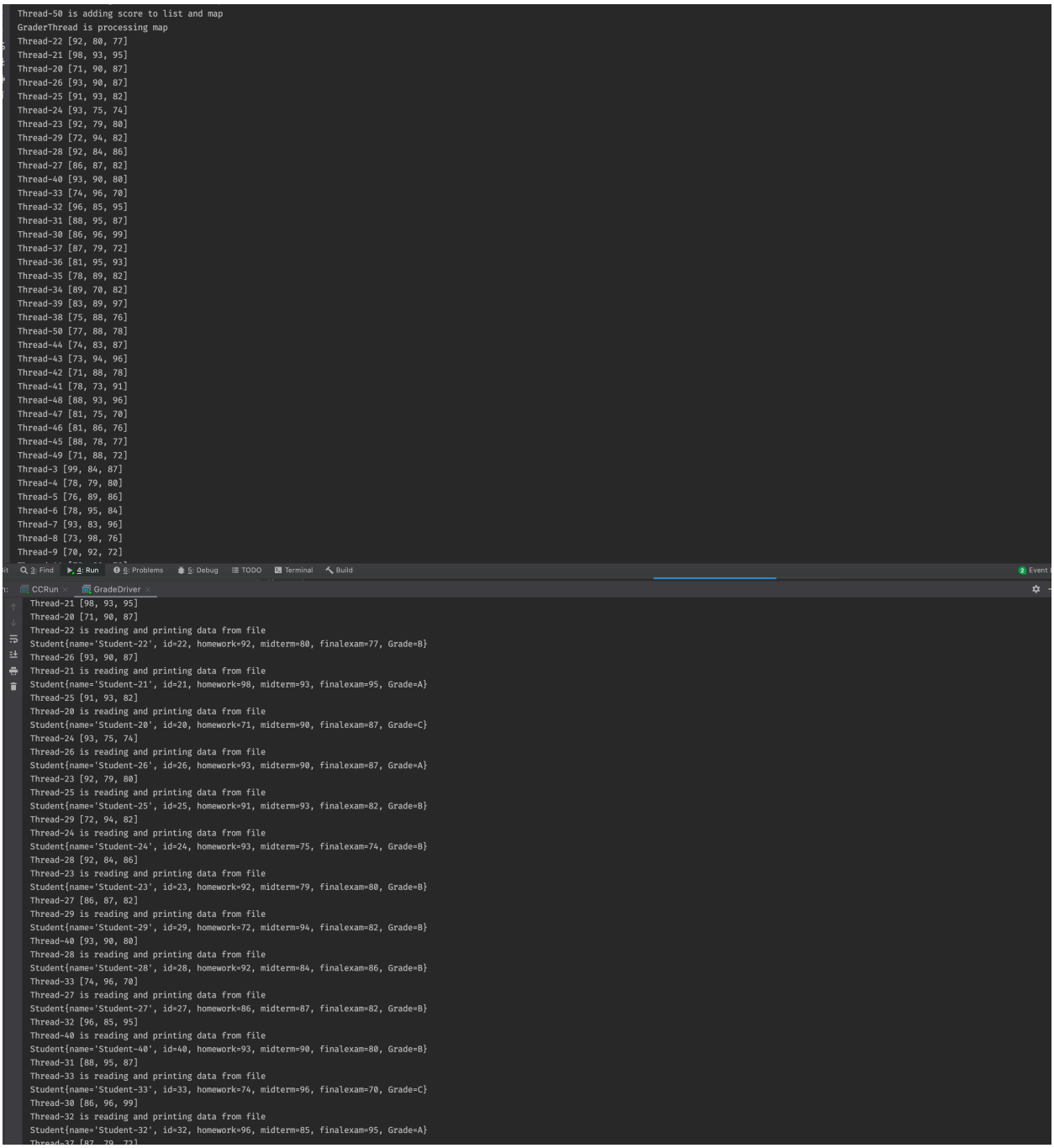
6. In Homework4, you created 50 Student threads and one Grader thread and man-aged the concurrency using Explicit Locks. In this problem, consider creating threads using ThreadPoolExecutor. How would you design and solve (problem-4 in home-work4) using ThreadPoolExecutor? a) Show your design, b)Write code, compile and run.
Note: https://howtodoinjava.com/java/multi-threading/java-thread-pool-executor-example/

Approach :

- I have used the ThreadPoolExecutor class to create the 50 Student thread
- Assigned Student object to each Thread of ThreadPool executer
  And execute the threadpool executor.
- I have implemented explicit locking to protect the file and hashmap.

Output :


GradeDriver
Git:
main.java
CCLoader.java
Utils.java
Question5/GradeDriver.java
CCRun.java
Question6/GradeDriver.java
Grade.txt
Class.java
Foo.java
Foo.class
Ant
Database
name,nextId,threadId,homework,midterm,final,Grade
Student-22,22,Thread-22,92,80,77,B
Student-21,21,Thread-21,98,93,95,A
Student-20,20,Thread-20,71,90,87,C
Student-26,26,Thread-26,93,90,87,A
Student-25,25,Thread-25,91,93,82,B
Student-24,24,Thread-24,93,75,74,B
Student-23,23,Thread-23,92,79,80,B
Student-29,29,Thread-29,72,94,82,B
Student-28,28,Thread-28,92,84,86,B
Student-27,27,Thread-27,86,87,82,B
Student-40,40,Thread-40,93,90,80,B
Student-33,33,Thread-33,74,96,70,C
Student-32,32,Thread-32,96,85,95,A
Student-31,31,Thread-31,88,95,87,B
Student-30,30,Thread-30,86,96,99,A
Student-37,37,Thread-37,87,79,72,B
Student-36,36,Thread-36,81,95,93,B
Student-35,35,Thread-35,78,89,82,B
Student-34,34,Thread-34,89,70,82,B
Student-39,39,Thread-39,83,89,97,B
Student-38,38,Thread-38,75,88,76,C
Student-50,50,Thread-50,77,88,78,B
Student-44,44,Thread-44,74,83,87,C
Student-43,43,Thread-43,73,94,96,B
Student-42,42,Thread-42,71,88,78,C
Student-41,41,Thread-41,78,73,91,C
Student-48,48,Thread-48,88,93,96,A
Student-47,47,Thread-47,81,75,70,C
Student-46,46,Thread-46,81,86,76,B
Student-45,45,Thread-45,88,78,77,B
Student-49,49,Thread-49,71,88,72,C
Student-3,3,Thread-3,99,84,87,A
/Library/Java/JavaVirtualMachines/jdk1.8.0_261.jdk/Contents/Home/bin/java ...
Thread-1 is adding score to list and map
Thread-2 is adding score to list and map
Thread-3 is adding score to list and map
Thread-4 is adding score to list and map
Thread-5 is adding score to list and map
Thread-6 is adding score to list and map
Thread-7 is adding score to list and map
Thread-8 is adding score to list and map
Thread-9 is adding score to list and map
Thread-10 is adding score to list and map
Thread-11 is adding score to list and map
Thread-12 is adding score to list and map
Thread-13 is adding score to list and map
Thread-14 is adding score to list and map
Thread-15 is adding score to list and map
Thread-16 is adding score to list and map
Thread-17 is adding score to list and map
Thread-18 is adding score to list and map
Thread-19 is adding score to list and map
Thread-20 is adding score to list and map
Thread-21 is adding score to list and map
Thread-22 is adding score to list and map
Thread-23 is adding score to list and map
Thread-24 is adding score to list and map
Thread-25 is adding score to list and map
Thread-26 is adding score to list and map
Thread-27 is adding score to list and map
Thread-28 is adding score to list and map
Thread-29 is adding score to list and map
Thread-30 is adding score to list and map
Thread-31 is adding score to list and map
Thread-32 is adding score to list and map
Thread-33 is adding score to list and map
Thread-34 is adding score to list and map
Thread-35 is adding score to list and map
Thread-36 is adding score to list and map
Thread-37 is adding score to list and map
Thread-38 is adding score to list and map
Thread-39 is adding score to list and map


Thread-50 is adding score to list and map
GraderThread is processing map
Thread-22 [92, 80, 77]
Thread-21 [98, 93, 95]
Thread-20 [71, 90, 87]
Thread-26 [93, 90, 87]
Thread-25 [91, 93, 82]
Thread-24 [93, 75, 74]
Thread-23 [92, 79, 80]
Thread-29 [72, 94, 82]
Thread-28 [92, 84, 86]
Thread-27 [86, 87, 82]
Thread-40 [93, 90, 80]
Thread-33 [74, 96, 70]
Thread-32 [96, 85, 95]
Thread-31 [88, 95, 87]
Thread-30 [86, 96, 99]
Thread-37 [87, 79, 72]
Thread-36 [81, 95, 93]
Thread-35 [78, 89, 82]
Thread-34 [89, 70, 82]
Thread-39 [83, 89, 97]
Thread-38 [75, 88, 76]
Thread-50 [77, 88, 78]
Thread-44 [74, 83, 87]
Thread-43 [73, 94, 96]
Thread-42 [71, 88, 78]
Thread-41 [78, 73, 91]
Thread-48 [88, 93, 96]
Thread-47 [81, 75, 70]
Thread-46 [81, 86, 76]
Thread-45 [88, 78, 77]
Thread-49 [71, 88, 72]
Thread-3 [99, 84, 87]
Thread-4 [78, 79, 80]
Thread-5 [76, 89, 86]
Thread-6 [78, 95, 84]
Thread-7 [93, 83, 96]
Thread-8 [73, 98, 76]
Thread-9 [70, 92, 72]
3: Find
4: Run
6: Problems
5: Debug
TODO
Terminal
Build
CCRun
GradeDriver
Thread-21 [98, 93, 95]
Thread-20 [71, 90, 87]
Thread-22 is reading and printing data from file
Student{name='Student-22', id=22, homework=92, midterm=80, finalexam=77, Grade=B}
Thread-26 [93, 90, 87]
Thread-21 is reading and printing data from file
Student{name='Student-21', id=21, homework=98, midterm=93, finalexam=95, Grade=A}
Thread-25 [91, 93, 82]
Thread-20 is reading and printing data from file
Student{name='Student-20', id=20, homework=71, midterm=90, finalexam=87, Grade=C}
Thread-24 [93, 75, 74]
Thread-26 is reading and printing data from file
Student{name='Student-26', id=26, homework=93, midterm=90, finalexam=87, Grade=A}
Thread-23 [92, 79, 80]
Thread-25 is reading and printing data from file
Student{name='Student-25', id=25, homework=91, midterm=93, finalexam=82, Grade=B}
Thread-29 [72, 94, 82]
Thread-24 is reading and printing data from file
Student{name='Student-24', id=24, homework=93, midterm=75, finalexam=74, Grade=B}
Thread-28 [92, 84, 86]
Thread-23 is reading and printing data from file
Student{name='Student-23', id=23, homework=92, midterm=79, finalexam=80, Grade=B}
Thread-27 [86, 87, 82]
Thread-29 is reading and printing data from file
Student{name='Student-29', id=29, homework=72, midterm=94, finalexam=82, Grade=B}
Thread-40 [93, 90, 80]
Thread-28 is reading and printing data from file
Student{name='Student-28', id=28, homework=92, midterm=84, finalexam=86, Grade=B}
Thread-33 [74, 96, 70]
Thread-27 is reading and printing data from file
Student{name='Student-27', id=27, homework=86, midterm=87, finalexam=82, Grade=B}
Thread-32 [96, 85, 95]
Thread-40 is reading and printing data from file
Student{name='Student-40', id=40, homework=93, midterm=90, finalexam=80, Grade=B}
Thread-31 [88, 95, 87]
Thread-33 is reading and printing data from file
Student{name='Student-33', id=33, homework=74, midterm=96, finalexam=70, Grade=C}
Thread-30 [86, 96, 99]
Thread-32 is reading and printing data from file
Student{name='Student-32', id=32, homework=96, midterm=85, finalexam=95, Grade=A}

Q:7 Student program with callable and future task

7. Define Student class with instance variables name, id, homework, midterm, and fi-nal-exam. The name is a string whereas others are all integers. Add a static variable nextId which is integer and statically initialized to 1. In each of them, the id should be assigned to the next available id given by nextId. The default constructor should set the name of the student object to "StudentX" where X is the next id.

Create 30 Callable Student Threads, each to be identified with string name-Thread-nextId that stores their average grade scores into ScoresHashmap, and also returns the thread string name after 1 second. What should the map hold as key/value? How do you protect the map?

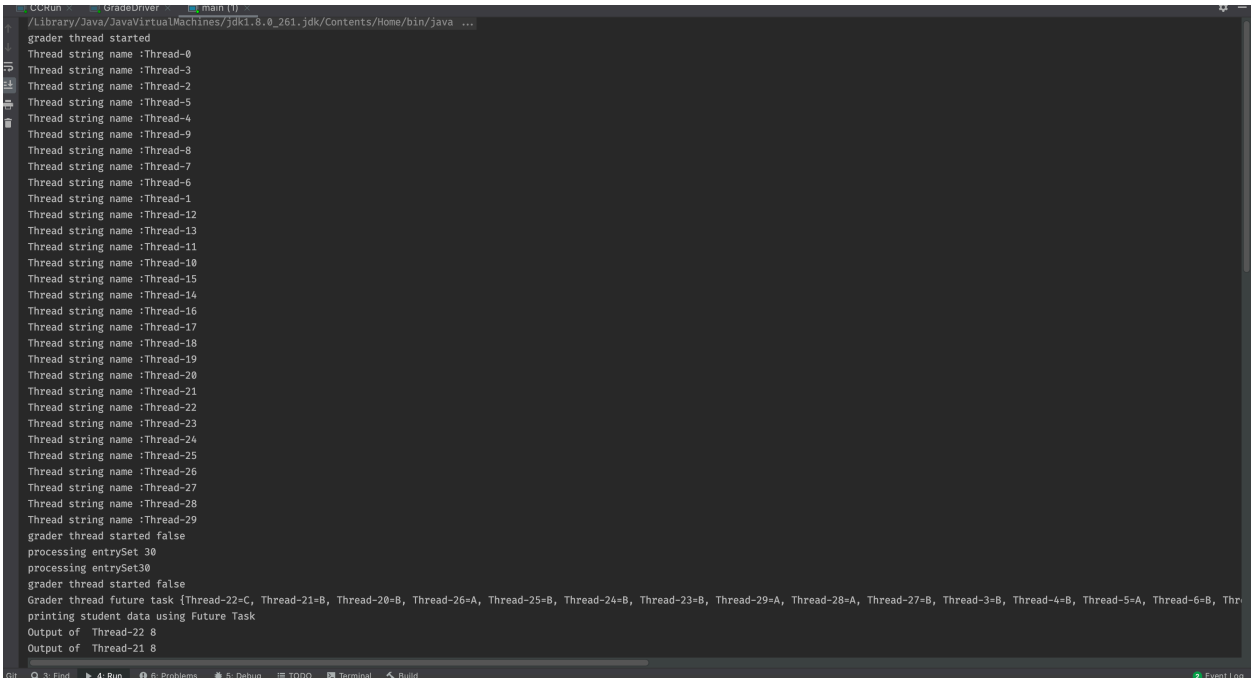
Create Callable GraderThread to read grade score from ScoresHashmap and calcu-late the Final grade using letter grades A,B,C,D,E,F from ScoresHashmap (you need to build score range for each letter grade) and then send the final grade to each student. And also stores the final grade for All students in FinalGrades file. Do you need to pro-tect the map? What method do you use to send final grades to students?

