/\*  
 1. Name / Date: satya Yoganand Addala / 26-10-2022  
  
 2. Java version used (java -version), if not the official version for the class: 18.0.2  
  
 3. Precise command-line compilation examples / instructions:  
  
 > javac BlockChain.java  
  
 4. Files Used for this process :  
  
 > BlockInput0.txt  
 > BlockInput1.txt  
 > BlockInput2.txt  
  
 5. Precise examples / instructions to run this program:  
  
 In separate shell windows run all the below commands :  
  
 > java -cp ".:gson-2.8.2.jar" Blockchain.java 0  
 > java -cp ".:gson-2.8.2.jar" Blockchain.java 1  
 > java -cp ".:gson-2.8.2.jar" Blockchain.java 2  
  
  
 All acceptable commands are displayed on the various consoles.  
  
 6. External Libraries Used : GSON Jar- gson-2.8.2.jar  
  
 7. Referenced Programs from Clark Elliott :  
  
 > Blockchain utilities sample program Version J  
 > Blockchain input utilty program,  
 > Sample Work Program  
 > Process Coordination  
  
 8. Notes:  
  
 \* The BlockChain.java file holds various classes and methods that helps to execute the BlockChain  
 \* This file runs three processes in parallel where are currently i am using three input files namely BlockInput0.txt,BlockInput1.txt,BlockInput2.txt.  
 \* Now the process ID will take by the process from commandline and sets up all the required ports with respect to processID.  
 \* Then all the process waits for process 2 to run in which once the process2 Starts running it will multicast a start message to all the running processes.  
 \* Then the other process starts to fetch all the three processes public keys and it will try to read there Input files based on their processID's.  
 \* Once it has been done then all the records present in the files are converted into unverified blocks and are added to the priority Queue for further processing.  
 \* All the Unverified blocks will be shared among all the three processes.  
 \* Then the Work will come into play which induces a simple puzzle to solve and helps to verify the unverified blocks and add them into a blockchain.  
 \* Now at the end of the process user will be given an opportunity to analyze the statistics of the processes by giving an user Input which inturn will give the processes records and credits for all the three processes.  
  
  
 9. Extra Notes :  
  
 The sleep time has been incremented to 10000 milliseconds because when trying to start the second process the first process is not able to pick the pub,ic key which has been settled using increased sleep time  
  
 10.Referenced Web Sources :  
  
 https://mkyong.com/java/how-to-parse-json-with-gson/  
 http://www.java2s.com/Code/Java/Security/SignatureSignAndVerify.htm  
 https://www.mkyong.com/java/java-digital-signatures-example/  
 https://javadigest.wordpress.com/2012/08/26/rsa-encryption-example/  
 https://www.programcreek.com/java-api-examples/index.php?api=java.security.SecureRandom  
 https://www.mkyong.com/java/java-sha-hashing-example/  
 https://stackoverflow.com/questions/19818550/java-retrieve-the-actual-value-of-the-public-key-from-the-keypair-object  
 https://www.java67.com/2014/10/how-to-pad-numbers-with-leading-zeroes-in-Java-example.html  
 https://www.javacodegeeks.com/2013/07/java-priority-queue-priorityqueue-example.html  
  
  
\*/  
  
import com.google.gson.Gson;  
import com.google.gson.GsonBuilder;  
  
import java.io.\*;  
import java.net.ServerSocket;  
import java.net.Socket;  
import java.security.\*;  
import java.security.spec.X509EncodedKeySpec;  
import java.util.\*;  
  
  
public class BlockChain {  
  
 /\* ServerName Set to Localhost \*/  
 static String *serverName* = "localhost";  
  
 /\* Total Processes that are being used \*/  
 static int *numProcesses* = 3; // needs to be 3 total processes  
  
 /\* Initialize Process ID \*/  
 static int *PID* = 0;  
  
 /\* The Unverified block array relative to this process \*/  
 static List<BlockRecord> *blockArr* = new ArrayList<>();  
  
 /\* Variable to read the file name that is picked with respect to pid \*/  
 private static String *fileName*;  
  
 /\* The Global String used for generating random Seed \*/  
 private static final String *ALPHA\_NUMERIC\_STRING* = "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";  
  
 /\* LinkedList to hold all the blocks \*/  
 public static LinkedList<BlockRecord> *blockChain* = new LinkedList<>();  
  
 /\* Array to hold all the publicKeys with the list Object as PublicKeyObject where the public key and processID has been stored \*/  
 public static List<PublicKeyObject> *publicKeyArray* = new ArrayList<>();  
  
 /\* Variable to store private key \*/  
 public static PrivateKey *privateKey*;  
  
 /\* Start variable to begin the process and is set to default as wait \*/  
 public static String *start* = "wait";  
  
 /\* Indexes for getting the file contents and are listed accordingly \*/  
 private static final int *fName\_Index* = 0;  
 private static final int *lName\_Index* = 1;  
 private static final int *dOB\_Index* = 2;  
 private static final int *iSSNUM\_Index* = 3;  
 private static final int *iDIAG\_Index* = 4;  
 private static final int *treatment\_Index* = 5;  
 private static final int *iRX* = 6;  
  
 /\* Creating a publicKeyObject of the type PublicKeyObject to use it in the process \*/  
 public static PublicKeyObject *publicKeyObj* = new PublicKeyObject();  
  
 public static Comparator<BlockRecord> *blockTSComparator* = new Comparator<BlockRecord>()  
 {  
 static Queue<BlockRecord> *blockPriorityQueue* = new PriorityQueue<>(4, *blockTSComparator*);  
  
 /\* Overriding the default compare method to do the compare task with respect to timestamps \*/  
 @Override  
 public int compare(BlockRecord r1, BlockRecord r2) {  
 /\* Fetching the time stamps for each block to compare against time \*/  
 String t1 = r1.getTimeStamp();  
 String t2 = r2.getTimeStamp();  
 /\* If both time stamps are equal then return 0 \*/  
 if (t1 == t2) {return 0;}  
 /\*If either t1 or t2 are null then return -1 0r 1 respectively \*/  
 if (t1 == null) {return -1;}  
 if (t2 == null) {return 1;}  
 /\* return the standard compare method with respect to the timestamps \*/  
 return t1.compareTo(t2);  
 }  
 };  
  
 static Queue<BlockRecord> *blockPriorityQueue* = new PriorityQueue<>(4, *blockTSComparator*);  
  
 public static String hashBlock(String blockContents) {  
 /\* Setting up the hash Algorithm to SHA-256 \*/  
 String hashAlgorithm = "SHA-256";  
 String SHA256String = "";  
  
 try {  
 /\* Message Digest Class has been used to get an instance of hash Algorithm \*/  
 MessageDigest md = MessageDigest.*getInstance*(hashAlgorithm);  
 md.update(blockContents.getBytes());  
 /\* Converting the contents to bytes \*/  
 byte byteData[] = md.digest();  
 /\* Converting the hashed bytes to hex \*/  
 /\* Initializing a new String Buffer Class \*/  
 StringBuffer sb = new StringBuffer();  
 /\* Looping on the hash bytes \*/  
 for (int i = 0; i < byteData.length; i++) {  
 sb.append(Integer.*toString*((byteData[i] & 0xff) + 0x100, 16).substring(1));  
 }  
 /\* Converting the obtained hex value to String \*/  
 SHA256String = sb.toString();  
  
