DESIGN

A. CONCURRENCY CONTROL

In our project, we heavily rely on the use of java concurrency utilities, namely cyclic barriers, priority blocking queues, and reentrant read-write locks, as this was clearly not ruled out in the instructions. While we didn't explicitly employ semaphores on their own, some of the utilities mentioned beforehand incorporate them, as well as other concurrency control mechanisms. In the following, we will explain how we used these tools to handle concurrency:

1) **CyclicBarrier (java.util.concurrent.CyclicBarrier)**
   A cyclic barrier is „A synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point. CyclicBarriers are useful in programs involving a fixed sized party of threads that must occasionally wait for each other. The barrier is called *cyclic* because it can be re-used after the waiting threads are released." (Oracle Docs).
   This barrier allows us to synchronize the looped execution of several BankStaff threads with each other and with the general Bank thread, which contains the global timer variable. Since we know the number of threads beforehand (M+1), we have a *fixed* sized party of threads. The barrier basically forces this party of threads into a linear order of execution, since they have to wait for each other at certain points during their execution. Within the CyclicBarrier source code itself, ReentrantLocks are used are used to handle concurrent access.
   Since our program contains several loops, it is particularly useful, that the barrier is *cyclic*, which is a contrast to, for example, a CountDownLatch. The following graphic gives a great overview how and where we used barriers to handle our threads.
   Insert graphic of concurrent BankStaff/Bank execution
2) **PriorityBlockingQueue (java.util.concurrent.PriorityBlockingQueue)**
   For job-handling, we rely on the use of the PriorityBlockingQueue class. All BanksStaff threads (Consumers) take jobs from these queues, while the Bank class (Producer) puts jobs into them. A PriorityBlockingQueue is „An unbounded blocking queue that uses the same ordering rules as class PriorityQueue and supplies blocking retrieval operations [...]" (Oracle Docs).
   At this point, the priority implementation is of no further interest to us, as it is only used later on in the bonus section, where we will go into more detail. For synchronization matters, it is import to understand the BlockingQueue interface. According to Oracle, the „BlockingQueue implementations are thread-safe. All queuing methods achieve their effects atomically using internal locks or other forms of concurrency control. [...] Note that a BlockingQueue can safely be used with multiple producers and multiple consumers." (Oracle Docs). This means, that the BlockingQueue basically works like the solution to the BoundedBufferProblem presented during lecture.
3) **ReentrantReadWriteLock (java.util.concurrent.locks.ReentrantReadWriteLock)**
   For synchronizing access to companies' bank accounts, we employ ReentrantReadWriteLocks. Every company instance is assigned with its own lock during construction. Threads that want to access a company's account, must first acquire either the respective ReadLock or WriteLock, dependent on their task. These locks work exactly as the solution to the ReadersAndWriters problem presented in class, and their functionality should be well-known, so no further discussion is needed.

B. OBJECTS

Our project consists of a total of eight .java classes. Below, we will explain each one's purpose and its main functions (getter, setter, etc. are not considered main functions).

* Insert graphic of how the classes are connected *

1) Main

Contains the well-known main(String[] args) function, starts the program

2) BankFrame

3) FileScanner

Is used to read input from the .txt configuration files. Every time when we iterate over the tokens in the .txt-files, a new Scanner object is created to reset the iterator. This is more economical than to reset the scanner (which is also why Oracle didn't even incorporate this function to begin with).

1) public FileScanner(int chosenFileNumber)

2) private void setup()
   Reads the configuration data (M, T_d, T_w, T_b, T_in, T_out) at the beginning of each .txt-file and saves it in a global array. Since all the provided config-samples had this form, it is assumed that every configuration file contains these details in exactly this order.
3) private void createEconomy()
   This is used to search the config-file for details on companies. Creates an instance of Company for every company specified in the configuration and stores it in a global array.
4) public PriorityBlockingQueue<Job> assignJobs(int timer)
   This is very frequently invoked by the Bank class. Searches the config-file for jobs where the assigned time equals the timer-parameter, and creates Job instances if there are any. All created instances are stored in a PriorityBlockingQueue which is then returned to the caller.
5) public boolean isDone(int timer)
   Is also invoked by the Bank class. Checks if all the entries in the config-file have been read. and returns a boolean variable depending on the result.

4) Bank

Is basically the most important class in the program. Contains and handles the global timer variable as well as the CyclicBarriers, both of which are used to synchronize the threads.