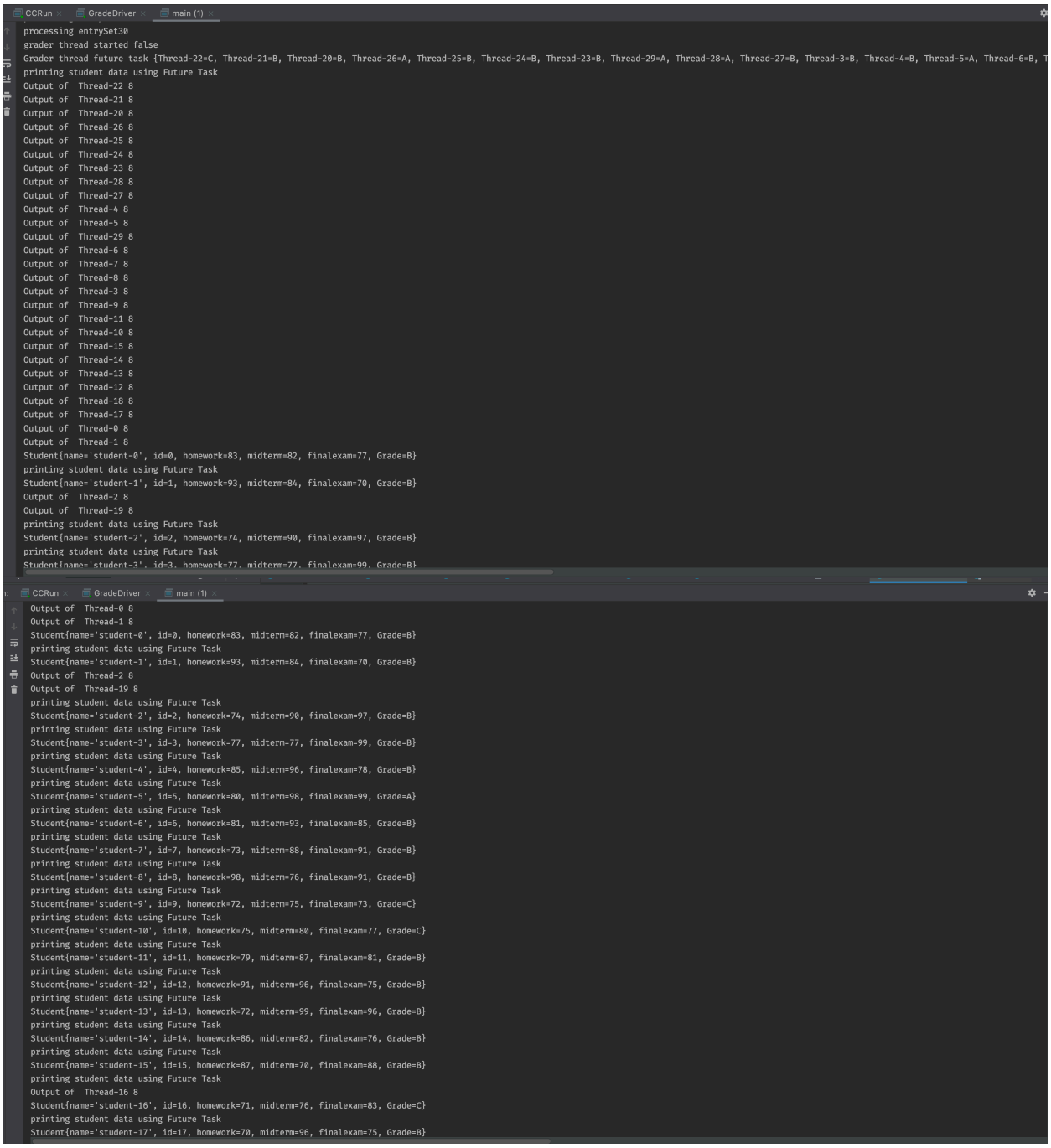
Create FutureTask thread to report the final grades for all students stored in Final-Grades file by GraderThread. Compile and Run all code.

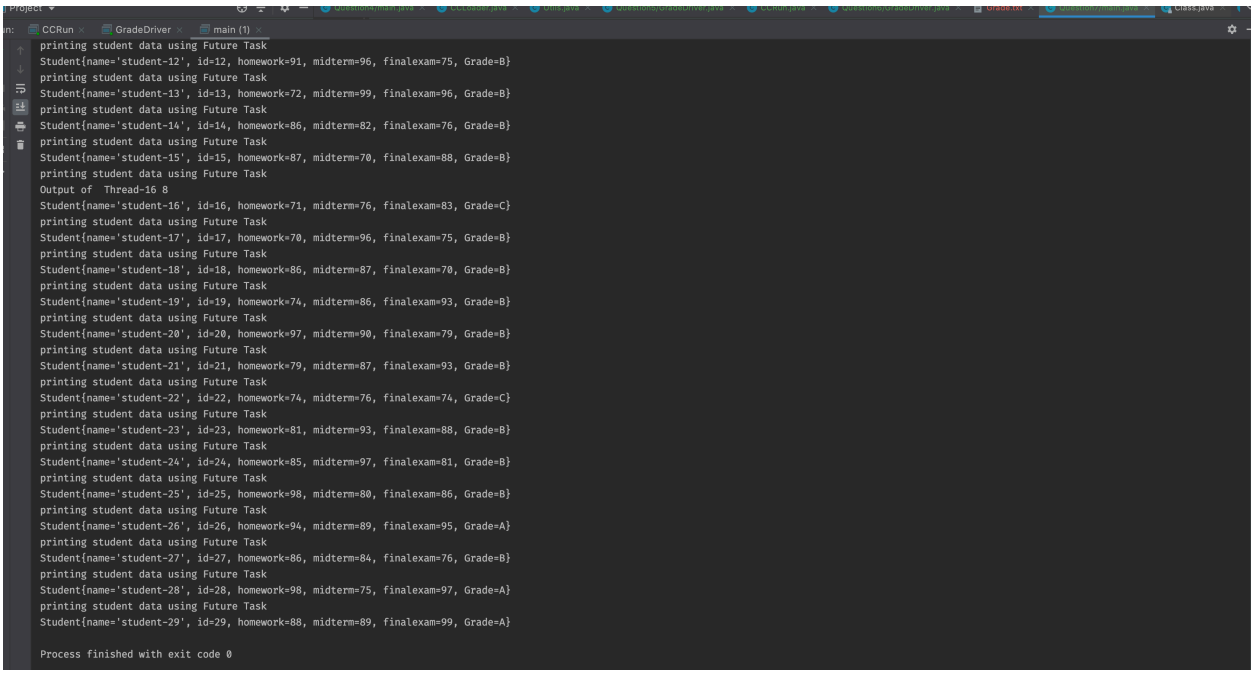
Approach:

- Created callable Student and grader class.
- Student class call method will return the student object without final grade.
- Grader class call method will return the hashmap which contains all the student thread and their final grade detail
- Created one future task to hold the output of Grader thread and later on pass that output value to student future task and set the Final grader of each student.
- For detail implementation please review code file

Output:




CCRun
GradeDriver
main (1)
processing entrySet30
grader thread started false
Grader thread future task {Thread-22=C, Thread-21=B, Thread-20=B, Thread-26=A, Thread-25=B, Thread-24=B, Thread-23=B, Thread-29=A, Thread-28=A, Thread-27=B, Thread-3=B, Thread-4=B, Thread-5=A, Thread-6=B, T
printing student data using Future Task
Output of Thread-22 8
Output of Thread-21 8
Output of Thread-20 8
Output of Thread-26 8
Output of Thread-25 8
Output of Thread-24 8
Output of Thread-23 8
Output of Thread-28 8
Output of Thread-27 8
Output of Thread-4 8
Output of Thread-5 8
Output of Thread-29 8
Output of Thread-6 8
Output of Thread-7 8
Output of Thread-8 8
Output of Thread-3 8
Output of Thread-9 8
Output of Thread-11 8
Output of Thread-10 8
Output of Thread-15 8
Output of Thread-14 8
Output of Thread-13 8
Output of Thread-12 8
Output of Thread-18 8
Output of Thread-17 8
Output of Thread-0 8
Output of Thread-1 8
Student{name='student-0', id=0, homework=83, midterm=82, finalexam=77, Grade=B}
printing student data using Future Task
Student{name='student-1', id=1, homework=93, midterm=84, finalexam=70, Grade=B}
Output of Thread-2 8
Output of Thread-19 8
printing student data using Future Task
Student{name='student-2', id=2, homework=74, midterm=90, finalexam=97, Grade=B}
printing student data using Future Task
Student{name='student-3', id=3, homework=77, midterm=77, finalexam=99, Grade=B}
CCRun
GradeDriver
main (1)
Output of Thread-0 8
Output of Thread-1 8
Student{name='student-0', id=0, homework=83, midterm=82, finalexam=77, Grade=B}
printing student data using Future Task
Student{name='student-1', id=1, homework=93, midterm=84, finalexam=70, Grade=B}
Output of Thread-2 8
Output of Thread-19 8
printing student data using Future Task
Student{name='student-2', id=2, homework=74, midterm=90, finalexam=97, Grade=B}
printing student data using Future Task
Student{name='student-3', id=3, homework=77, midterm=77, finalexam=99, Grade=B}
printing student data using Future Task
Student{name='student-4', id=4, homework=85, midterm=96, finalexam=78, Grade=B}
printing student data using Future Task
Student{name='student-5', id=5, homework=80, midterm=98, finalexam=99, Grade=A}
printing student data using Future Task
Student{name='student-6', id=6, homework=81, midterm=93, finalexam=85, Grade=B}
printing student data using Future Task
Student{name='student-7', id=7, homework=73, midterm=88, finalexam=91, Grade=B}
printing student data using Future Task
Student{name='student-8', id=8, homework=98, midterm=76, finalexam=91, Grade=B}
printing student data using Future Task
Student{name='student-9', id=9, homework=72, midterm=75, finalexam=73, Grade=C}
printing student data using Future Task
Student{name='student-10', id=10, homework=75, midterm=80, finalexam=77, Grade=C}
printing student data using Future Task
Student{name='student-11', id=11, homework=79, midterm=87, finalexam=81, Grade=B}
printing student data using Future Task
Student{name='student-12', id=12, homework=91, midterm=96, finalexam=75, Grade=B}
printing student data using Future Task
Student{name='student-13', id=13, homework=72, midterm=99, finalexam=96, Grade=B}
printing student data using Future Task
Student{name='student-14', id=14, homework=86, midterm=82, finalexam=76, Grade=B}
printing student data using Future Task
Student{name='student-15', id=15, homework=87, midterm=70, finalexam=88, Grade=B}
printing student data using Future Task
Output of Thread-16 8
Student{name='student-16', id=16, homework=71, midterm=76, finalexam=83, Grade=C}
printing student data using Future Task
Student{name='student-17', id=17, homework=70, midterm=96, finalexam=75, Grade=B}


CCRun
GradeDriver
main (1)
printing student data using Future Task
Student{name='student-12', id=12, homework=91, midterm=96, finalexam=75, Grade=B}
printing student data using Future Task
Student{name='student-13', id=13, homework=72, midterm=99, finalexam=96, Grade=B}
printing student data using Future Task
Student{name='student-14', id=14, homework=86, midterm=82, finalexam=76, Grade=B}
printing student data using Future Task
Student{name='student-15', id=15, homework=87, midterm=70, finalexam=88, Grade=B}
printing student data using Future Task
Output of Thread-16 8
Student{name='student-16', id=16, homework=71, midterm=76, finalexam=83, Grade=C}
printing student data using Future Task
Student{name='student-17', id=17, homework=70, midterm=96, finalexam=75, Grade=B}
printing student data using Future Task
Student{name='student-18', id=18, homework=86, midterm=87, finalexam=70, Grade=B}
printing student data using Future Task
Student{name='student-19', id=19, homework=74, midterm=86, finalexam=93, Grade=B}
printing student data using Future Task
Student{name='student-20', id=20, homework=97, midterm=90, finalexam=79, Grade=B}
printing student data using Future Task
Student{name='student-21', id=21, homework=79, midterm=87, finalexam=93, Grade=B}
printing student data using Future Task
Student{name='student-22', id=22, homework=74, midterm=76, finalexam=74, Grade=C}
printing student data using Future Task
Student{name='student-23', id=23, homework=81, midterm=93, finalexam=88, Grade=B}
printing student data using Future Task
Student{name='student-24', id=24, homework=85, midterm=97, finalexam=81, Grade=B}
printing student data using Future Task
Student{name='student-25', id=25, homework=98, midterm=80, finalexam=86, Grade=B}
printing student data using Future Task
Student{name='student-26', id=26, homework=94, midterm=89, finalexam=95, Grade=A}
printing student data using Future Task
Student{name='student-27', id=27, homework=86, midterm=84, finalexam=76, Grade=B}
printing student data using Future Task
Student{name='student-28', id=28, homework=98, midterm=75, finalexam=97, Grade=A}
printing student data using Future Task
Student{name='student-29', id=29, homework=88, midterm=89, finalexam=99, Grade=A}
Process finished with exit code 0

8. A deadlock is when two or more threads are blocked waiting to obtain locks that some of the other threads in the deadlock are holding. Deadlock can occur when mul-tiple threads need the same locks, at the same time, but obtain them in different order. For instance, if thread1 locks A, and tries to lock B, and thread2 has already locked B, and tries to lock A, a deadlock arises. Thread1 can never get B, and thread2 can never get A. In addition, neither of them will ever know. They will remain blocked on each of their object, A and B, forever. This situation is a deadlock.

Thread-1 locks A, waits for B
Thread-2 locks B, waits for A

http://tutorials.jenkov.com/java-concurrency/deadlock.html

A) Explain As why this code Deadlocks?
B) Compile and Run this code.
C) Is Race Condition possible in this code, Yes/No, Why?

```
public class TreeNode {
  TreeNode parent   = null;
  List  children = new ArrayList();
  public synchronized void addChild(TreeNode child){
    if(!this.children.contains(child)) {
      this.children.add(child);
```

```
    child.setParentOnly(this);
   }
  }
  public synchronized void addChildOnly(TreeNode child){
   if(!this.children.contains(child){
    this.children.add(child);
   }
  }
  public synchronized void setParent(TreeNode parent){
   this.parent = parent;
   parent.addChildOnly(this);
  }
  public synchronized void setParentOnly(TreeNode parent){
   this.parent = parent;
  }
}
```

Explain As why this code Deadlocks?

Possible Deadlock scenario :

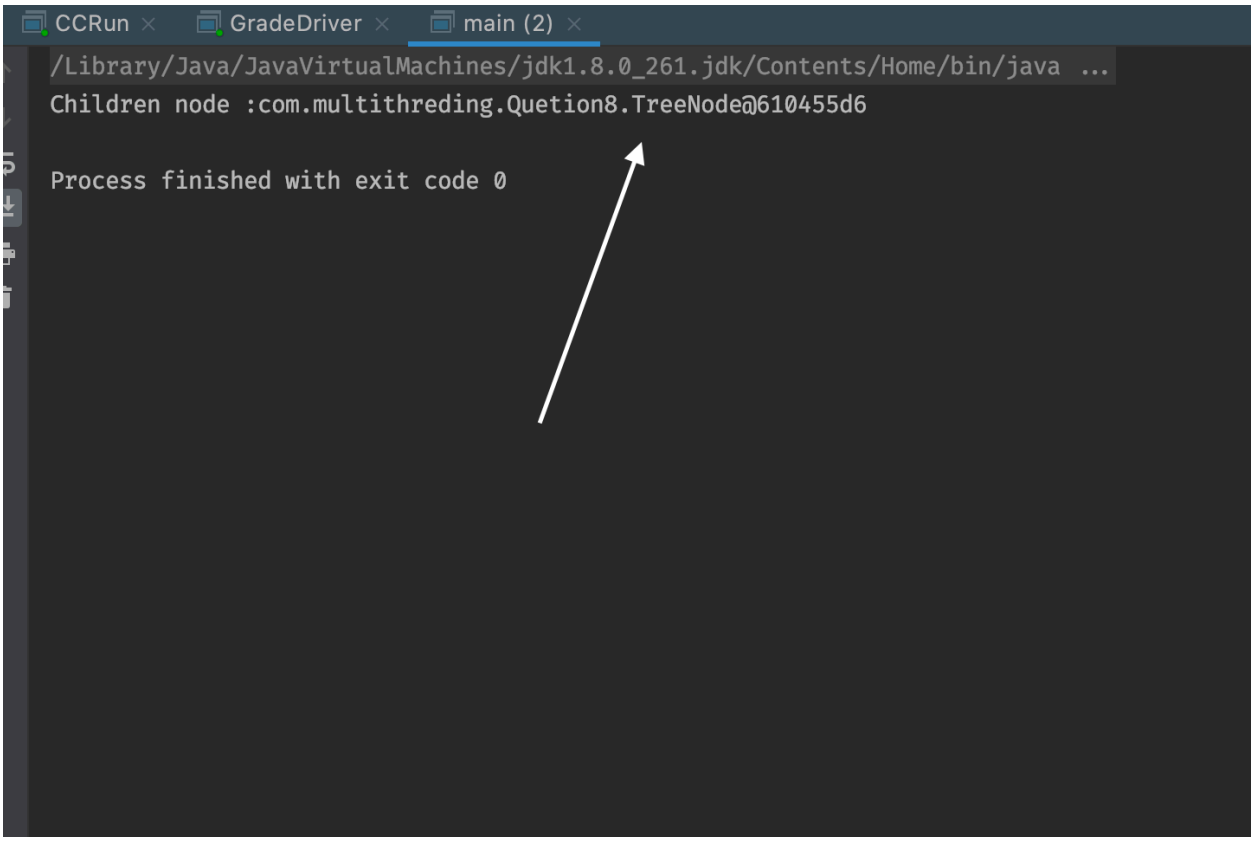
Thread 1: parent.addChild(child); //locks parent
--> child.setParentOnly(parent);

Thread 2: child.setParent(parent); //locks child
--> parent.addChildOnly()

- First thread 1 calls parent.addChild(child). Since addChild() is synchronized thread 1 effectively locks the parent object for access from other treads.