 } catch (NoSuchAlgorithmException x) {  
 /\* Printing Stack Trace \*/  
 x.printStackTrace();  
 }  
 ;  
  
 /\* Returning the Converted String \*/  
 return SHA256String.toUpperCase();  
 }  
  
 public static LinkedList<BlockRecord> blockChainInit(){  
 LinkedList<BlockRecord> blockRecord = new LinkedList<>();  
 BlockRecord block0 = new BlockRecord();  
 /\* Generating the UniqueId with random UUID method and converting it to string \*/  
 String unique\_ID = UUID.*randomUUID*().toString();  
 /\* Setting the blockID for block0 with uniqueId that has been generated \*/  
 block0.setBlockID(unique\_ID);  
 /\* Setting the block number to initial block with is 0 \*/  
 block0.setBlockNum(0);  
 /\* Making the thread to sleep for 1001 milliseconds \*/  
 try{  
 Thread.*sleep*(1001);  
 }  
 catch(InterruptedException e){  
 /\* Printing Stack Trace \*/  
 e.getStackTrace();  
 }  
 /\* Initializing the date Variable \*/  
 Date date = new Date();  
 /\* Generating the time stamp with respect to the date variable, and it generates something like year-month-date.hour:minutes:seconds \*/  
 String timeStamp = String.*format*("%1$s %2$tF.%2$tT", "", date);  
 /\* Concat the process ID with the generated Timestamp\*/  
 String timeStampString = timeStamp + "." + BlockChain.*PID*;  
 /\* Setting the TimeStamp to block0 \*/  
 block0.setTimeStamp(timeStampString);  
 /\* Setting the verification ProcessID to 0\*/  
 block0.setVerificationProcessID("0");  
 /\* Setting the previous hash to a random value \*/  
 block0.setPreviousHash("0000000000000000000000000000000000000000000000000000000000000000");  
 /\* Setting the FirstName,LastName and DateOfBirth to match my details \*/  
 block0.setFname("Satya Yoganand");  
 block0.setLname("Addala");  
 block0.setDOB("05-22-1998");  
 /\* Setting some dummy SSN Number \*/  
 block0.setSSNum("123-45-6789");  
 /\* Setting the diag to Blockchain Initial Block \*/  
 block0.setDiag("Blockchain Initial Block");  
 /\* Setting RX to Blockchain pills \*/  
 block0.setRx("Blockchain pills");  
 /\* Setting Treat to Writing more blockChain code \*/  
 block0.setTreat("Writing more block chain code");  
 /\* Setting some random seed \*/  
 block0.setRandomSeed("08L920RE");  
 /\* Generating the block data with the help of random seed \*/  
 String blockData = *concat*(block0) + "08L920RE";  
 /\* Setting the Winning hash with the block data \*/  
 block0.setWinningHash(*hashBlock*(blockData));  
 /\* Adding this block0 to blockRecord \*/  
 blockRecord.add(block0);  
 return blockRecord;  
 }  
  
 /\* Method for generating public keys for each process \*/  
 public static PublicKeyObject publicKeyInit(int pid) throws Exception {  
  
 /\* Initializing the random variable with the help of Random method \*/  
 Random random = new Random();  
 long randomNum = random.nextInt(1000);  
 /\* Generating the Key Value Pair with the help of randomNum \*/  
 KeyPair keyPair = *generateKeyPair*(randomNum);  
 /\* Fetching the private key through the generated KeyPair \*/  
 *privateKey* = keyPair.getPrivate();  
 System.*out*.println("Key Pair : "+ keyPair);  
 /\* Fetch the public key from the key value pair,and store it in the bytePublicKey variable of the type byte array \*/  
 byte[] bytePublicKey = keyPair.getPublic().getEncoded();  
 /\* By using base64 encoding convert the public key which is in byte array to a string \*/  
 String stringKey = Base64.*getEncoder*().encodeToString(bytePublicKey);  
 /\* Set the converted string to the public key variable in publicKeyObj Class \*/  
 *publicKeyObj*.setPublicKey(stringKey);  
 /\* Set the ProcessId of the process to ProcessId variable in publicKeyObj Class \*/  
 *publicKeyObj*.setProcessID(pid);  
 System.*out*.println("Printing Public Key : "+*publicKeyObj*);  
 /\* Return the public key Object \*/  
 return *publicKeyObj*;  
 }  
  
 public static boolean verifySig(byte[] data, PublicKey key, byte[] sig) throws Exception {  
 /\* Getting an Instance of Signature Class and using SHA1withRSA Algorithm \*/  
 Signature signer = Signature.*getInstance*("SHA1withRSA");  
 /\* Initializing the data to get signed with the help of public key \*/  
 signer.initVerify(key);  
 signer.update(data);  
 /\* Returning the verified signature \*/  
 return (signer.verify(sig));  
 }  
  
 public static KeyPair generateKeyPair(long seed) throws Exception {  
 /\* Fetching a new Instance of KeyPairGenerator and using RSA Algorithm \*/  
 KeyPairGenerator keyGenerator = KeyPairGenerator.*getInstance*("RSA");  
 SecureRandom rng = SecureRandom.*getInstance*("SHA1PRNG", "SUN");  
 /\* Setting the random seed \*/  
 rng.setSeed(seed);  
 keyGenerator.initialize(1024, rng);  
 /\* Returning the generated Key Pair \*/  
 return (keyGenerator.generateKeyPair());  
 }  
 /\* Method for Signing the winning hash \*/  
 public static byte[] signData(byte[] data, PrivateKey key) throws Exception {  
 /\* Initializing the Signature Instance of SHA1withRSA Algorithm \*/  
 Signature signer = Signature.*getInstance*("SHA1withRSA");  
 /\* Initializing the signer object for signing and the private key is used to identity whose signature is going to be generated.\*/  
 signer.initSign(key);  
 /\* This updates the data to be signed \*/  
 signer.update(data);  
 return (signer.sign());  
 }  
  