1) public Bank(FileScanner reader)
   Is used only once. Sets up all the variables in the class. The passed on reader instance is especially important, since the Bank class needs to have access to the config-file
2) public void doBusiness()
   If Bank is the main class of the program, then this is its main method. Creates all other threads, reads input from the config-file (acts as a producer), assigns jobs, handles the timer, and determines when the program is done. For better understanding of this method, refer to the graphic presented in the CyclicBarrier subsection of the concurrency control section.
3) public void awaitBarrier(int barrierNo)
   This is used to give everyone access to the CyclicBarriers. We created an extra method for this to make the source code more readable, since every await() instruction needs an extra try-multicatch surrounding it.

5) BankStaff

Implements the Runnable interface so it can run in parallel to the Bank thread. There are several instances of this, dependent on M.

1) public BankStaff(int i, Bank bank)
   Sets up an instance of BankStaff. i represents the tellerID, bank is used so this instance has a dynamic reference to Bank.
2) public void run()
   Most important method in this class. This is basically the corresponding code to Bank's doBusiness() method. Simulates how tellers handle jobs and customers. Same as in Bank doBusiness(), refer to the graphic provided earlier to get a better understanding of this method

6) Job

There is one instance of Job created for every job in the config-file. Every instance contains a lot of information which is set when the constructor is called. Implements the Comparable interface, which, in combination with PriorityBlockingQueue, is used for one of the bonus points.

1) public Job(...)
   Creates an instance of Job. Needs a lot of information and dynamic references in order to function properly.
2) public void setAdmitted(int admitted, BankStaff teller)
   When a teller starts to process a Job, admitted is set as the current timer and a dynamic reference to said teller is created.
3) public void execute(int timer)
   This is used to pass a task BankStaff to Employee for execution

7) Employee

Instances of Employee are created within FileScanner every time a customer comes to the Bank.

1) public Employee(int ID, Company employer)
   The constructor is called from within the FileScanner class and assigns every employee his own ID. Furthermore, a dynamic link to an instance of company is created.
2) public void doTask(...)
   This is called from within Job and signals that the employee can now execute his task. The function is overloaded with a lot of parameters that contain information about the Job, as well as a dynamic link to the teller that supervises the action. Calls the action which is defined in transactionType.
3) private void withdraw(...)
   Is called from doTask and executes a withdraw command. For this, a WriteLock from the associated company is acquired, then the method forces the thread to idle for a while to simulate the transaction time. Once the transaction is done, the lock is released.
4) private void deposit(...)
   Is called from doTask and executes a deposit command. For this, a WriteLock from the associated company is acquired, then the method forces the thread to idle for a while to simulate the transaction time. Once the transaction is done, the lock is released.
5) private void checkBalance(...)
   Is called from doTask and executes a checkBalance command. For this, a ReadLock from the associated company is acquired, then the method forces the thread to idle for a while to simulate the transaction time. Once the transaction is done, the lock is released.

8) Company

Instances of Company are created at the beginning of the program within the FileScanner class. Each company has a balance, a distinct number and a ReentrantReadWriteLock which is used to protect access to its account (balance).

1) public Company(int companyNo, int balance)
   The constructor assigns values to the global variables and creates a lock object.

C. PSEUDO CODE

D. BONUS POINTS

1) **You can develop a reasonable method so that all the customers get served in the shortest amount of time;**
   Our program contains two methods which try to assure that every customer gets served in the shortest time possible. These two methods are the ReentrantReadWriteLock and the PriorityBlockingQueue.
   The ReentrantReadWriteLock gives threads, which only wish to check the balance of a company, the ability to access the company's account concurrently, which speeds up the program in general.
   The PriorityBlockingQueue imposes a FCFS rule on employees arriving at the bank. We chose FCFS because we wanted to achieve a minimal average turnaround time, a goal which a real world bank would probably pursue as well. SJF would have been a possibility as well, but since we did not have an easy way to implement aging, starvation was a serious problem when applying this algorithm.
2) **The system comes with a nice interface to illustrate the process;**
3) **You implement creative/innovative way of applying synchronization.**
   We achieve synchronization mainly through the use of CyclicBarriers. We figured that, within the scope of the lecture, this qualifies as an innovative way of achieving synchronization, since we did not learn about this before. We only discovered this utility while exploring the java concurrency utensils, so it was new to us.