- Then thread 2 calls child.setParent(parent). Since setParent() is synchronized thread 2 effectively locks the child object for access from other threads.

B) Compile and Run this code.



C) Is Race Condition possible in this code, Yes/No, Why?

As all the methods are **synchronized race condition is not possible here.**

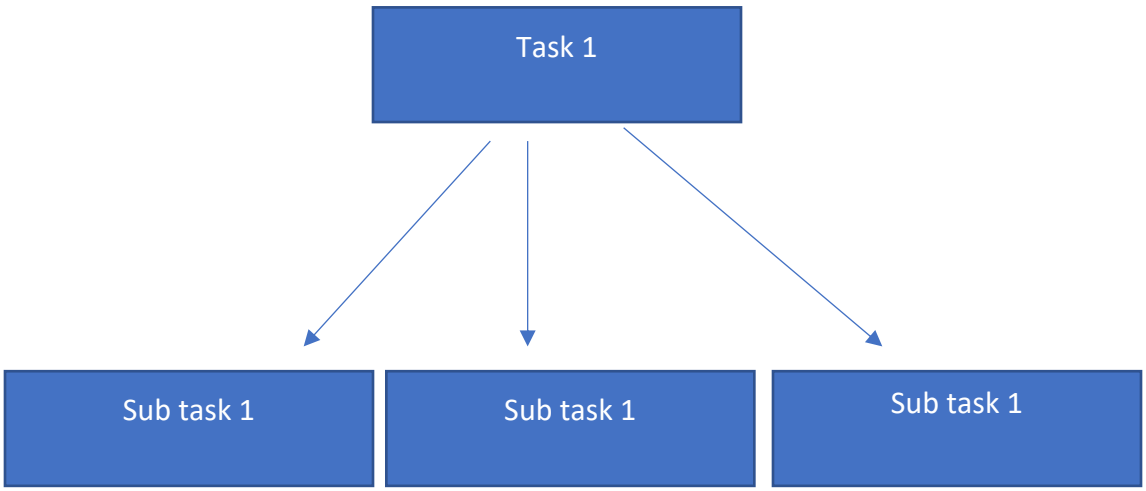
Assignment 6

1.Provide Descriptions

Parallelism versus Concurrency, provide diagram

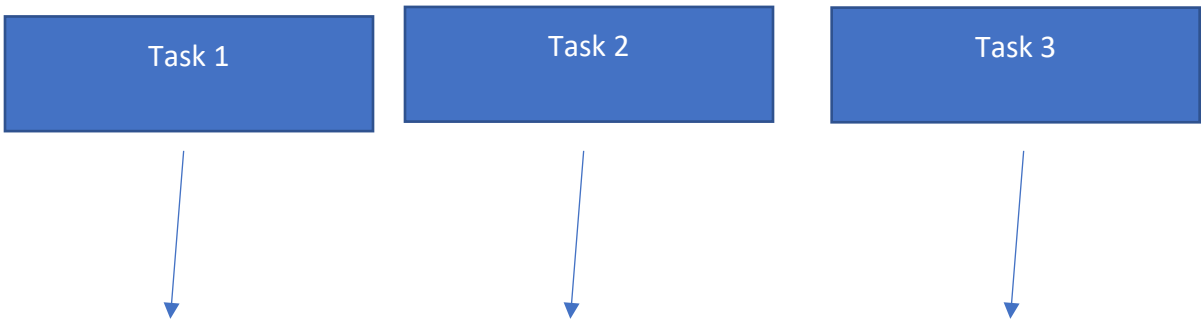
Parallelism :

Parallelism means that an **application splits its tasks up into smaller subtasks which can be processed in parallel**, for **instance on multiple CPUs at the exact same time.**



Concurrency :

Concurrency means that an **application is making progress on more than one task at the same time (concurrently).** if the computer only has one CPU the application may not make progress on more than one task at exactly the same time, but more than one task is being processed at a time inside the application. It does not completely finish one task before it begins the next. Instead, the CPU switches between the different tasks until the tasks are complete.



All task is running concurrently.

CPU core, How do you make cpu core?

A CPU core is the part of something central to its existence or character. In the same way in the computer system, the CPU is also referred to as the core.

There are basically two types of core processor:

- ➔ Single-Core Processor
- ➔ Multi-Core Processor

CPU Core recieve the instruction and perform the operation
A core can work on one task, while another core works a different task, so the more cores a CPU has, the more efficient it is.
In parallel execution, the tasks to be performed by a process are broken down into sub-parts, and multiple CPUs (or multiple cores) process each sub-task at precisely the same time.

Adding more core means designing more chip in CPU

32-bit versus 64-bit architecture, give example

- ➔ A 32-bit processor includes a 32-bit register, which can store 2^32 or 4,294,967,296 values.
- ➔ A 64-bit processor includes a 64-bit register, which can store 264 or 18,446,744,073,709,551,616 values. Therefore, a 64-bit register is not twice as large as a 32-bit register, but is 4,294,967,296 times larger.

- ➔ The CPU register stores memory addresses, which is how the processor accesses data from RAM. One bit in the register can reference an individual byte in memory, so a 32-bit system can address a maximum of 4 gigabytes (4,294,967,296 bytes) of RAM. The actual limit is often less – around 3.5 gigabytes – since part of the registry is used to store other temporary values besides memory addresses.

- ➔ A 64-bit register can theoretically reference 18,446,744,073,709,551,616 bytes, or 17,179,869,184 gigabytes (16 exabytes) of memory.

- ➔ This is several million times more than an average workstation would need to access. What's important is that a 64-bit computer (which means it has a 64-bit processor) can access more than 4 GB of RAM. **If a**

computer has 16 GB of RAM, it better have a 64-bit processor. Otherwise, at least 12 GB of the memory will be inaccessible by the CPU.

➔ While 64 bits is far more storage than what modern computers require, it removes all bottlenecks associated with 32-bit systems. For example, 64-bit systems run more efficiently since memory blocks are more easily allocated. They also support 64-bit instructions and have 64-bit data paths, which enables them to process more data at once than 32-bit systems can.

Client/Server Socket programming

➔ Client and server programs need to be able to establish connections to each other over the network. Client and server applications connect through methods defined by Transmission Control Protocol/Internet Protocol -- or TCP/IP -- standards.

➔ The server is linked or bound to a specific IP address on the network that is reachable by the client.

➔ Once a connection is established, the client sends the server whatever data and instructions are needed to complete the tasks it needs the server to perform.

➔ The client application takes the data it receives from the server, displays it, saves it to a storage device, prints it or forwards it to other clients on the network.

➔ The server program plays a passive role: it constantly checks its network connection to see if there are any clients sending it requests for a service it can provide.

➔ Once the server program detects a service request, it establishes the socket and begins receiving and sending information to the client until the job is done.

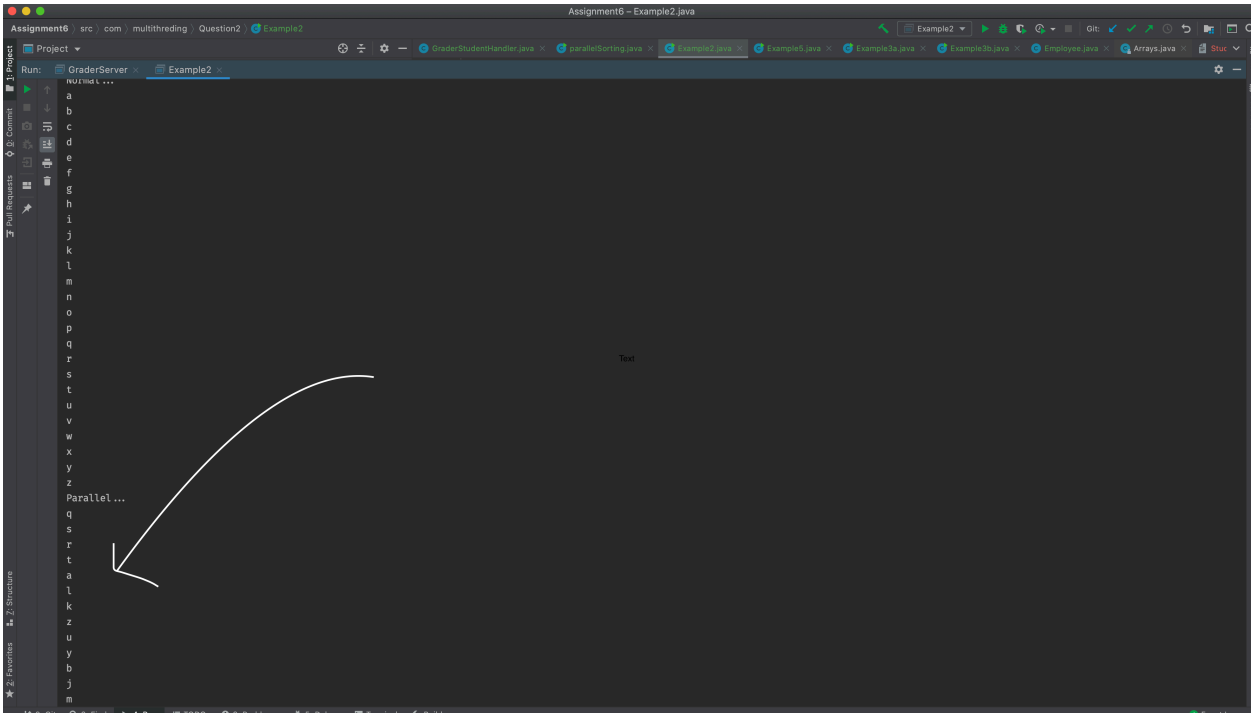
➔ Finally, the server closes the connection to the client and resumes the wait for more client program requests for services.

2. You can run tasks in parallel using Java8 Collection.parallelStream. Consider ParallelExample2, ParallelExample3a, ParallelExample3b, and ParallelExample5. Compile and run. Explain how the code works in each program and discuss the Results.

ParallelExample2:

- In This Example we are printing the value of a to z using stream() and parallel stream.
- Output of stream() method is **in sequence** on the other hand output of parallel stream is **not in sequential manner.**

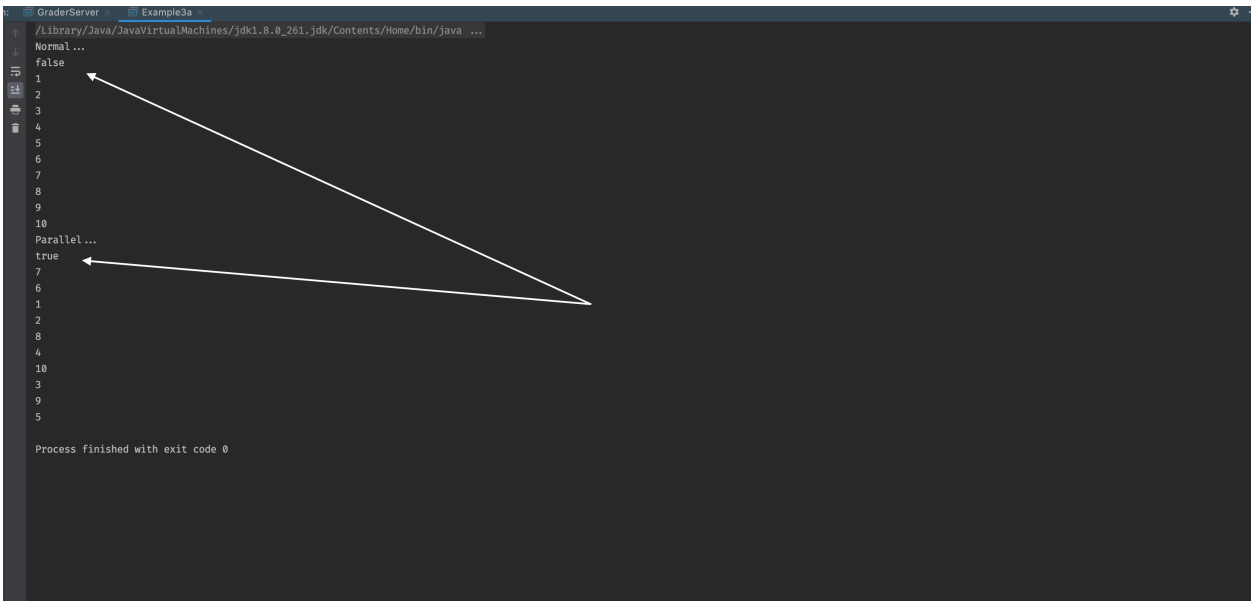
Output



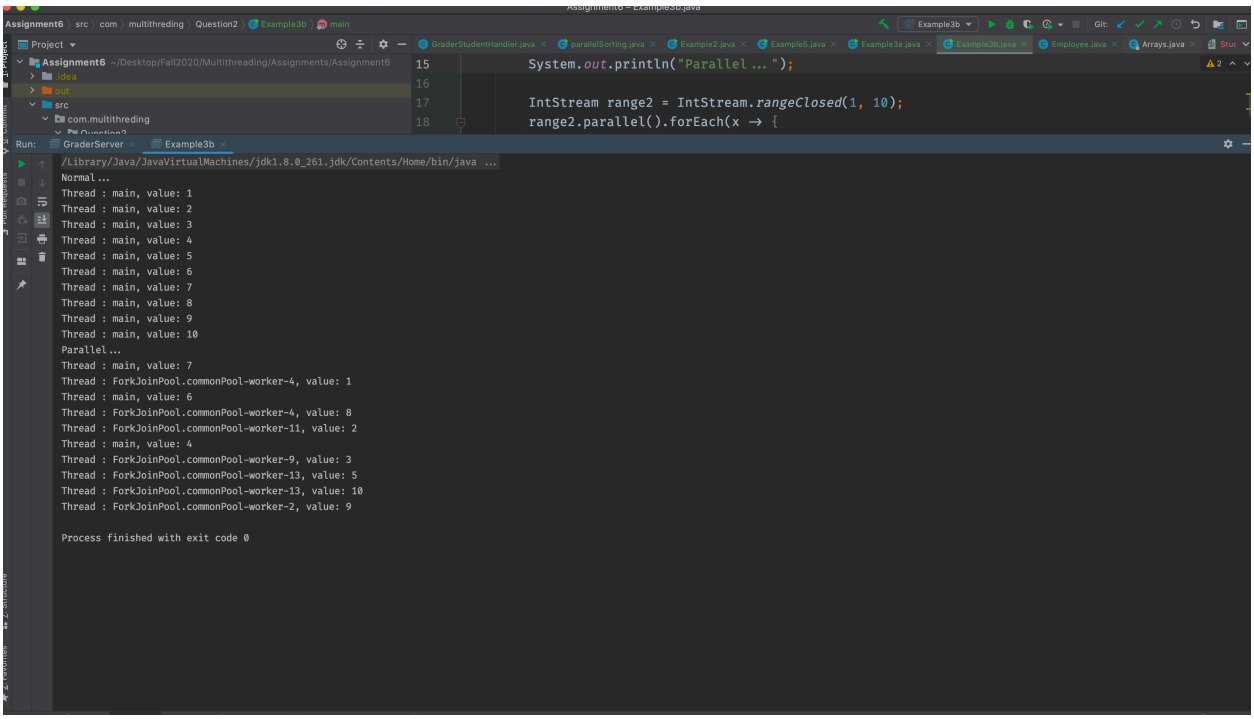
ParallelExample3a

→In This example first we are generating range of values .

→ First we are printing the values with For Each and verifying that operation is parallel or not using isParallel() method.
→ In Output we can clearly check that values are printing in sequential manner
→ after that we are making that range parallel using parallel() method and we can observe the output is not printing the value in sequential manner.



ParallelExample3b



→ In Example 3b are printing the currently running thread name and we can see that the first foreach loop is ran by Main- thread only while parallel method creates the multiple threads using forkJoin threadpool and print the values.