 /\* Method for getting randomSeed based on the count variable \*/  
 public static String randomAlphaNumeric(int count) {  
 /\* Initializing String Builder to new for getting different random seeds for different processes \*/  
 StringBuilder builder = new StringBuilder();  
 /\* Running loop based on count so that the random seed has the same number of characters \*/  
 while (count-- != 0) {  
 /\* fetching single character from the Alpha numeric string \*/  
 int character = (int)(Math.*random*()\**ALPHA\_NUMERIC\_STRING*.length());  
 /\* Appending each character to the builder String \*/  
 builder.append(*ALPHA\_NUMERIC\_STRING*.charAt(character));  
 }  
 /\* Returning the random Seed \*/  
 return builder.toString();  
 }  
  
 /\* Method for Concatenating the block Record values into a String \*/  
 public static String concat(BlockRecord blockRecord){  
 /\* Concat the BlockRecord values together and appending it to a string \*/  
  
 /\* Returning the formed String \*/  
 return blockRecord.getTimeStamp() +  
 blockRecord.getBlockNum() +  
 blockRecord.getBlockID() +  
 blockRecord.getSignedID() +  
 blockRecord.getPreviousHash() +  
 blockRecord.getFname() +  
 blockRecord.getLname() +  
 blockRecord.getDOB() +  
 blockRecord.getSSNum() +  
 blockRecord.getVerificationProcessID() +  
 blockRecord.getDiag() +  
 blockRecord.getTreat() +  
 blockRecord.getRx() +  
 blockRecord.getTimeStamp();  
 }  
  
 public static BlockRecord Work(BlockRecord blockRecord){  
 /\* Initializing the randomSeed to null \*/  
 String randomSeed = "";  
 /\* Initializing dataSeed to null \*/  
 String dataSeed = "";  
 /\* Initializing hash to null \*/  
 String hash = "";  
 /\* Initializing workNumber to 0 \*/  
 int workNumber = 0;  
 /\* Setting the previous Hash with that of the Winning hash of the previous block \*/  
 blockRecord.setPreviousHash(BlockChain.*blockChain*.get(0).getWinningHash());  
 /\* Setting the block number such that it will always be one number higher than that of the previous block number \*/  
 blockRecord.setBlockNum(BlockChain.*blockChain*.get(0).getBlockNum() + 1);  
 /\* Setting the Verification ProcessID to the current processID \*/  
 blockRecord.setVerificationProcessID(Integer.*toString*(BlockChain.*PID*));  
 /\* Concatenating all the above values using concat method and stored it in a string \*/  
 String blockData = *concat*(blockRecord);  
 try {  
 while (true) {  
 /\* Calling the randomAlphaNumeric Method and setting count to 8 to generate a random seed \*/  
 randomSeed = *randomAlphaNumeric*(8);  
 /\* Combining the BlockData with that of random seed to generate dataSeed \*/  
 dataSeed = blockData + randomSeed;  
 /\* Initializing the SHA-256 Algorithm Instance \*/  
 MessageDigest md = MessageDigest.*getInstance*("SHA-256");  
 /\* Hashing the dataSeed into bytes and storing ot on a variable \*/  
 byte[] bytesHash = md.digest(dataSeed.getBytes("UTF-8"));  
 /\* Converting the bytes to hex using stringBuild \*/  
 StringBuilder stringBuilder = new StringBuilder();  
 /\* Looping through the bytes \*/  
 for ( byte b : bytesHash) {  
 /\* Appending each byte by converting it using String format to the stringBuilder \*/  
 stringBuilder.append(String.*format*("%02x", b));  
 }  
 hash = stringBuilder.toString();  
 System.*out*.println("Hash is: " + hash);  
 /\* Fetching the first four hex digits and convert that to decimal. Here you could make the work harder by using more of the hex digits, so that the range of possible numbers is far larger and the chance of solving the puzzle becomes much smaller and easier \*/  
 workNumber = Integer.*parseInt*(hash.substring(0,4),16);  
 System.*out*.println("First 16 bits in Hex and Decimal: " + hash.substring(0,4) +" and " + workNumber);  
 /\* Checking for the puzzle condition if the work number is not less than 20000 then the puzzle is not solved \*/  
 if (!(workNumber < 20000)){  
 System.*out*.format("%d is not less than 20,000 so we did not solve the puzzle\n\n", workNumber);  
 }  
 /\* Checking for the puzzle condition that if the worknumber is less than 200000 then the puzzle is solved and the block gets verified. \*/  
 if (workNumber < 20000){  
 System.*out*.println("block verified!");  
 /\* Setting the random seed to the verified block \*/  
 blockRecord.setRandomSeed(randomSeed);  
 /\* Setting the winning hash for the verified block \*/  
 blockRecord.setWinningHash(hash);  
 /\* Signing the Winning hash with the help of private key \*/  
 byte[] signedWinHash = *signData*(bytesHash, *privateKey*);  
 /\* Converting the signed winning hash to string using base64 encoder \*/  
 String signedWinHashStr = Base64.*getEncoder*().encodeToString(signedWinHash);  
 /\* Setting the SignedWinningHash for the current verified block \*/  
 blockRecord.setSignedWinningHash(signedWinHashStr);  
 break;  
 }  
 /\* Looping through blockchain to verify whether any unverified blocks are added to blockchain by any other process \*/  
 for (BlockRecord b: *blockChain*){  
 /\* Condition to check for any matched BlockID's \*/  
 if (b.getBlockID().equals(blockRecord.getBlockID())){  
 /\* If there is any match then the block should be abandoned \*/  
 System.*out*.println("Abandoning block...");  
 /\* Creating a new block record for abandoned Block \*/  
 BlockRecord abandonedBlock = new BlockRecord();  
 /\* Setting the BlockID of that block to Abandoned \*/  
 abandonedBlock.setBlockID("Abandoned");  
 /\* Returning the abandoned block \*/  
 return abandonedBlock;  
 }  
 }  
 /\* Sleeping to simulate more work (fake work) \*/  
 try{  
 Thread.*sleep*(7001);  
 }catch(InterruptedException e){}  
 }  
 }catch(Exception ex) {  
 /\* Printing Stack Trace \*/  
 ex.printStackTrace();  
 }  
 /\* Return a verified block \*/  
 return blockRecord;  
 }  
  
 /\* Method for multicasting publicKeys over different processes \*/  
 public void multiCastPublicKey(PublicKeyObject publicKey) {  
 Socket sock;  
 PrintStream toServer;  
 /\* Initializing the gson variable with the help of gson library \*/  
 Gson gson = new GsonBuilder().create();  
  