ParallelExample5

→ In this example we are comparing the performance of the sequential vs parallel method

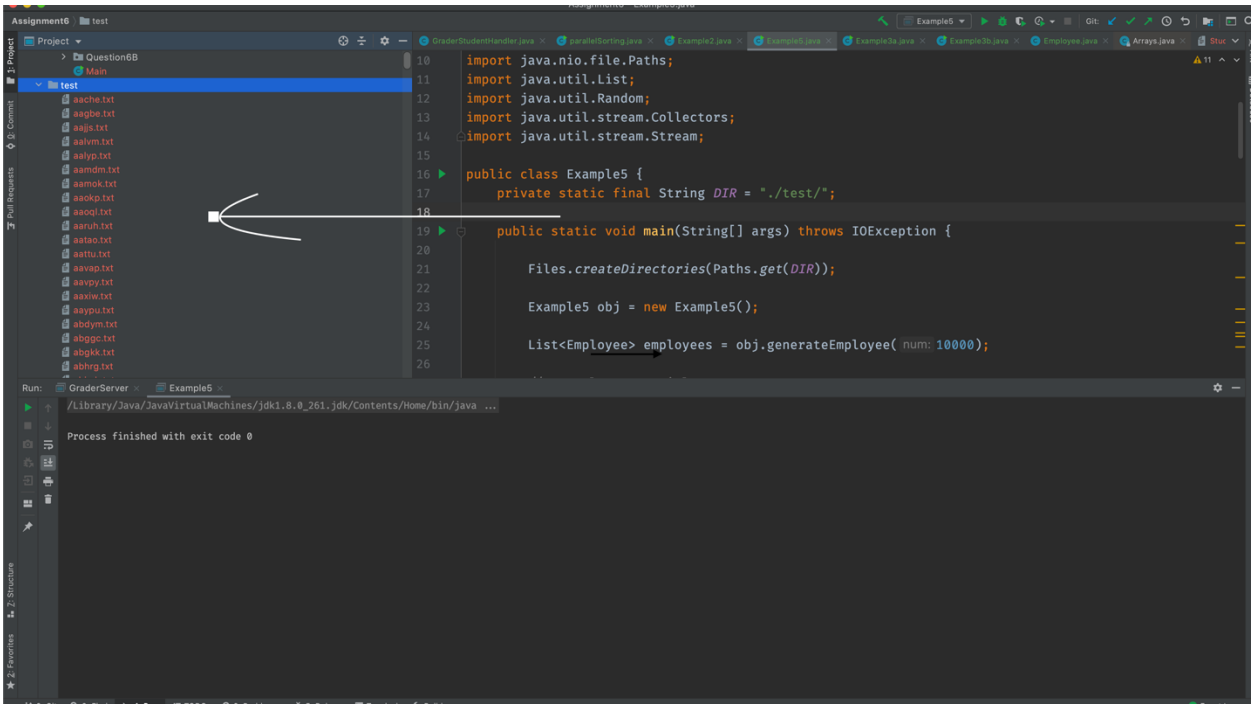
➔ We have created one employee class with all the employee details.

➔ Inside the main method we are creating **10000** employee and store the data of each employee in different file

➔ First time we are creating 10000 employee's files using **stream() (sequential) method and observer the running time.**

➔ **In second time we will run the program using parallel stream and observer running time.**

➔ **If compare the running time of both the program we can clearly say that parallel stream is taking ¼ time of stream method.**


Assignment6
test
Example5
Git:
Project
Question6B
Main
test
aache.txt
aagbe.txt
aajjs.txt
aalvm.txt
aalyp.txt
aamdm.txt
aamok.txt
aaokp.txt
aaoql.txt
aaruh.txt
aatao.txt
aattu.txt
aavap.txt
aavpy.txt
aaxiw.txt
aaypu.txt
abdym.txt
abggc.txt
abgkk.txt
abhrg.txt
GraderStudentHandler.java
parallelSorting.java
Example2.java
Example5.java
Example3a.java
Example3b.java
Employee.java
Arrays.java
import java.nio.file.Paths;
import java.util.List;
import java.util.Random;
import java.util.stream.Collectors;
import java.util.stream.Stream;
public class Example5 {
private static final String DIR = "./test/";
public static void main(String[] args) throws IOException {
Files.createDirectories(Paths.get(DIR));
Example5 obj = new Example5();
List<Employee> employees = obj.generateEmployee(num: 10000);
Run:
GraderServer
Example5
/Library/Java/JavaVirtualMachines/jdk1.8.0_261.jdk/Contents/Home/bin/java ...
Process finished with exit code 0

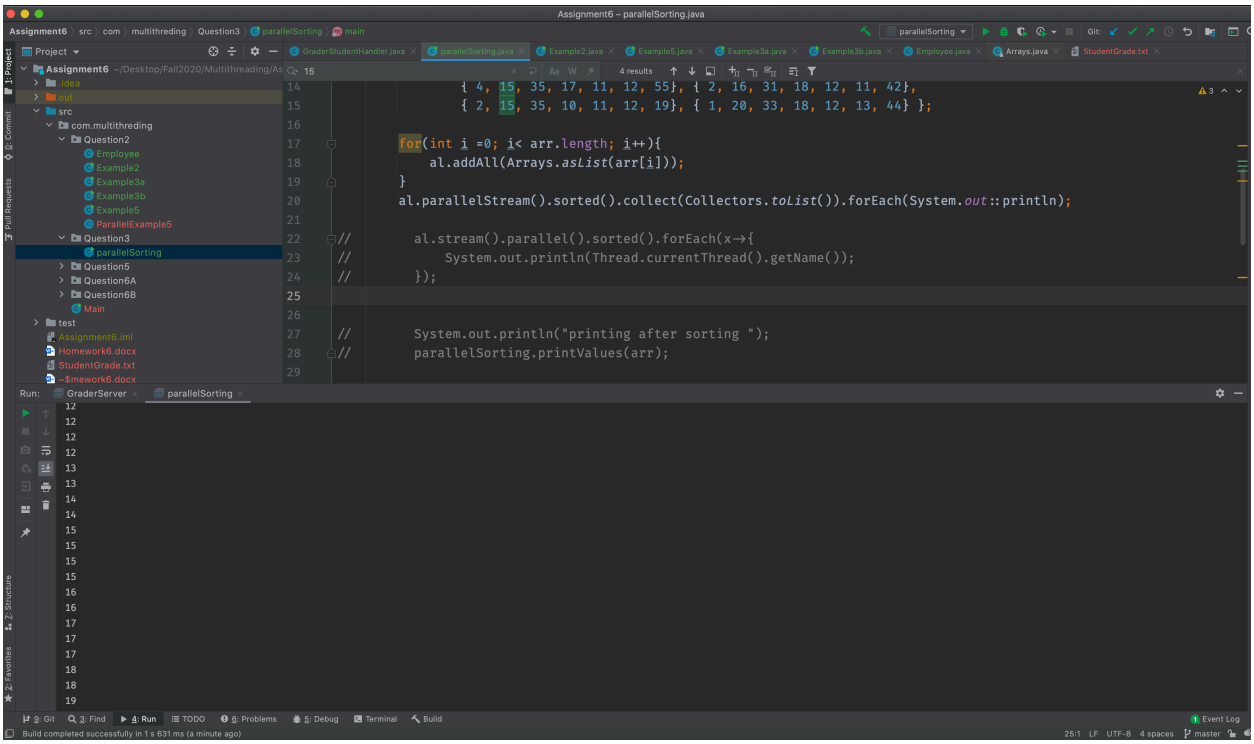
3. Write Java parallelism code to sort each array entry in two dimensional array input data. Note: You need to use Java8 Collection.parallelStream, and run eight threads in parallel.

```
  int[][] arr = { { 9, 12, 6, 14, 10, 21, 13}, { 3, 5, 41, 16, 14, 10,
21},
                { 3, 15, 41, 17, 11, 10, 51}, { 3, 15, 41, 17, 11, 10, 51},
                { 4, 15, 35, 17, 11, 12, 55}, { 2, 16, 31, 18, 12, 11, 42},
                { 2, 15, 35, 10, 11, 12, 19}, { 1, 20, 33, 18, 12, 13, 44}
};
```

➔ To use Collection.parallel stream first we need to store all the values in list anf then we can sort the list using parallelStream.

List.ParrallelStream().sort()

Output:



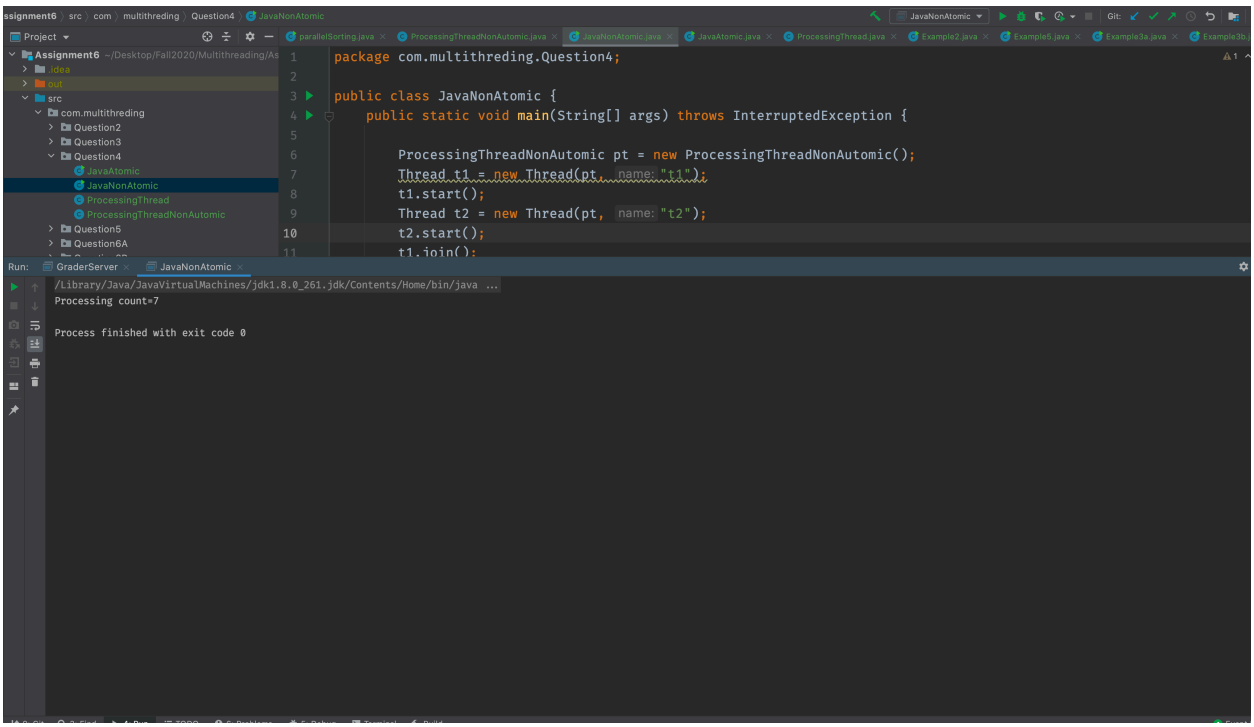
4. Consider Java AtomicInteger example in this article, initially Part-1 is implemented Without integer atomic operation, and part-2 is implemented With integer atomic operation.

https://www.journaldev.com/1095/atomicinteger-java

a) Compile and run Part-1, report and explain the results and code Problem

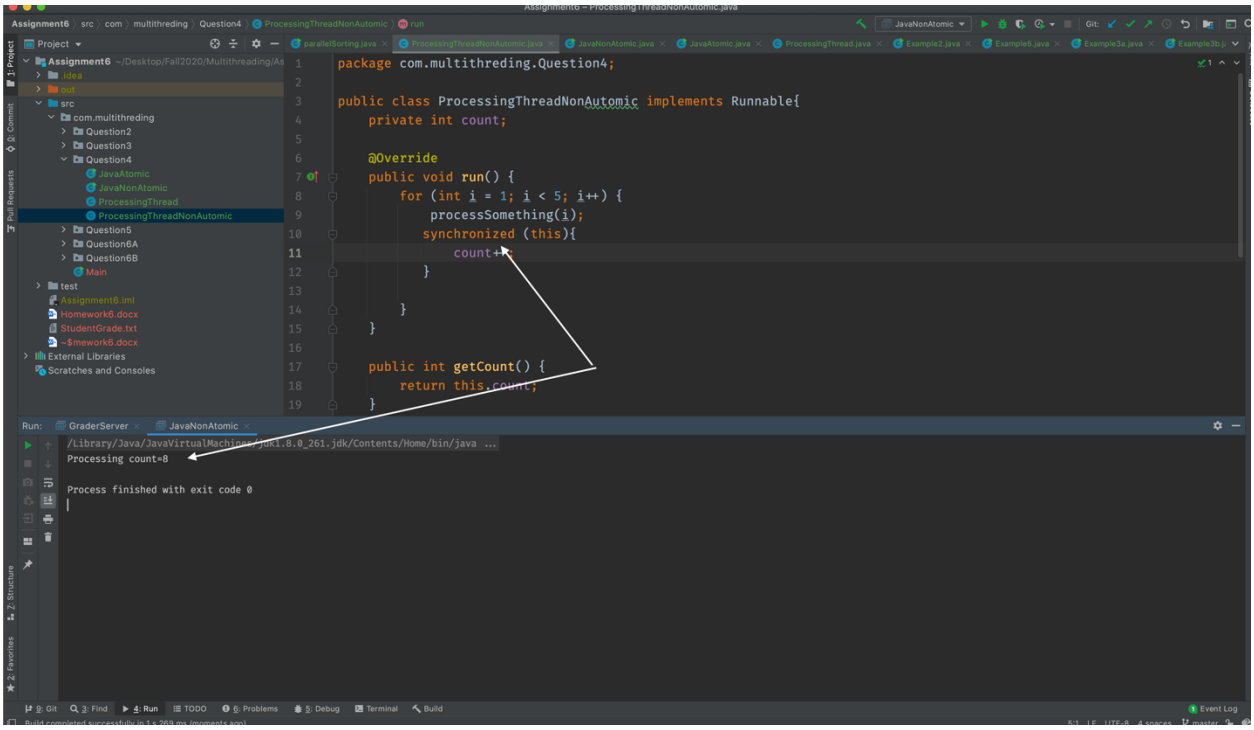
➔ In Part-1 we have are trying to update the value of shared resource without implementing any locking mechanism.

➔ As we can see in output that the value of count is varying.



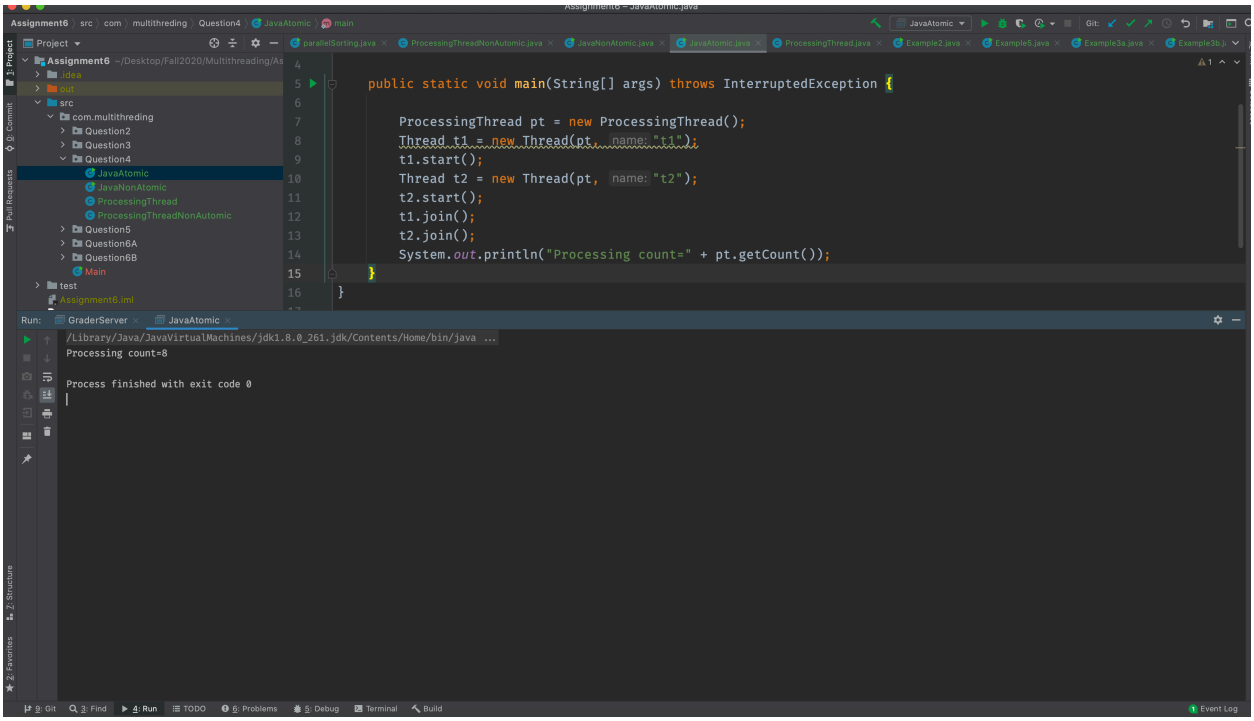
b) Fix the problem in Part-1 without Atomic Operation, Compile and run

➔ We can fix this problem by putting count increament operation inside synchronized block or by making getcount method synchronized.



c) Compile and run Part-2 program which uses Atomic operation

- In part 2 we have used the AtomicIntger count variable which will make sure that the increment operation is atomic by using **increamentAndGetMethod() hence every time we will get the same output.**



d) Compare results between (a) and (c), and then compare results between (b) and (c)

➔ In Result (a) we get the different value count variable because we have not used any locking mechanism to protect the value of count.