 /\* Fetch the public key and convert it to Json Object \*/  
 String JSON = gson.toJson(publicKey);  
 try{  
  
 /\* Looping all the processes \*/  
 for(int i=0; i<*numProcesses*; i++){  
 /\* Creating a new connection to each port here we are using the port number which is spaced out based on the processID \*/  
 System.*out*.println("multicast public key Port : "+(Ports.*publicKeyServerBase* + (i)));  
 sock = new Socket(*serverName*, (Ports.*publicKeyServerBase* + (i)));  
 /\* Setting the output Stream to the receiving server \*/  
 toServer = new PrintStream(sock.getOutputStream());  
 /\* Sending the public key with that of processID in JSON format \*/  
 toServer.println(JSON);  
 toServer.flush();  
 }  
 }catch (Exception x) {  
 /\* Printing Stack Trace \*/  
 x.printStackTrace ();  
 }  
 }  
  
 /\* Method for multicasting start message to all the running processes \*/  
 public void multiCastStart(){  
 Socket sock;  
 PrintStream toServer;  
 try{  
 /\* Looping for all the processes \*/  
 for(int i=0; i<*numProcesses*; i++){  
 /\* Creating a new connection to each port here we are using the port number which is spaced out based on the processID \*/  
 sock = new Socket(*serverName*, (Ports.*StartServerBase* + (i)));  
 /\* Setting the Output Print Stream \*/  
 toServer = new PrintStream(sock.getOutputStream());  
 /\* Sending the Start Message to all the other processes \*/  
 toServer.println("go");  
 toServer.flush();  
 }  
 }catch (Exception x) {  
 /\* Printing Stack Trace \*/  
 x.printStackTrace ();  
 }  
 }  
  
 /\* Method for multicasting unverified blocks to all the processes \*/  
 public void multiCastUB(BlockRecord blockRecord, int serverBase){  
 Socket sock;  
 PrintStream toServer;  
 try{  
 /\* Initializing new gson variable \*/  
 Gson gson = new GsonBuilder().create();  
 /\* Converting the BlockRecord Object to JSON Object \*/  
 String jSon = gson.toJson(blockRecord);  
 /\* Looping for all the processes \*/  
 for(int i=0; i< *numProcesses*; i++){  
 /\* Creating a new Socket connection with the port number which is spaced out based on ProcessID \*/  
 sock = new Socket(*serverName*, (serverBase + (i)));  
 /\* Setting the Output Print Stream \*/  
 toServer = new PrintStream(sock.getOutputStream());  
 /\* Sending the JSON Object along with the processID over all the processes \*/  
 toServer.println(jSon);  
 toServer.flush();  
 }  
 }catch (Exception x) {  
 /\* Printing Stack Trace \*/  
 x.printStackTrace ();  
 }  
 }  
  
 /\* Method for Multicasting BlockChain to all the other processes \*/  
 public void multiCastBC(LinkedList<BlockRecord>bc, int serverBase){  
 Socket sock;  
 PrintStream toServer;  
  
 try{  
 Gson gson = new GsonBuilder().create();  
 /\* Converting the blockchain record into a JSon using the method blockchain\_JSon\_Converter \*/  
 String jSon = *blockChain\_JSon\_Converter*(bc);  
 /\* Loop for all the running processes \*/  
 for(int i=0; i< *numProcesses*; i++){  
 /\* Creating a new Socket connection with the port which is based on the processID \*/  
 sock = new Socket(*serverName*, (serverBase + (i)));  
 /\* Setting up an Output PrintStream \*/  
 toServer = new PrintStream(sock.getOutputStream());  
 /\* Send the JSON to all the other processes \*/  
 toServer.println(jSon);  
 toServer.flush();  
 }  
  
 }catch (Exception x) {  
 /\* Printing Stack Trace \*/  
 x.printStackTrace ();  
 }  
 }  
  
 /\* Method for reading an INPUT File and Converting it to block record \*/  
 public static List<BlockRecord> readFile(int pid){  
 /\* Record list of the type BlockRecord to contain the records in the file \*/  
 List<BlockRecord> recordList = new ArrayList<>();  
  
 String[] tokens;  
 String inputLineStr;  
 String unique\_ID;  
  
 /\* Switch Condition to pick the appropriate file based on ProcessID \*/  
 switch(pid){  
 /\* Read BlockInput1 for processID 1 \*/  
 case 1: *fileName* = "BlockInput1.txt";  
 break;  
 /\* Read BlockInput2 for processID 2 \*/  
 case 2: *fileName* = "BlockInput2.txt";  
 break;  
 /\* Read BlockInput3 for processID 0 \*/  
 default: *fileName*= "BlockInput0.txt";  
 break;  
 }  
  
 System.*out*.println("Using input file: " + *fileName*);  
 /\* Buffered Reader to go through the contents of the file \*/  
 try {  
 BufferedReader br = new BufferedReader(new FileReader(*fileName*));  
  
 /\* Variable to keep track of the records present in a file \*/  
 int recordCount = 0;  
  
 while ((inputLineStr = br.readLine()) != null) {  
 /\* Creating a new BlockRecord for each line in the file \*/  
 BlockRecord blockRecord = new BlockRecord();  
 /\* Initializing a thread sleep so that it will give some time to read the data of each line \*/  
 try{  
 Thread.*sleep*(1001);  
 }catch(InterruptedException e){  
 /\* Printing Stack Trace \*/  
 e.printStackTrace();  
 }  
 /\* Creating a new date variable to generate the timestamp \*/  
 Date date = new Date();  
 /\* Generating the TimeStamp using String format \*/  
 String T1 = String.*format*("%1$s %2$tF.%2$tT", "", date);  
 /\* Adding the processID to the generated TimeStamp \*/  
 String TimeStampString = T1 + "." + pid;  
 /\* Setting the timeStamp for the blockRecord \*/  
 blockRecord.setTimeStamp(TimeStampString);  
 /\* Setting the recordCount to the BlockNumber \*/  
 blockRecord.setBlockNum(recordCount);  
  