➔ In result (c) every time we will get the same value of count variable because we have used the atomic integer which make sure that the increment operation should be atomic

→ We will get the same result for (b) and (c) as in result be we have used the synchronized block to protect the value of count variable.

➔ **But result(c) is improve the performance as it has used the atomic operation hence we don't need to worry about any blocking**

5. In socket programming, there is server side and there is client(s) side. Consider this code discussed in class: https://www.geeksforgeeks.org/introducing-threads-socket-programming-java/

a) Explain the client/server code

b) Compile the server side code and run it. Explain what you see and why?

c) Compile client-side and run it. Explain what you see and why?

a) Explain the client/server code

➔ In Given code, we are creating one server class which will handle the client accept the client request and respond to that client.

➔ To handle the multiple client at the same time we have created the clientHandler class which implements the runnable interface.

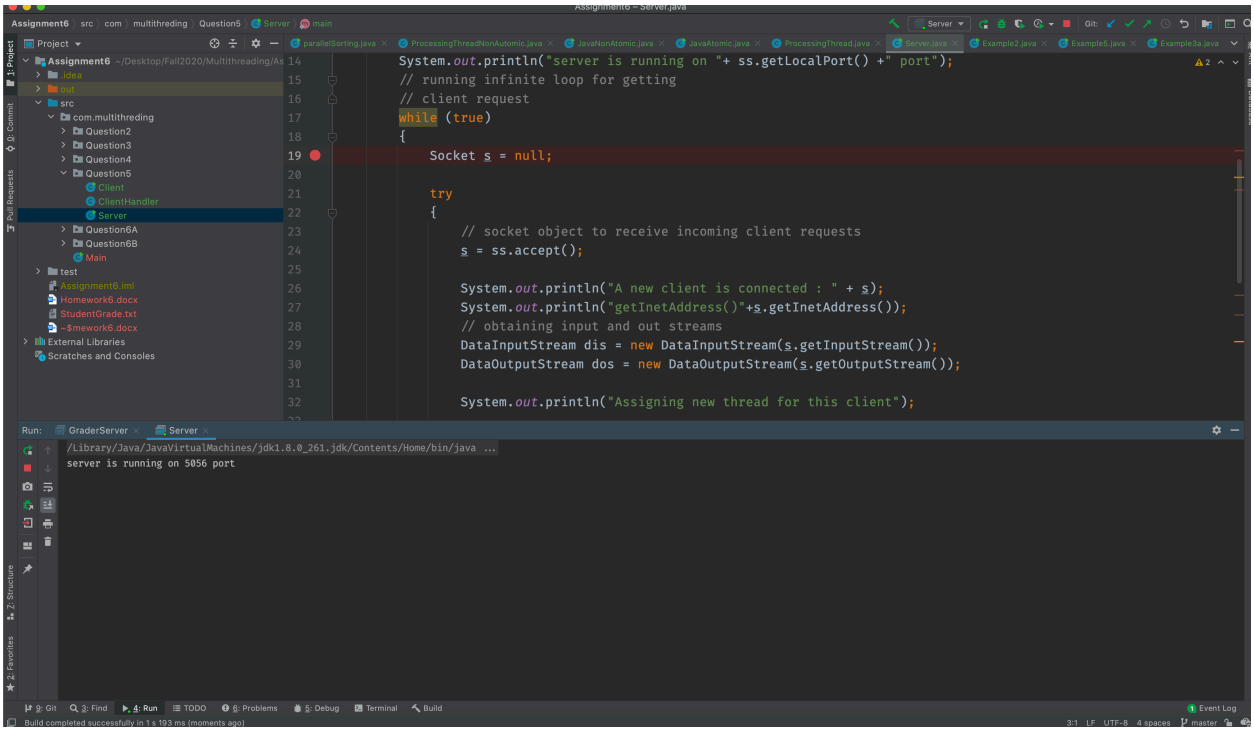
➔ So every time when new client send a request to a server server will create separate thread of client handler class which will respond to that particular client request.

b) Compile the server side code and run it. Explain what you see and why?

→ When we comple and run the sever code it will start the sever on given port number.

➔ Once the server started it will continuously look for any client request.

➔ If client try to connect server then it will accept the request and create new Thread of client hander class to process that request.

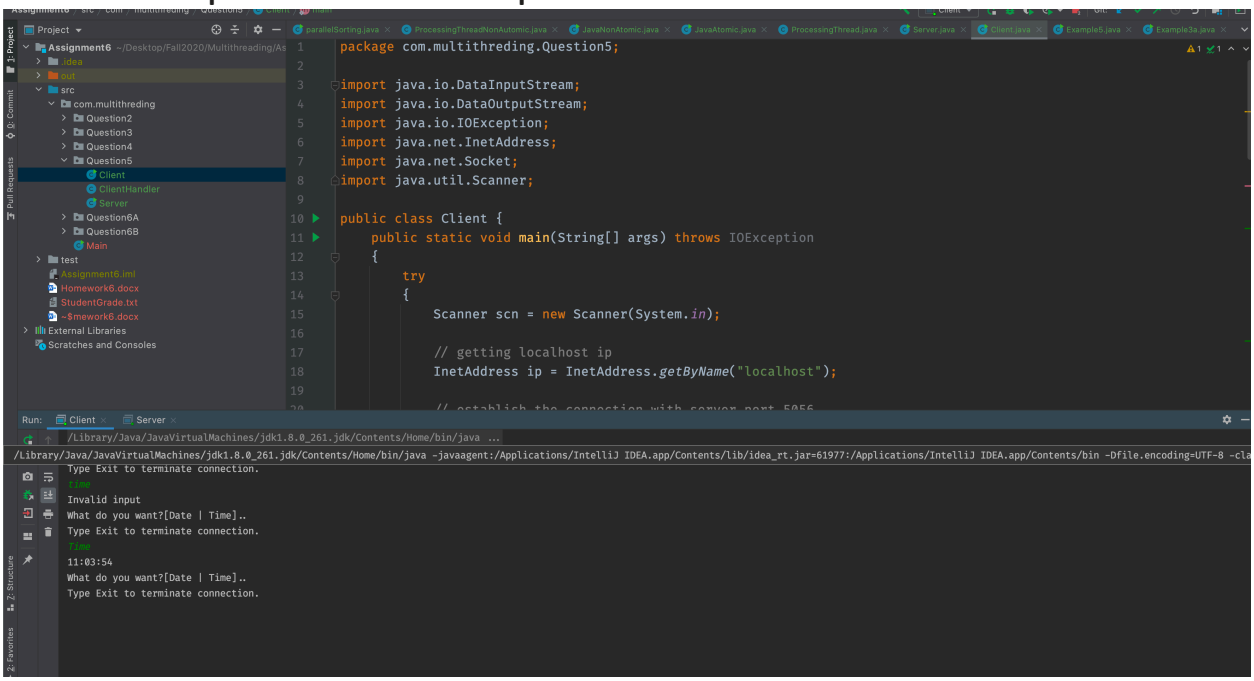


c) Compile client-side and run it. Explain what you see and why?

➔ When we compile and run the client side code it will connect to sever and server will create a new thread of client handler.

➔ Now client handler will prompt to client that which thing you want time or date

➔ When client send the valid input , client handler will respond to that request.



6. You have learned how to create multi-threaded Student grading system using both implicit locking and explicit locking. Now you are to create student grading system using Socket client/server programming.

A) Consider example “A Network Tic-Tac-Toe Game” in the this article, analyze,

compile and run the code:

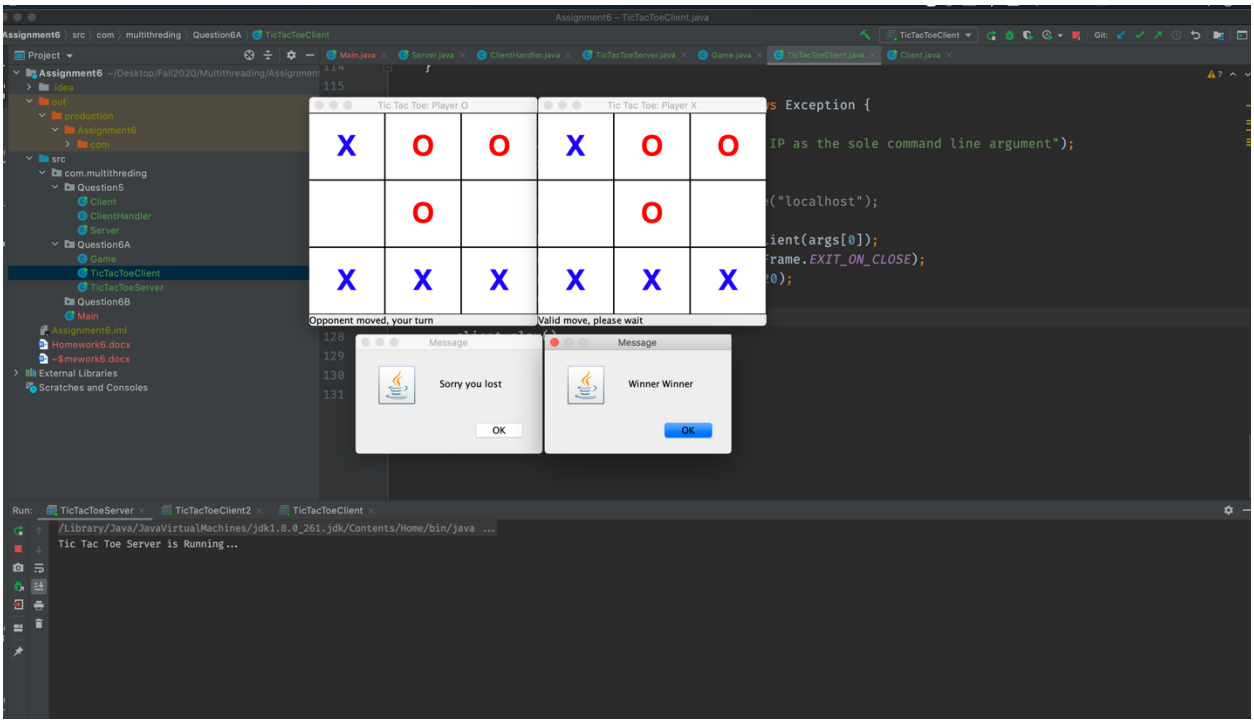
https://cs.lmu.edu/~ray/notes/javanetexamples/

Code Analysis :

→ we have create on TicTacToeServer class which will act as sever.In this class we are trying to connect two players.

- ➔ We have created ThreadPool to process each request in separate thread.
- ➔ Game class will validate the basic rule of the game while player class implements the runnable interface so each player have their own thread.
- ➔ While TictacToe client class has UI Part where we can load the Jframe and start the game.

Output:



B) Student Program with Client server approach

B) Use example model to build student grading system with 40 client student threads and one server GraderThread. Each client-student thread generates 3 scores (homework, midterm,

final), and each score is randomly generated between 70 to 100. The scores are stored in ScoresMap as key/value, ({homework, score}, {midterm, score}, {final, score}). Then, each student thread connects to the GraderServer (with incremental 100 milliseconds interval), and once it is connected to the server, it sends the scoresMap, studentId, firstName, lastName. Note: You need to create a thread within GraderServer to connect to each client. The GraderServer thread receives student's information: a) parses the ScoresMap for student scores, calculates letter grade A, B, C, D, F, and stores letter grade in its local cache GradesMap (threadId, grade), and then writes it to the file "FinalGrades", and replies grade back to the client student thread.
b) In subsequent student requests with missing/mistake grade (updated specific score in ScoresMap), the Server parses the scoresMap information, Verifies grade information in cache, calculates letter grade, updates cache, updates file, and replies the new grade back to student thread.
Note: you need to create cache key/value

Approach :

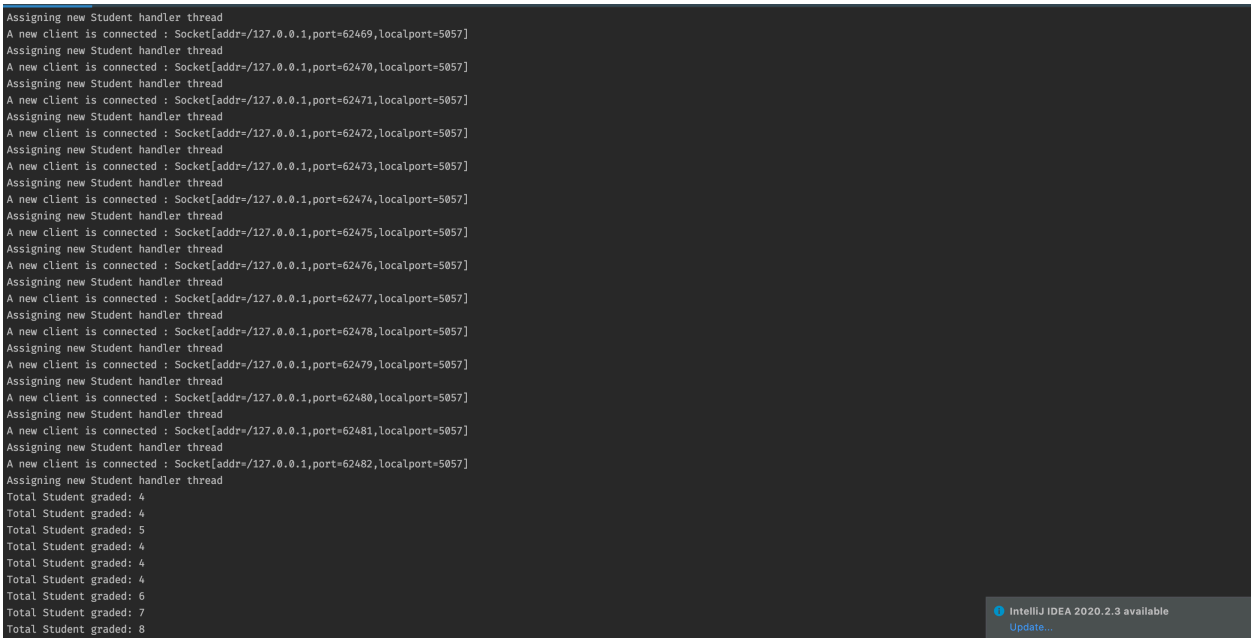
- ➔ Created GraderSever class which will act like sever and accept the request of each student.
- ➔ To handler the multiple student request concurrently I have created GradeStudentHandler class.
- ➔ Every time a student send the score to graderserver it will create the separate thread of GraderStudent handler class and calculate the letter grade and save it

into grade file and local cache and at the end it will send grade it back to that particular student.