 /\* Generating a UniqueID using randomUUID call \*/  
 unique\_ID = new String(UUID.*randomUUID*().toString());  
 /\* Setting the UniqueID to BlockID \*/  
 blockRecord.setBlockID(unique\_ID);  
 /\* Signing the UniqueID with the privateKey \*/  
 byte[] digitalSignature = *signData*(unique\_ID.getBytes(), *privateKey*);  
 /\* Converting the Signed Variable to String using Base64 Encoding \*/  
 String SignedSHA256ID = Base64.*getEncoder*().encodeToString(digitalSignature);  
 /\* Setting the Signed ID \*/  
 blockRecord.setSignedID(SignedSHA256ID);  
 /\* Splitting the contents of the line and setting up the data \*/  
 tokens = inputLineStr.split(" +");  
 /\* Setting the firstname \*/  
 blockRecord.setFname(tokens[*fName\_Index*]);  
 /\* Setting the lastname \*/  
 blockRecord.setLname(tokens[*lName\_Index*]);  
 /\* Setting SSN Number \*/  
 blockRecord.setSSNum(tokens[*iSSNUM\_Index*]);  
 /\* Setting dateOfBirth \*/  
 blockRecord.setDOB(tokens[*dOB\_Index*]);  
 /\* Setting Diag \*/  
 blockRecord.setDiag(tokens[*iDIAG\_Index*]);  
 /\* Setting Treat \*/  
 blockRecord.setTreat(tokens[*treatment\_Index*]);  
 /\* Setting Rx \*/  
 blockRecord.setRx(tokens[*iRX*]);  
 /\* Setting Verification process ID \*/  
 blockRecord.setVerificationProcessID(Integer.*toString*(BlockChain.*PID*));  
 /\* Add the block record to the recordList \*/  
 recordList.add(blockRecord);  
 /\* Incrementing the record Count \*/  
 recordCount++;  
 }  
 } catch (Exception e){  
 /\* Printing Stack Trace \*/  
 System.*out*.println(e);  
 }  
 /\* Return the unverified blocks \*/  
 return recordList;  
 }  
  
 /\* Method for write the verified blocks to a file \*/  
 public static void writeToFile(LinkedList<BlockRecord> bc){  
 /\* For printing the Json in a readable format we use gson pretty \*/  
 Gson gsonPretty = new GsonBuilder().setPrettyPrinting().create();  
 /\* The JSON Write variable initialized to open square bracket as the Json starts with this \*/  
 String JSONWrite = "[";  
 /\* Looping each block in block Chain \*/  
 for (BlockRecord block: bc){  
 JSONWrite += gsonPretty.toJson(block);  
 /\* Adding a comma separator between each block \*/  
 if (bc.indexOf(block) != bc.size() - 1)  
 JSONWrite += ",";  
 }  
 /\* Setting the end bracket at the end of each block \*/  
 JSONWrite = JSONWrite + "]";  
 /\* Writing all the data to a file named BlockChainLedger \*/  
 try (FileWriter writer = new FileWriter("BlockchainLedger.json", false)) {  
 writer.write(JSONWrite);  
 } catch (IOException e) {  
 /\* Printing Stack Trace \*/  
 e.printStackTrace();  
 }  
 }  
  
 /\* Method for converting BlockRecord object to a json Object \*/  
 public static String blockChain\_JSon\_Converter(LinkedList<BlockRecord> bc){  
 /\* Created a new GsonBuilder Instance \*/  
 Gson gson = new GsonBuilder().create();  
 /\* Initializing the Json Object to open Square bracket \*/  
 String jSON = "[";  
 /\* looping through block record \*/  
 for (BlockRecord blockRecord: bc){  
 jSON += gson.toJson(blockRecord);  
  
 /\* Adding commas between the values \*/  
 if (bc.indexOf(blockRecord) != bc.size() - 1)  
 jSON += ",";  
 }  
 /\* Closing the Json with a closed Square Bracket \*/  
 jSON = jSON + "]";  
 /\* Return the converted Json \*/  
 return jSON;  
 }  
  
 /\* Method to print the Blocks \*/  
 public static void printBlock(BlockRecord block){  
 String printRecord = "";  
 printRecord = block.getBlockNum() + ". "  
 + block.getTimeStamp() + " "  
 + block.getFname() + " "  
 + block.getLname() + " "  
 + block.getSSNum() + " "  
 + block.getDOB() + " "  
 + block.getDiag() + " "  
 + block.getTreat() + " "  
 + block.getRx() + " ";  
 System.*out*.println(printRecord);  
 }  
  
 /\* Method for printing list credits for each processes \*/  
 public static void listCredit(){  
 /\* Initializing credit for processID 0 \*/  
 int credit0 = 0;  
 /\* Initializing credit for processID 1 \*/  
 int credit1 = 0;  
 /\* Initializing credit for processID 2 \*/  
 int credit2 = 0;  
  
 /\* Looping all the blockchain records \*/  
 for (BlockRecord blockRecord: *blockChain*){  
 if (Integer.*parseInt*(blockRecord.getVerificationProcessID()) == 0)  
 /\* Incrementing credits for processID 0 \*/  
 credit0 += 1;  
 if (Integer.*parseInt*(blockRecord.getVerificationProcessID()) == 1)  
 /\* Incrementing credits for processID 1 \*/  
 credit1 += 1;  
 if (Integer.*parseInt*(blockRecord.getVerificationProcessID()) == 2)  
 /\* Incrementing credits for processID 2 \*/  
 credit2 += 1;  
 }  
 System.*out*.println("Process 0: " + credit0 + " Process 1: " + credit1 + " Process 2: " + credit2);  
 }  
  
 public static void main(String args[]) throws Exception {  
 int q\_len = 6;  
  
 /\* Checking for the Arguments that are received from command line \*/  
 if (args.length < 1)  
 *PID* = 0;  
 else if (Integer.*parseInt*(args[0]) > 2){  
 System.*out*.println("Process numbers are 0, 1, or 2");  
 throw new IllegalArgumentException();  
 }  
 else  
 /\* Reading the Process Id and storing it in a variable \*/  
 *PID* = Integer.*parseInt*(args[0]);  
  
 System.*out*.println("Satya Yoganand's Blockchain program. Ctl-c to quit\n");  
 System.*out*.println("Using processID : " + *PID* + "\n");  
  
  
 /\* Setting up ports for the current process with respect to processID \*/  
 new Ports().setPorts();  
  
 /\* Creating a blockChain new Variable \*/  
 BlockChain blockChain1 = new BlockChain();  
  
 /\* Initializing the publicKey with respect to porocess ID \*/  
 PublicKeyObject publicKey = *publicKeyInit*(*PID*);  
  
 System.*out*.println("PublicKey for processID : "+*PID*+" ----- "+ publicKey.getPublicKey());  
  
 /\* Generating a new thread to listen for the start message \*/  
 new Thread(new StartServer()).start();  
 /\* Generating a new thread to do listen for an incoming public key \*/  
 new Thread(new PublicKeyServer()).start();  
 /\* Generating a new thread to listen for incoming unverified blocks \*/  
 new Thread(new UnverifiedBlockServer()).start();  
 /\* Generating a new thread to listen for the blockchain \*/  
 new Thread(new BlockchainServer()).start();  
 System.*out*.println("Servers set, waiting for start signal...");  
 /\* If the processID is 2 then multicast the start message to all the running processes \*/  
 if (*PID* == 2){  
 blockChain1.multiCastStart();  
 }  
 /\* Inducing sleep so that the start message is multicasted and all servers are started up,then all the processes start doing the blockchain work at exactly the same time \*/  
 try{Thread.*sleep*(10000);}  
 catch(InterruptedException e){  
 /\* Printing Stack Trace \*/  
 e.printStackTrace();  
 }  
 /\* After receiving start message the blockChain process will Start \*/  
 if (*start*.equals("go")){  
 /\* Multicasting the public key \*/  
 blockChain1.multiCastPublicKey(publicKey);  
 /\* Inducing sleep so that all the processes will receive all the publicKeys \*/  
 try{Thread.*sleep*(10000);}  
 catch(InterruptedException e){  
 /\* Printing Stack Trace \*/  
 e.printStackTrace();  
 }  
 /\* Printing all the public keys on the console \*/  
 for (PublicKeyObject pubK: *publicKeyArray*){  
 System.*out*.println(pubK.getProcessID() + ": " + pubK.getPublicKey());  
 }  
 System.*out*.println("----------------------------------------------");  
 /\* Initializing the blockchain with a dummy block \*/  
 *blockChain* = *blockChainInit*();  
 /\* Read the files with respect to the processID and store the blocks in a block array \*/  
 *blockArr* = *readFile*(*PID*);  
 /\* Looping through blockArray so that all the blocks will be marked as Unverified and then added to priority Queue for further processing \*/  
 for (BlockRecord block: *blockArr*)  
 blockChain1.multiCastUB(block, Ports.*UnverifiedBlockServerPortBase*);  
 System.*out*.println("Unverified Blocks sent to Priority Queue...");  
 /\* Inducing sleep so that all the process will fetch all their unverified blocks \*/  
 try{Thread.*sleep*(4000);}  
 catch(InterruptedException e){  
 /\* Printing Stack Trace \*/  
 e.printStackTrace();  
 }  
  
 /\* The main driver code for the verification the blocks in the priority queue, doing work, and then putting them in the blockchain \*/  
 while (true){  
 /\* Inducing sleep on the thread so that the work will be done on each unverified block \*/  
 try{Thread.*sleep*(2001);}  
 catch(InterruptedException e){  
 /\* Printing Stack Trace \*/  
 e.printStackTrace();  
 }  
 System.*out*.println(*blockPriorityQueue*.size() + " unverified blocks remaining");  
 /\* Checking for the items in priorityQueue if null then break the loop \*/  
 BlockRecord tempBlock = *blockPriorityQueue*.poll();  
 if (tempBlock == null)  
 break;  
 BlockRecord verifiedBlock = new BlockRecord();  
  
 /\* Setting up a flag for block which refers whether the block is present in the blockchain or not \*/  
 boolean blockExists = false;  
 /\* Initializing a temporary public key variable \*/  
 String tempPubKey = "";  
 for (PublicKeyObject pub: *publicKeyArray*){  
 if (Integer.*toString*(pub.getProcessID()).equals(tempBlock.getVerificationProcessID())){  
 /\* Fetch the public key for the current process and store it in the temp variable \*/  
 tempPubKey = pub.getPublicKey();  
 System.*out*.println("Using the public key from process: " + pub.getProcessID());  
 }  
 }  
 /\* Converting this public key into bytes \*/  
 byte[] publicKeyInBytes = Base64.*getDecoder*().decode(tempPubKey);  
 /\* converting the signed blockID into bytes to get ready for verification \*/  
 byte[] idSignature = Base64.*getDecoder*().decode(tempBlock.getSignedID());  
 /\* converting the bytes of the public key into a public key object reference \*/  
 X509EncodedKeySpec publicKeySpec = new X509EncodedKeySpec(publicKeyInBytes);  
 KeyFactory keyFactory = KeyFactory.*getInstance*("RSA");  
 PublicKey RestoredKey = keyFactory.generatePublic(publicKeySpec);  
 /\* Now by using verify Sig method with the key and both byte arrays, verify if it is signed \*/  
 boolean verified = *verifySig*(tempBlock.getBlockID().getBytes(), RestoredKey, idSignature);  
 if(!verified){  
 System.*out*.println("This block is not signed by the correct owner of the private key, moving on...");  
 }  
 else {  
 try{Thread.*sleep*(1000);}catch(InterruptedException e){}  
 /\* checking if the block is already in the blockchain or not \*/  
 for (BlockRecord b: *blockChain*){  
 if (b.getBlockID().equals(tempBlock.getBlockID())){  
 blockExists = true;  
 System.*out*.println("Block already in blockchain");  
 }  
 }  
 /\* If the block is not present in the blockchain then start verifying it \*/  
 while (!blockExists){  
 System.*out*.println("Attempting to verify block");  
 /\* Do the work for the current block \*/  
 verifiedBlock = *Work*(tempBlock);  
 /\* Checking if blockchain was modified through looking at the winning hash of the head block,if it is different to a verified block's previous, chain has been modified \*/  
 String previousHash = *blockChain*.get(0).getWinningHash();  
 /\* If the block was abandoned, move on to the next block in the PriorityQueue \*/  
 if (verifiedBlock.getBlockID().equals("Abandoned"))  
 break;  
 if (!(verifiedBlock.getBlockID().equals("Abandoned"))){  
 /\* if the blockChain was not modified then add it to the blockchain \*/  
 if (verifiedBlock.getPreviousHash().equals(previousHash)){  
 System.*out*.println("Block verified, adding to blockchain and multicasting...");  
 *blockChain*.addFirst(verifiedBlock);  
 /\* Multicast the newly modified blockchain to the other processes here \*/  
 blockChain1.multiCastBC(*blockChain*, Ports.*BlockchainServerPortBase*);  
 /\* Change the status of the blockExists to true \*/  
 blockExists = true;  
 }  
 /\* If the block was modified \*/  
 else {  
 /\* Checking to see again if the block is in the blockchain, if it exists then start processing the next block in PriorityQueue \*/  
 for (BlockRecord b: *blockChain*){  
 if (b.getBlockID().equals(verifiedBlock.getBlockID())){  
 blockExists = true;  
 }  
 }  
 System.*out*.println("Attempting to work on block again...");  
 }  
 }  
 /\* Go back here if the same block needs to be worked on again \*/  
 }  
 }  
 /\* Go back here for a new block in the priority queue \*/  
 }  
 /\* Here is where the program is done processing all Unverified Blocks, hence blockchain is complete \*/  
 System.*out*.println("BLOCKCHAIN COMPLETE");  
 }  
 //enter loop to get console command(s) from user  
 BufferedReader in = new BufferedReader(new InputStreamReader(System.*in*));  
 try {  
 String input;  
 /\* Loop for accepting inputs from users \*/  
 do {  
 System.*out*.print("(l) to list blockchain, (c) to view credit, (q) to quit: ");  
 System.*out*.flush();  
 /\* The input will be either one of these which are "l","c" or "q" \*/  
 input = in.readLine();  
 /\* User Scenario if "l" is clicked \*/  
 if (input.equals("l")){  
 for (BlockRecord b: *blockChain*)  
 *printBlock*(b);  
 }  
 /\* User Scenario if "c" is clicked \*/  
 else if (input.equals("c")) {  
 *listCredit*();  
 }  
 /\* Entering "q" will end the loop and the exit message will get printed \*/  
 } while (!input.contains("q"));  
 System.*out*.println("Exited.");  
 } catch (IOException e) {  
 /\* Printing Stack Trace \*/  
 e.printStackTrace();  
 }  
 }  
  