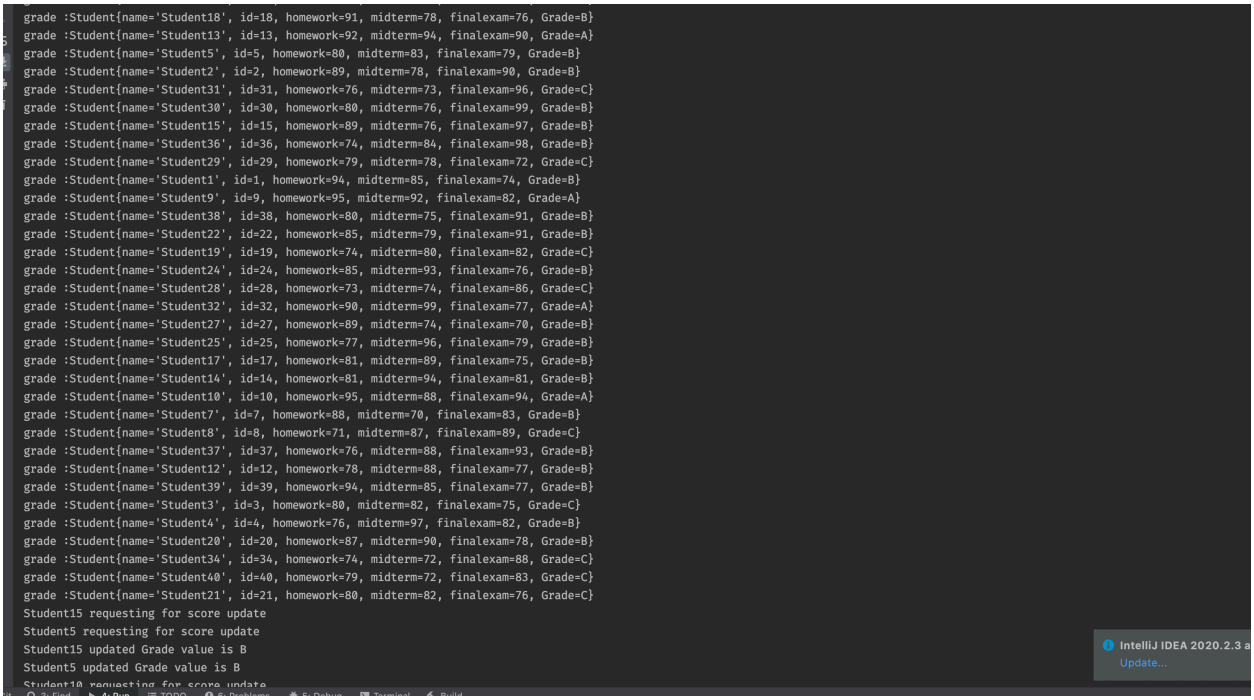
➔ To Handle the different types of request such as missing/mistake grade update and report new grades i will send the value of requesttype to Gradersever and it will process the request according to requesttype.

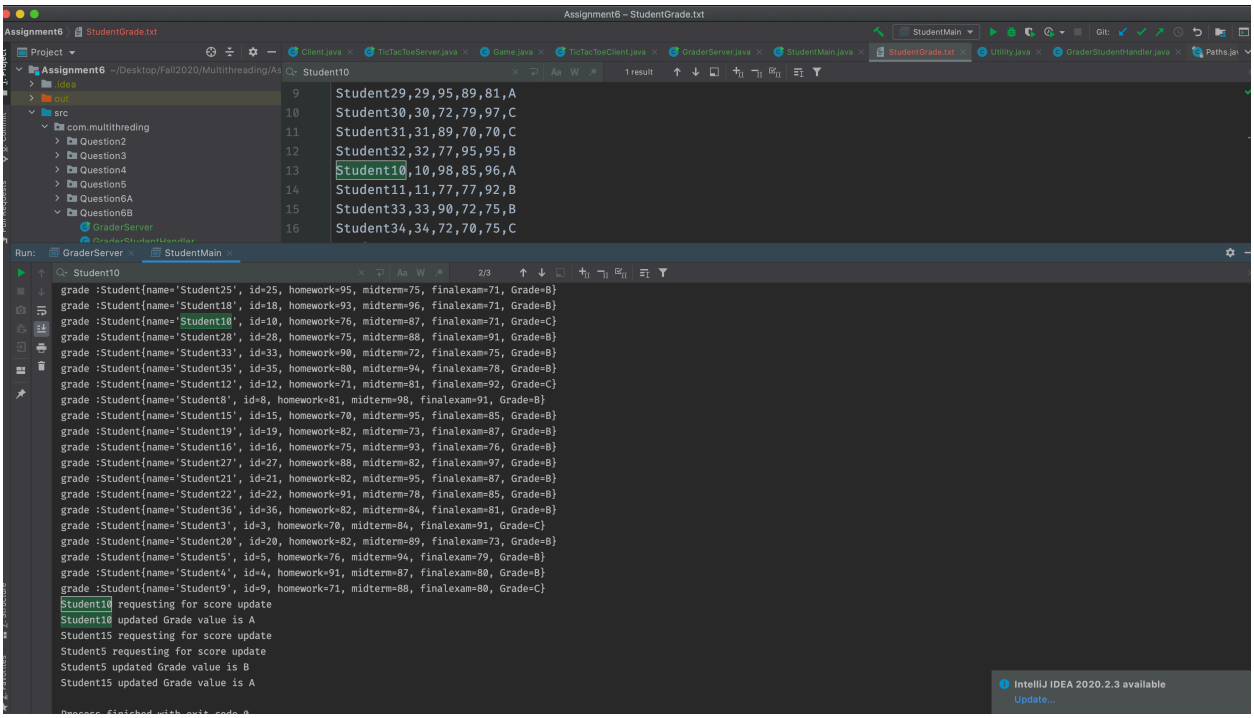
➔ For Implementation details please revive the code

Output:



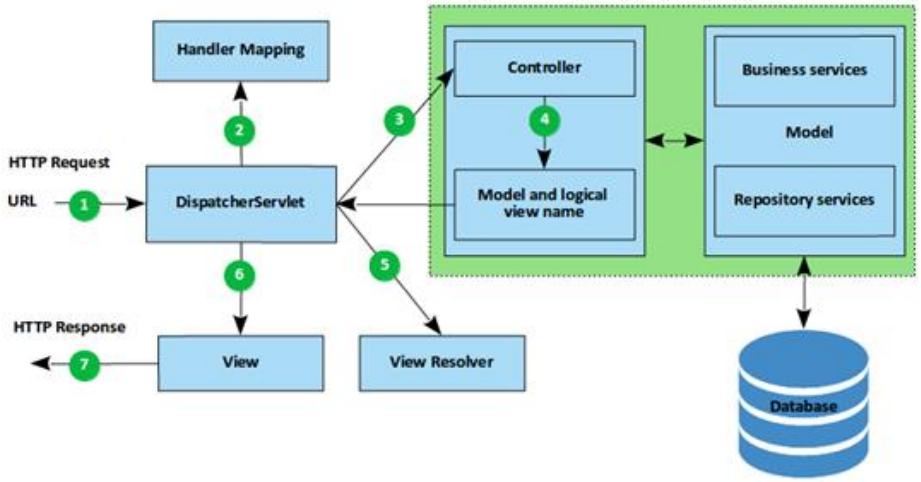

GraderServer
StudentMain
Thread-13 is Reading grade
Thread-1 is Reading grade
Thread-2 is Reading grade
Thread-17 is Reading grade
Thread-23 is Reading grade
Thread-24 is Reading grade
Thread-15 is Reading grade
Thread-18 is Reading grade
Thread-7 is Reading grade
Thread-12 is Reading grade
Thread-6 is Reading grade
Thread-3 is Reading grade
Thread-20 is Reading grade
Thread-22 is Reading grade
Inside run method
Inside run method
Thread-14 is Reading grade
Thread-8 is Reading grade
Thread-28 is submitting score
Thread-27 is submitting score
Thread-10 is Reading grade
Thread-16 is Reading grade
Thread-9 is Reading grade
Inside run method
Thread-29 is submitting score
Thread-28 is Reading grade
Thread-27 is Reading grade
Thread-29 is Reading grade
Inside run method
Thread-30 is submitting score
Thread-30 is Reading grade
Inside run method
Thread-31 is submitting score
Inside run method
Thread-32 is submitting score
Thread-31 is Reading grade
Thread-32 is Reading grade
Inside run method
Thread-33 is submitting score
IntelliJ IDEA 2020.2.3 available
Update...


grade :Student{name='Student18', id=18, homework=91, midterm=78, finalexam=76, Grade=B}
grade :Student{name='Student13', id=13, homework=92, midterm=94, finalexam=90, Grade=A}
grade :Student{name='Student5', id=5, homework=80, midterm=83, finalexam=79, Grade=B}
grade :Student{name='Student2', id=2, homework=89, midterm=78, finalexam=90, Grade=B}
grade :Student{name='Student31', id=31, homework=76, midterm=73, finalexam=96, Grade=C}
grade :Student{name='Student30', id=30, homework=80, midterm=76, finalexam=99, Grade=B}
grade :Student{name='Student15', id=15, homework=89, midterm=76, finalexam=97, Grade=B}
grade :Student{name='Student36', id=36, homework=74, midterm=84, finalexam=98, Grade=B}
grade :Student{name='Student29', id=29, homework=79, midterm=78, finalexam=72, Grade=C}
grade :Student{name='Student1', id=1, homework=94, midterm=85, finalexam=74, Grade=B}
grade :Student{name='Student9', id=9, homework=95, midterm=92, finalexam=82, Grade=A}
grade :Student{name='Student38', id=38, homework=80, midterm=75, finalexam=91, Grade=B}
grade :Student{name='Student22', id=22, homework=85, midterm=79, finalexam=91, Grade=B}
grade :Student{name='Student19', id=19, homework=74, midterm=80, finalexam=82, Grade=C}
grade :Student{name='Student24', id=24, homework=85, midterm=93, finalexam=76, Grade=B}
grade :Student{name='Student28', id=28, homework=73, midterm=74, finalexam=86, Grade=C}
grade :Student{name='Student32', id=32, homework=90, midterm=99, finalexam=77, Grade=A}
grade :Student{name='Student27', id=27, homework=89, midterm=74, finalexam=70, Grade=B}
grade :Student{name='Student25', id=25, homework=77, midterm=96, finalexam=79, Grade=B}
grade :Student{name='Student17', id=17, homework=81, midterm=89, finalexam=75, Grade=B}
grade :Student{name='Student14', id=14, homework=81, midterm=94, finalexam=81, Grade=B}
grade :Student{name='Student10', id=10, homework=95, midterm=88, finalexam=94, Grade=A}
grade :Student{name='Student7', id=7, homework=88, midterm=70, finalexam=83, Grade=B}
grade :Student{name='Student8', id=8, homework=71, midterm=87, finalexam=89, Grade=C}
grade :Student{name='Student37', id=37, homework=76, midterm=88, finalexam=93, Grade=B}
grade :Student{name='Student12', id=12, homework=78, midterm=88, finalexam=77, Grade=B}
grade :Student{name='Student39', id=39, homework=94, midterm=85, finalexam=77, Grade=B}
grade :Student{name='Student3', id=3, homework=80, midterm=82, finalexam=75, Grade=C}
grade :Student{name='Student4', id=4, homework=76, midterm=97, finalexam=82, Grade=B}
grade :Student{name='Student20', id=20, homework=87, midterm=90, finalexam=78, Grade=B}
grade :Student{name='Student34', id=34, homework=74, midterm=72, finalexam=88, Grade=C}
grade :Student{name='Student40', id=40, homework=79, midterm=72, finalexam=83, Grade=C}
grade :Student{name='Student21', id=21, homework=80, midterm=82, finalexam=76, Grade=C}
Student15 requesting for score update
Student5 requesting for score update
Student15 updated Grade value is B
Student5 updated Grade value is B
IntelliJ IDEA 2020.2.3
Update...



7) Spring MVC design pattern

7. Consider the following ModelViewControl (MVC) model. Use this Model to identify and design three services for the University myNortheastern blackboard portal. You need to show the controller layer, service layer, and database layer for each of the services that you identify. For example: Login is a service. You provide http URL: myNortheastern/profile/login
Note: Read article: https://www.developer.com/java/data/exploring-rest-apis-with-spring-mvc.html



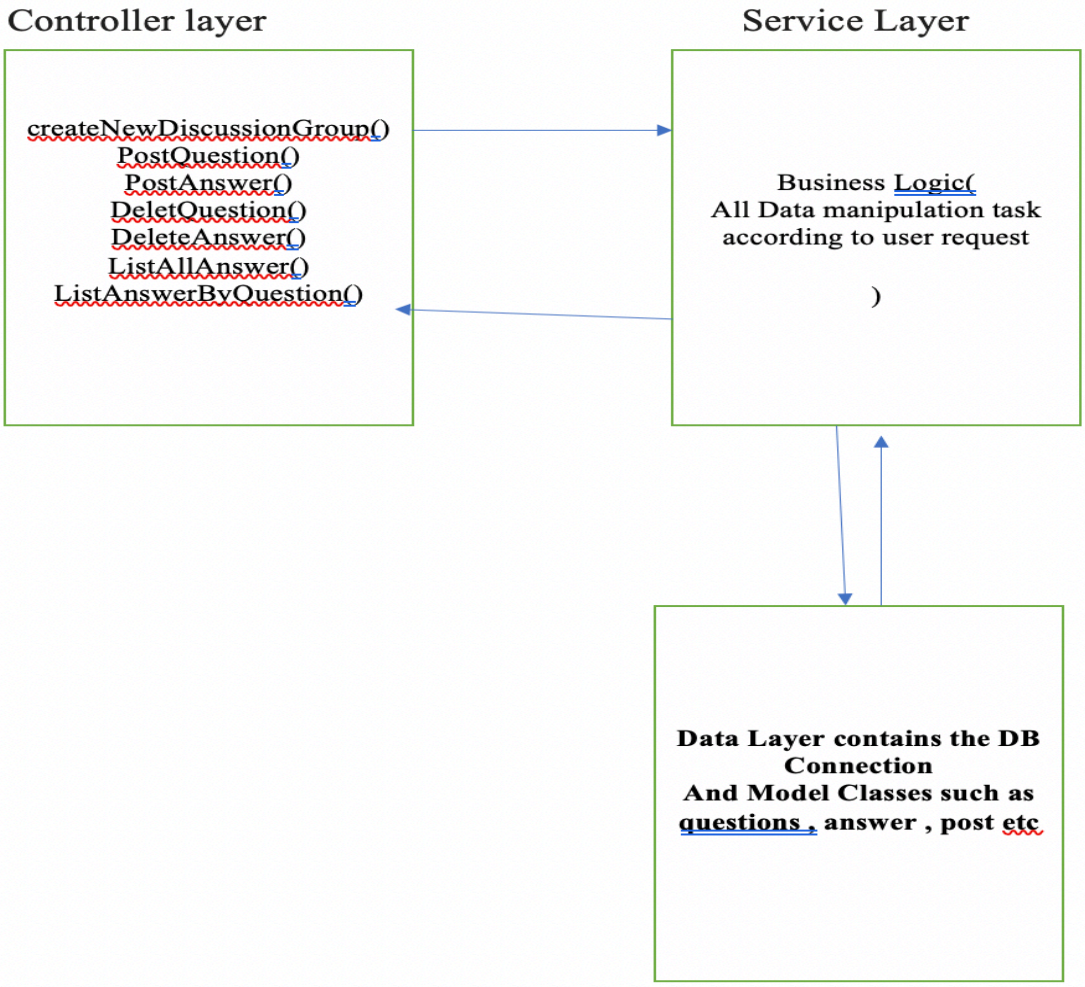
controller layer Identify the request and then forward it to respective service layer

Service layer -- Connect to Database layer and manipulate data according to business logic

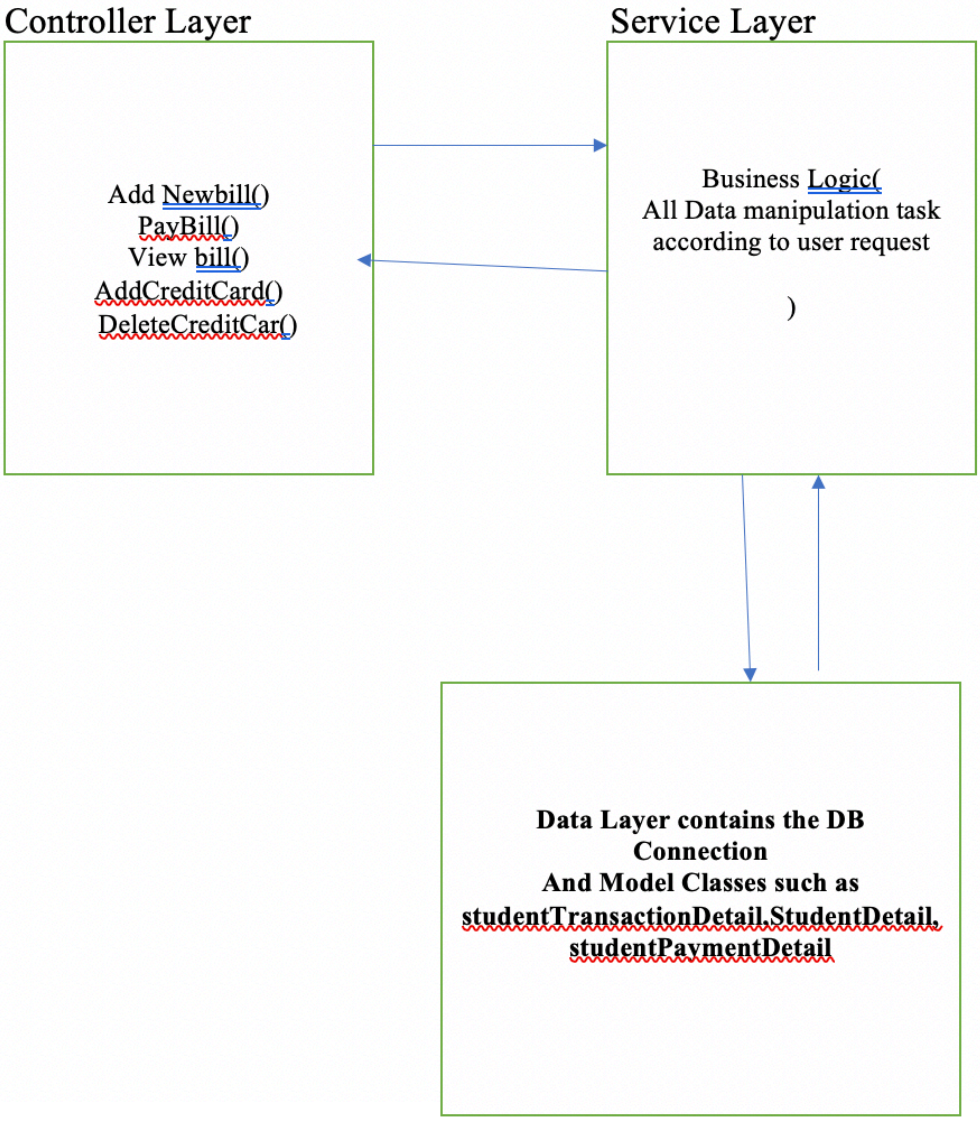
Database layer --> Established connection with database and retrieve data and pass to service layer