}  
class Ports{  
 /\* Setting the StartServerBase to 4600 \*/  
 public static int *StartServerBase* = 4600;  
  
 /\* Setting the publicKeyServerBase to 4710 \*/  
 public static int *publicKeyServerBase* = 4710;  
  
 /\* Setting the UnverifiedBlockServerPortBase to 4820 \*/  
 public static int *UnverifiedBlockServerPortBase* = 4820;  
  
 /\* Setting the BlockchainServerPortBase to 4930 \*/  
 public static int *BlockchainServerPortBase* = 4930;  
  
 public static int *publicKeyServerPort*;  
 public static int *StartServerPort*;  
 public static int *UnverifiedBlockServerPort*;  
 public static int *BlockchainServerPort*;  
  
 public void setPorts(){  
 /\* Setting the StartServerPort to StartServerBase incremented with processID \*/  
 *StartServerPort* = *StartServerBase* + (BlockChain.*PID*);  
 /\* Setting the publicKeyServerPort to publicKeyServerBase incremented with processID \*/  
 *publicKeyServerPort* = *publicKeyServerBase* + (BlockChain.*PID*);  
 /\* Setting the UnverifiedBlockServerPort to UnverifiedBlockServerBase incremented with processID \*/  
 *UnverifiedBlockServerPort* = *UnverifiedBlockServerPortBase* + (BlockChain.*PID*);  
 /\* Setting the BlockChainServerPort to BlockChainServerBase incremented with processID \*/  
 *BlockchainServerPort* = *BlockchainServerPortBase* + (BlockChain.*PID*);  
 }  
}  
  
class BlockRecord{  
 /\* Block ID for the current Block \*/  
 String BlockID;  
 /\* Block Number for the current Block \*/  
 int blockNum;  
 /\* Time Stamp for the block \*/  
 String TimeStamp;  
 /\* VerificationID for the process \*/  
 String VerificationProcessID;  
 /\* PreviousHash for the block which is the winning hash of previous block \*/  
 String PreviousHash;  
 /\* Unique ID \*/  
 UUID uuid; // Just to show how JSON marshals this binary data.  
 /\* First Name for the current Block \*/  
 String fName;  
 /\* Last Name for the current Block \*/  
 String lName;  
 /\* SSN Number for the current Block \*/  
 String SSNum;  
 /\* Date of Birth for the current Block \*/  
 String DOB;  
 /\* Random Seed is our guess \*/  
 String RandomSeed;  
 String winningHash;  
 String signedID;  
 String signedWinningHash;  
 String Diag;  
 String Treat;  
 String Rx;  
  
 /\* Initializing setters and getters for all the variables defined \*/  
 public String getBlockID() {return BlockID;}  
 public void setBlockID(String BID){this.BlockID = BID;}  
  
 public int getBlockNum() {return blockNum;}  
 public void setBlockNum(int num){this.blockNum = num;}  
  
 public String getTimeStamp() {return TimeStamp;}  
 public void setTimeStamp(String TS){this.TimeStamp = TS;}  
  
 public String getVerificationProcessID() {return VerificationProcessID;}  
 public void setVerificationProcessID(String VID){this.VerificationProcessID = VID;}  
  
 public String getPreviousHash() {return this.PreviousHash;}  
 public void setPreviousHash (String PH){this.PreviousHash = PH;}  
  
 public UUID getUUID() {return uuid;} // Later will show how JSON marshals as a string. Compare to BlockID.  
 public void setUUID (UUID ud){this.uuid = ud;}  
  
 public String getSignedID() {return signedID;}  
 public void setSignedID (String sid){this.signedID = sid;}  
  
 public String getLname() {return lName;}  
 public void setLname (String LN){this.lName = LN;}  
  
 public String getFname() {return fName;}  
 public void setFname (String FN){this.fName = FN;}  
  
 public String getSSNum() {return SSNum;}  
 public void setSSNum (String SS){this.SSNum = SS;}  
  
 public String getDOB() {return DOB;}  
 public void setDOB (String RS){this.DOB = RS;}  
  
 public String getDiag() {return Diag;}  
 public void setDiag (String D){this.Diag = D;}  
  
 public String getTreat() {return Treat;}  
 public void setTreat (String Tr){this.Treat = Tr;}  
  
 public String getRx() {return Rx;}  
 public void setRx (String Rx){this.Rx = Rx;}  
  
 public String getRandomSeed() {return RandomSeed;}  
 public void setRandomSeed (String RS){this.RandomSeed = RS;}  
  
 public String getWinningHash() {return winningHash;}  
 public void setWinningHash (String wh){this.winningHash = wh;}  
  
 public String getSignedWinningHash() {return signedWinningHash;}  
 public void setSignedWinningHash (String swh){this.signedWinningHash = swh;}  
  
  
}  
  
/\* Class that holds the public key and processID of the current process \*/  
class PublicKeyObject {  
 String publicKey;  
 int processID;  
 /\* Initializing getters and setters for the defined variables \*/  
 public String getPublicKey(){return this.publicKey;}  
 public void setPublicKey(String pk){this.publicKey = pk;}  
  
 public int getProcessID(){return this.processID;}  
 public void setProcessID(int id){this.processID = id;}  
}  
  