In Blackboard portal

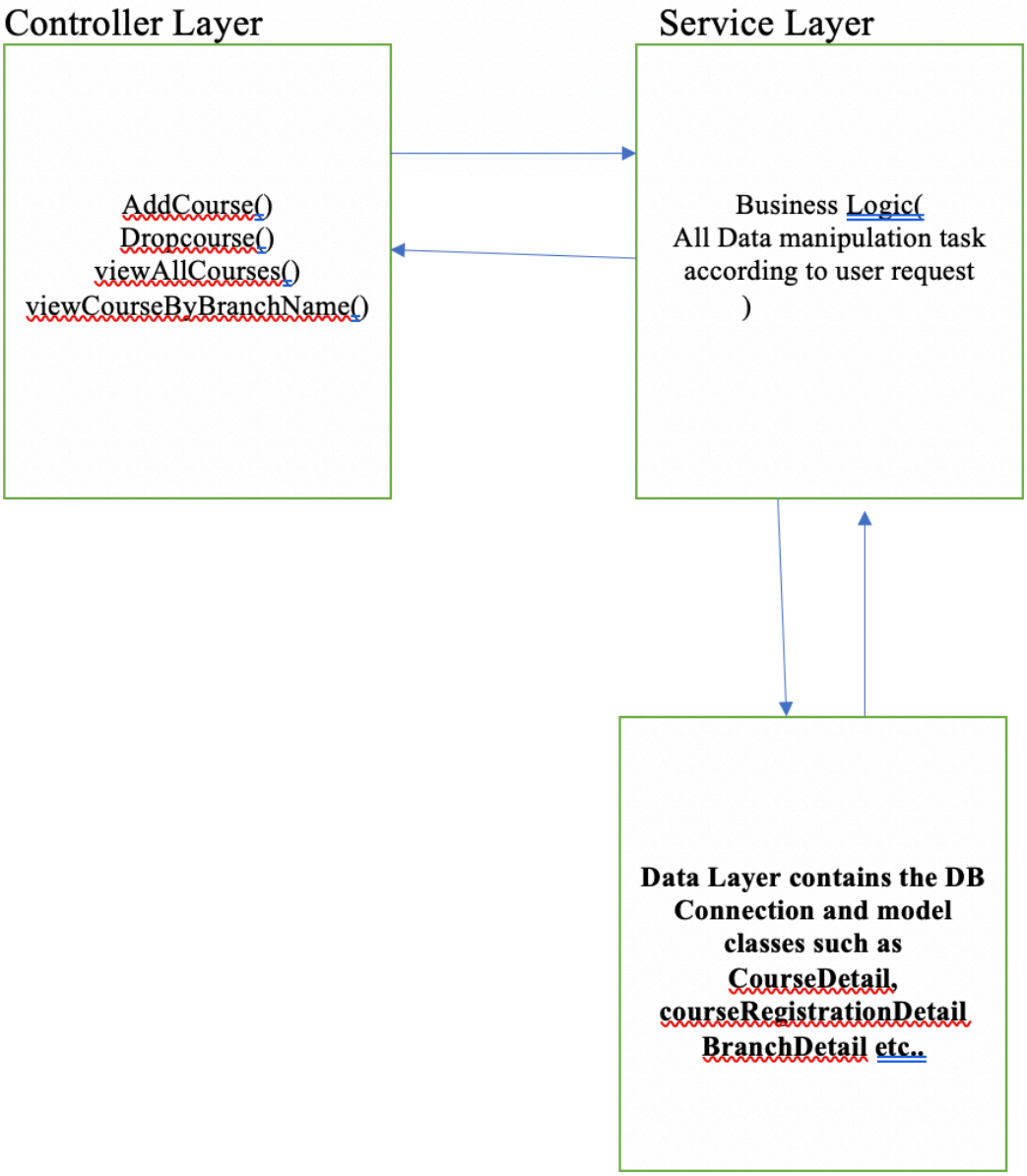
Discussion Forum



Billing


Controller Layer
Service Layer
Add Newbill()
PayBill()
View bill()
AddCreditCard()
DeleteCreditCar()
Business Logic(
All Data manipulation task
according to user request
)
Data Layer contains the DB
Connection
And Model Classes such as
studentTransactionDetail,StudentDetail,
studentPaymentDetail

Course Registration



Assignment 7

1. Provide detail descriptions:

Blocking Algorithms

A blocking concurrency algorithm is an algorithm which either:

➔ Performs the action requested by the thread – OR
➔ Blocks the thread until the action can be performed safely

- Blocking algorithms **block the thread until the requested action can be performed**.

- Usually it will be the actions of another thread that makes it possible for the first thread to perform the requested action.

- If for some reason that other thread is suspended (blocked) somewhere else in the application, and thus cannot perform the action that makes the first thread's requested action possible, the first thread remains blocked

- either indefinitely, or until the other thread finally performs the necessary action.

Non-Blocking Algorithms

Non blocking algorithm use low-level atomic machine instructions such as compare and swap operation instead of locks to ensure data integrity under concurrent access

A non-blocking concurrency algorithm is an algorithm which either:

- ➔ Performs the action requested by the thread – OR
- ➔ Notifies the requesting thread that the action could not be performed

- Non blocking algorithms are considerably more complicated to design and implement than lock based alternatives, but they cab offer significant scalability and liveness advantages

- They coordinate at a finer level of granularity and can greatly reduce scheduling overhead because they don't block when multiple threads contend for the same data.

- In lock based algorithms, other threads cannot make progress if a thread goes to sleep or spins while holding a lock, whereas nonblocking algorithms are impervious to individual thread failure .

CAS operation

➔ CAS has **three operands** a memory location V on which to operate, the expected old value A, and the new value B. CAS atomically updates V to the new value B,but only if the value in V matches the expected old value A; otherwise it does nothing.

➔ it proceeds with the update in the hope of success, and can detect failure if another thread has updated the variable since it was last examined

➔ When multiple threads attempt to update the same variable simultaneously using CAS, one wins and updates the variable's value, and the rest lose. But the losers are not punished by suspension, as they could be if they failed to acquire a lock; instead, they are told that they didn't win the race this time but can try again.

➔ Because a thread that loses a CAS is not blocked, it can decide whether it wants to try again, take some other

recovery action, or do nothing. This flexibility eliminates many of the liveness hazards associated with locking.

➔ The typical pattern for using CAS is first to read the value A from V, derive the new value B from A, and then use CAS to atomically change V from A to B so long as no other thread has changed V to another value in the meantime. CAS addresses the problem of implementing atomic read-modify-write sequences without locking, because it can detect interference from other threads.

2. Consider the following code segments:

a) Compile each Java code and run with “javap” to disassemble the code

b) Show Stack Frame for each code segment

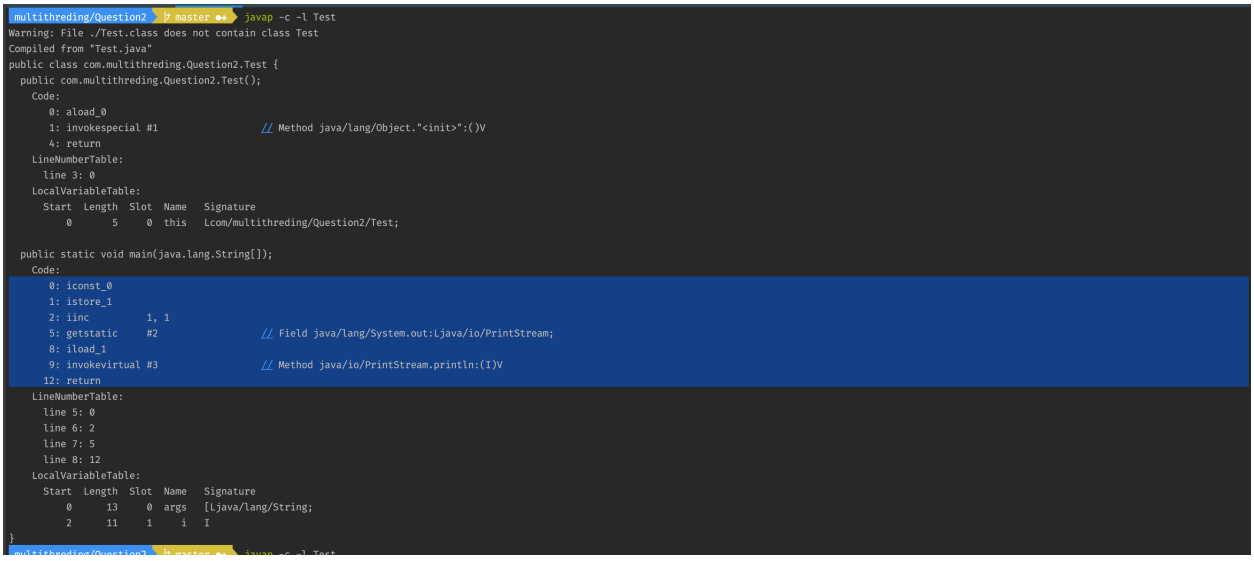
c) Is each code segment Atomic? Yes/No? Explain step-by-step

A)

```
public class Test {
```

```
    public static void main(String[] args) {
        int i=0;
        i++;
        System.out.println(i);
    }
}
```

Dissembled Code



Operations:

Iconst_0 → Push the int constant onto the operand stack

Istore_1 → Store the integer from LVA index 1 into local variable

iinc 1 , 1 → increment the local variable with constant here first “1” is index of LVA and second “1” is constant

getstatic → initialize reference class PrintStream

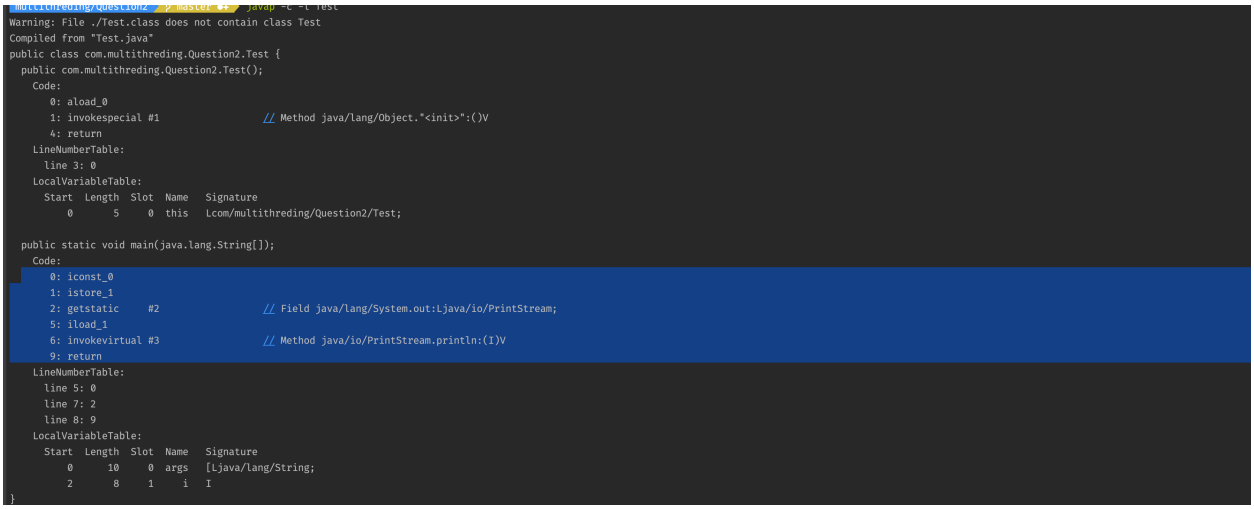
iload_1 →Load the integer value from LVA index 1 onto operand stack

invokevirtual → invoke println method

Current program is non-atomic as we can see in dissembled code that increment operation will first read the local variable value then increment the value and get the updated value using **iinc opcode.**
iinc operation is atomic in nature in terms of increment and save but in **multi- threaded environment read , increment and get operation can interleave over each other and give the incorrect result**.

B)

```
public class Test {

   public static void main(String[] args) {
      int i=1;
      System.out.println(i);
   }
}
```



```
Warning: File ./Test.class does not contain class Test
Compiled from "Test.java"
public class com.multithreding.Question2.Test {
  public com.multithreding.Question2.Test();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return
    LineNumberTable:
      line 3: 0
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0       5     0  this   Lcom/multithreding/Question2/Test;

  public static void main(java.lang.String[]);
    Code:
       0: iconst_0
       1: istore_1
       2: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
       5: iload_1
       6: invokevirtual #3                  // Method java/io/PrintStream.println:(I)V
       9: return
    LineNumberTable:
      line 5: 0
      line 7: 2
      line 8: 9
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0      10     0  args   [Ljava/lang/String;
          2       8     1     i   I
}
```

Operations:

Iconst_0 → Push the int constant onto the operand stack

Istore_1 → Store the integer from LVA index 1 into local variable

getstatic → initialize reference class PrintStream

iload_1 →Load the integer value from LVA index 1 onto operand stack

invokevirtual → invoke println method

The given program is atomic in nature as we are not modifying / updating the value of local variable. In multithreaded environment it will give same output.

3. Conversion of Number system:

a) 1011010010110101 convert to HEX —> convert to Decimal

Convert to hex

1011 = 1* 2^3+ 0*2^2 +1*2^1 +1*2^0 = 11= B

0100 = 0* 2^3+ 1*2^2 +0*2^1 +0*2^0 = 4= 4

1011 = 1* 2^3+ 0*2^2 +1*2^1 +1*2^0 = 11= B

0101 = 0* 2^3+ 1*2^2 +0*2^1 +1*2^0 = 5= 5

Hex = B4B5

HEX → Decimal

B4B5 = 11 * 16^3 + 4 * 16^2 + 11 * 16^1+ 5* 16^0

= 46261

b) 0xABCD Convert to Binary —> convert to Decimal

0x ABCD

A = 1010
B = 1011
C = 1100
D = 1101

Binary = 1010101111001101

Binary to Decimal
1010101111001101

1*2^15+0^2^14+1*2^13+0*2^12
+1*2^11+0^2^10+1*2^9+1*2^8+1*2^7+1*2^6+0*2^5+0*2^4+
1*2^2+1*2^2+0*2^1+1*2^0
=43981

c) 11010010110101 Convert to HEX. —> convert to Decimal

0011 =1*2^1+1*2^0 = 3
0100 =0*2^3 + 1*2^2 + +0*2^1+ +0*2^0 =4
1011 =1*2^3 + 0*2^2 + +1*2^1+ +1*2^0 =B
0101 =0*2^3 + 1*2^2 + +0*2^1+ +1*2^0 =5

Hex = 34B5

Hex -> Decimal

34B5= 3*16^3 +4* 16^2+11* 16^1+5*16^0
=13493

4. The architecture of your machine is 32-bits. Consider the following code and its
compiled disassembly:

```
uint64_t sharedValue = 0;
```

```
void storeValue() {
  sharedValue = 0x100000002;
}
```

```
$ gcc -O2 -S -masm=intel test.c
$ cat test.s
    ...
    mov  DWORD PTR sharedValue, 2
    mov  DWORD PTR sharedValue+4, 1
    ret
    ...
```

In given assembly code, the compiler implemented the 64-bit assignment using two separate machine instructions. The first instruction **sets the lower 32 bits to 0x00000002**, and the second sets the upper 32 bits **to 0x0000000**1. Clearly, **this assignment operation is not atomic**

sharedValue = 0x100000002 =**1**
00000000000000000000000000000010
00000000000000000000000000000010 – first 32 bits
00000000000000000000000000000001—second 32 bits

a) If sharedValue = 0x100000004; What changes in assembly code?

sharedValue = 0x100000004= **1**
00000000000000000000000000000100

00000000000000000000000000000100 – first 32 bit =4
00000000000000000000000000000001 – second 32 bits =1
So assembly code would be like

mov DWORD PTR sharedValue, 4
mov DWORD PTR sharedValue+4, 1

b) What are the root causes of Torn-read or Torn-write? Identify Thread
scenarios which causes torn-read or torn-write cases.

➔ If the reading and writing operations are not atomic then in concurrent reading and writing operation can lead the torn-read and torn write

➔ As we know that the assignment operation in current program is non atomic so if sharedValue is accessed by concurrently by different threads server things can go wrong

➔ If a thread calling **storeValue** is preempted between the two machine instructions, it will leave the value of 0x0000000000000002 in memory as **torn write**. At this point, if another thread reads sharedValue, it will receive this completely bogus value which nobody intended to store.

➔ Reading concurrently from sharedValue brings its own set of problems while reading the data concurrently compiler will implement the load operation using two machine instructions the first read the lower 32 bits into eax and the second reads the upper 32 bit into edx.

➔ In this case if a concurrent store to sharedValue become visible between the two instructions it will result to **torn read even if the concurrent store was atomic**

5. Consider the following two classes: SimulatedCAS and CAS Counter

```
public class SimulatedCAS {
    private int value;

    public synchronized int getValue() { return value; }

    public synchronized int compareAndSwap(int
expectedValue, int newValue) {
        int oldValue = value;
        if (value == expectedValue)
            value = newValue;
        return oldValue;
    }
}
```

```
public class CasCounter {
   private SimulatedCAS value;

   public int getValue() {
      return value.getValue();
   }

   public int increment() {
      int oldValue = value.getValue();
      while (value.compareAndSwap(oldValue, oldValue + 1) !=
oldValue)
         oldValue = value.getValue();
      return oldValue + 1;
   }
}
```

Explain each class

- ➔ Simulated CAS class has two synchronized method one will helps us to retrieve the latest value of value variable
- ➔ Second method is Compare and Swap method which compare the old value with expected value(expected old values) just to verify that no other thread has updated the value if so then function return the updated value only and doesn't perform any swap.

- ➔ If the old value is not changed then the function will swap the new value with old value and return the updated value.
- ➔ Compare and Swap is performing the atomic operation.
- ➔ CASCounter class implements a **thread safe counter using CAS**. The increment operation follows the canonical form fetch the old value, transform it to the new value (adding one), and use CAS to set the new value.
- ➔ If the CAS fails, the operation is immediately retried. CasCounter does not **block, though it may have to retry several times if other threads are updating the counter at the same time**

What are the differences between two classes?

- ➔ CAS Counter class use the simulated CAS class to perform the increment operation
- ➔ Both the class perform the atomic operation only but simulated CAS class has synchronized method to perform atomic operation.

Does one benefit over other? Explain details

- ➔ CAS Counter class use the simulatedCAS class to perform the automic increment operation so it definitely increase the performance and also avoid the unnecessary blocking issues.

d) Does one perform better than other? Why?

- ➔ CAS based counters significantly outperform lock based counters if there is even a small amount of contention, and often even if there is no contention.
- ➔ The fast path for uncontended lock acquisition typically requires at least one CAS plus other lock related housekeeping, so more work is going on in the best case for a lock based counter than in the normal case for the CASbased counter.
- ➔ Since the CAS succeeds most of the time (assuming low to moderate contention), the hardware will correctly predict the branch implicit in the while loop, minimizing the overhead of the more complicated control logic.
- ➔ Executing a CAS from within the program involves no JVM code, system calls, or scheduling activity. What looks like a longer code path at the application level is in fact a much shorter code path when JVM and OS activity are taken into account.

6. Consider the following code using AtomicLong

a) Compile code and Run

b) Rewrite the code as BlockingAlgorithm

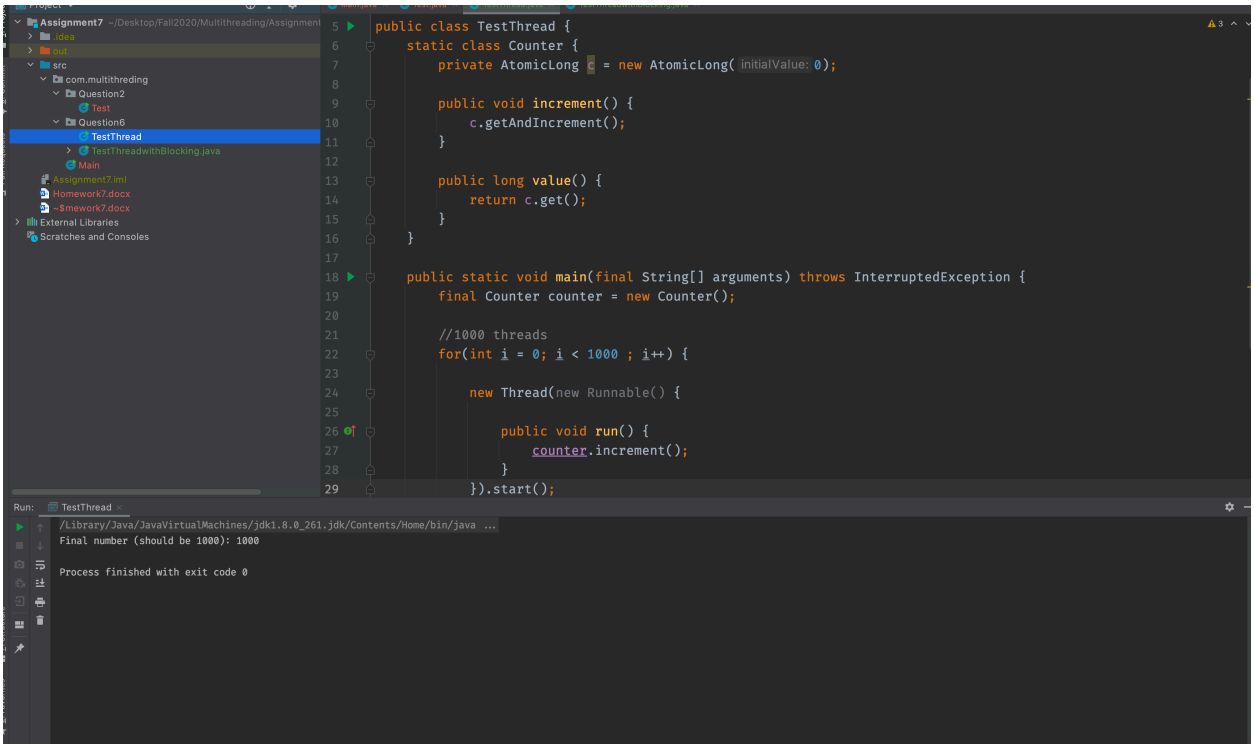
c) Compare performance (a) and (b)

```
import java.util.concurrent.atomic.AtomicLong;
```

```
public class TestThread {

  static class Counter {
    private AtomicLong c = new AtomicLong(0);

    public void increment() {
      c.getAndIncrement();
    }

    public long value() {
      return c.get();
    }
  }

  public static void main(final String[] arguments) throws
InterruptedException {
    final Counter counter = new Counter();

    //1000 threads
    for(int i = 0; i < 1000 ; i++) {

      new Thread(new Runnable() {

        public void run() {
          counter.increment();
        }
      }).start();
    }
    Thread.sleep(6000);
```

```
    System.out.println("Final number (should be 1000): " +
counter.value());
  }
}
```

a) Compile code and Run



b) Rewrite the code as BlockingAlgorithm

```
public class TestThreadwithBlocking {
    static class Counter {
        private long c = 0;

        public synchronized void increment() {
            c++;
        }

        public synchronized long value() {
            return c;
        }
    }
```

c) Compare performance (a) and (b)

In (a). we have used the non blocking algorithm to increment the value of variable using AtomicLong while in (b) we have used the synchronized methods to implement blocking

If we check the performance of (a) and (b) we find that the in **moderate contention non-blocking algorithm performs much better than blocking because the CAS succeeds** on first try and the penalty for contention when it does occur does not involve

thread suspension and context switching, just a few more iterations of the loop.

Under high contention when many threads are pounding on a single memory location lock-based **algorithms start to offer better throughput than nonblocking ones because when a thread blocks, it stops pounding and patiently waits its turn, avoiding further contention**.

No of Threads	Time in Blocking Algorithms in ms	Time in non blocking algorithm in ms
100000	3978	3877
1000000	35643	36119

7. Java does not support “AtomicDouble”. There are two code segments:

1) creates an instance of Double, and

2) uses AtomicReference to wrap “Double”. Compile each code segment and compare performance.

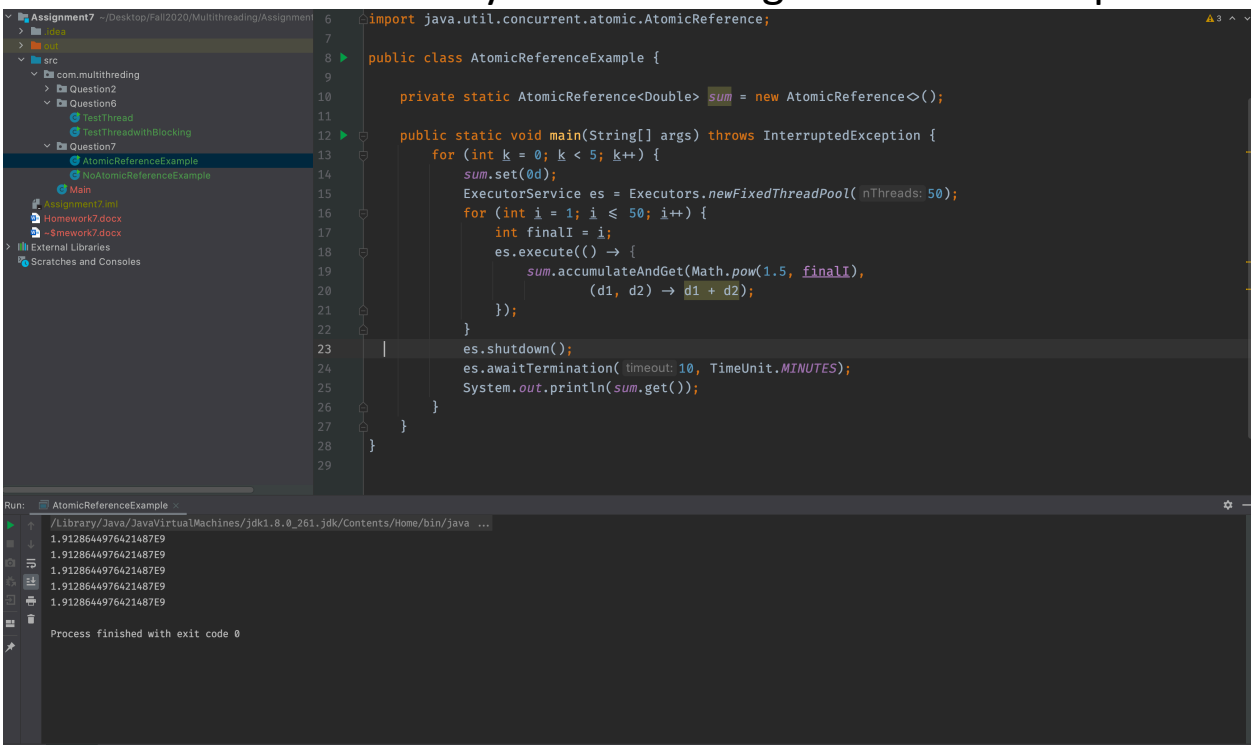
```
package com.logicbig.example;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class NoAtomicReferenceExample {
  private static Double sum;

  public static void main(String[] args) throws
InterruptedException {
     for (int k = 0; k < 5; k++) {
        sum=0d;
        ExecutorService es = Executors.newFixedThreadPool(50);
        for (int i = 1; i <= 50; i++) {
           int finalI = i;
           es.execute(() -> {
              sum+=Math.pow(1.5, finalI);
           });
        }
        es.shutdown();
        es.awaitTermination(10, TimeUnit.MINUTES);
        System.out.println(sum);
     }
```

```
  }
}
```

```
package com.logicbig.example;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicReference;

public class AtomicReferenceExample {
  private static AtomicReference<Double> sum = new
AtomicReference<>();
  public static void main(String[] args) throws
InterruptedException {
     for (int k = 0; k < 5; k++) {
        sum.set(0d);
        ExecutorService es = Executors.newFixedThreadPool(50);
        for (int i = 1; i <= 50; i++) {
           int finalI = i;
           es.execute(() -> {
              sum.accumulateAndGet(Math.pow(1.5, finalI),
                    (d1, d2) -> d1 + d2);
           });
        }
        es.shutdown();
        es.awaitTermination(10, TimeUnit.MINUTES);
        System.out.println(sum.get());
     }
```

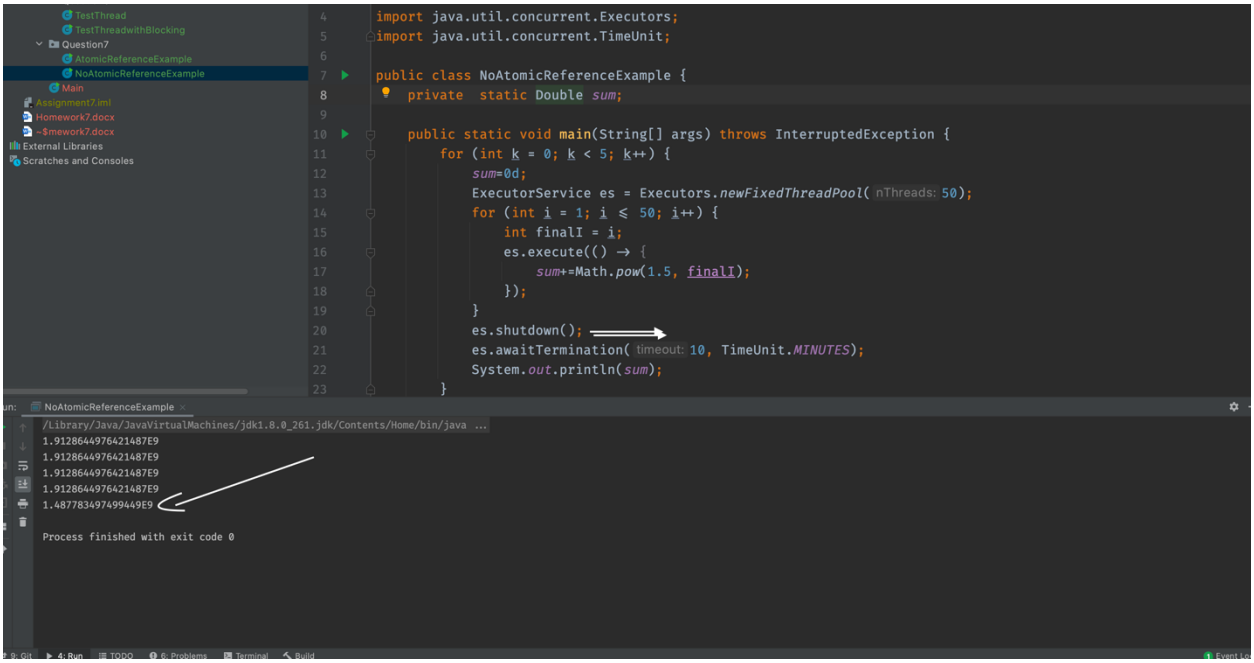
```
  }
}
```

Atomic Reference example:
In atomic reference every time we will get the same output.



Non Atomic reference example output :

In Non atomic result would not be same every time



➔ In NoAtomicReference class we get the different output if we run the program multiple times this thing happens because we are not protecting the shared

instance of Double , also we have not implemented any locking mechanism.

➔ Hence we can not compare the performance of NoAtomic reference class and Atomic Reference class because NoAtomic Reference class is not accurate.

➔ If we want to compare the performance of these two class then we need to implement any locking mechanism in NoAtomicReference class to get accurate output.

➔ If we implement the locking mechanism in NoAtomicReference class then in high contention level it will perform better then AtomicReference class, on the other hand Atomic reference class will perform very well in low and moderate contention level.