/\* Worker thread to process incoming public keys \*/  
class PublicKeyWorker extends Thread {  
 /\* Creating a socket Variable \*/  
 Socket sock;  
 /\* Initializing a new gson variable \*/  
 Gson gson = new Gson();  
 PublicKeyWorker (Socket s) {sock = s;} // Constructor, assign arg s to local sock  
 public void run(){  
 try{  
 /\* Buffered Reader to read the input from input stream \*/  
 BufferedReader in = new BufferedReader(new InputStreamReader(sock.getInputStream()));  
 String publicKeyInput = in.readLine ();  
 System.*out*.println("Got key: " + publicKeyInput);  
 /\* Fetching the publicKey,ProcessID from PublicKeyObject using from JSon \*/  
 PublicKeyObject publicKey = gson.fromJson(publicKeyInput, PublicKeyObject.class);  
 /\* Adding the obtained publicKeyObject to publicKeyArray \*/  
 BlockChain.*publicKeyArray*.add(publicKey);  
 System.*out*.println("Public key Array : "+BlockChain.*publicKeyArray*);  
 sock.close();  
 } catch (IOException x){  
 /\* Printing Stack Trace \*/  
 x.printStackTrace();  
 }  
 }  
}  
  
class PublicKeyServer implements Runnable {  
 int q\_len = 6;  
 Socket sock;  
  
 public void run(){  
  
 System.*out*.println("Starting Key Server input thread using " + Integer.*toString*(Ports.*publicKeyServerPort*));  
 try{  
 ServerSocket servSock = new ServerSocket(Ports.*publicKeyServerPort*, q\_len);  
 /\* Accepting all the connections \*/  
 while (true) {  
 sock = servSock.accept();  
 /\* Spawn off the publicKeyWorker thread \*/  
 new PublicKeyWorker (sock).start();  
 }  
 }catch (IOException ioe) {  
 /\* Printing Stack Trace \*/  
 System.*out*.println(ioe);  
 }  
 }  
}  
  
class UnverifiedBlockWorker extends Thread {  
 Socket sock;  
 Gson gson = new Gson();  
 /\* Constructor for assigning the incoming connection to a local variable \*/  
 public UnverifiedBlockWorker(Socket s){  
 this.sock = s;  
 }  
 public void run(){  
  
 try{  
 /\* Buffered reader for getting the input Stream \*/  
 BufferedReader in = new BufferedReader(new InputStreamReader(sock.getInputStream()));  
 /\* Reading the input and stored it in a string \*/  
 String input = in.readLine();  
 /\* Converting it to BlockRecord object which holds all the Unverified blocks \*/  
 BlockRecord recordData = gson.fromJson(input, BlockRecord.class);  
 /\* Adding the block to priorityQueue \*/  
 BlockChain.*blockPriorityQueue*.add(recordData);  
 }catch(IOException e){  
 /\* Printing Stack Trace \*/  
 System.*out*.print(e);  
 }  
 }  
}  
  
class UnverifiedBlockServer implements Runnable {  
 int q\_len = 6;  
 Socket sock;  
  
 public void run() {  
 try {  
 /\* Setting up the listener relative to this process \*/  
 ServerSocket servSock = new ServerSocket(Ports.*UnverifiedBlockServerPort*, q\_len);  
 /\* wait for an incoming message from another process to receive the block \*/  
 while (true) {  
 /\* wait for the connection, put the connection in sock variable \*/  
 sock = servSock.accept();  
 /\* Spawn off an unverified block worker to process the block \*/  
 new UnverifiedBlockWorker(sock).start();  
 }  
 } catch (IOException e) {  
 /\* Printing Stack Trace \*/  
 e.printStackTrace();  
 }  
 }  
 }  
  
class BlockChainWorker extends Thread {  
 Socket sock;  
 Gson gson = new Gson();  
 /\* Constructor to assign the incoming sock to the local variable \*/  
 BlockChainWorker (Socket s) {sock = s;}  
 public void run(){  
  
 try{  
 BufferedReader in = new BufferedReader(new InputStreamReader(sock.getInputStream()));  
 /\* Reading the incoming JSon File \*/  
 String input = in.readLine();  
 /\* Converting the String input to array of BlockRecord through fromJson \*/  
 BlockRecord[] blockRecordData = gson.fromJson(input, BlockRecord[].class);  
 /\* Empty the Blockchain list first \*/  
 BlockChain.*blockChain*.clear();  
 /\* Now Adding all the blocks to blockchain \*/  
 for (BlockRecord record: blockRecordData){  
 BlockChain.*blockChain*.add(record);  
 }  
 /\* If the process 0 hears any new update it will rewrite it to a file \*/  
 if (BlockChain.*PID* == 0){  
 BlockChain.*writeToFile*(BlockChain.*blockChain*);  
 }  
 } catch (IOException x){  
 /\* Printing Stack Trace \*/  
 x.printStackTrace();  
 }  
 }  
}  
  
class BlockchainServer implements Runnable {  
 /\* Initializing sockets and q\_len as per socket protocol \*/  
 int q\_len = 6;  
 Socket sock;  
 public void run(){  
 try{  
 /\* setting up the listener relative to this process \*/  
 ServerSocket servSock = new ServerSocket(Ports.*BlockchainServerPort*, q\_len);  
 /\* wait for the incoming message \*/  
 while (true) {  
 /\* wait for connection, put the connection in sock variable \*/  
 sock = servSock.accept();  
 /\*spawn off a new Block chain Worker Thread \*/  
 new BlockChainWorker(sock).start();  
 }  
 }catch(IOException e){  
 /\* Printing Stack Trace \*/  
 e.printStackTrace();  
 }  
 }  
}  
  
class StartWorker extends Thread {  
 Socket sock;  
 /\*constructor for assigning incoming connection to a local variable \*/  
 public StartWorker(Socket s){  
 this.sock = s;  
 }  
 public void run(){  
 try{  
 BufferedReader in = new BufferedReader(new InputStreamReader(sock.getInputStream()));  
 /\* Setting up the Global start variable in Blockain class as per the incoming output typically "go" \*/  
 BlockChain.*start* = in.readLine();  
  
 }catch(IOException e){  
 /\* Printing Stack Trace \*/  
 e.printStackTrace();  
 }  
 }  
}  
  
class StartServer implements Runnable {  
 /\* Initializing sockets and q\_len as per socket protocol \*/  
 int q\_len = 6;  
 Socket sock;  
 public void run(){  
 try{  
 /\* Setting up the listener relative to this process \*/  
 ServerSocket servSock = new ServerSocket(Ports.*StartServerPort*, q\_len);  
 /\* Wait for an incoming message from another process \*/  
 while (true) {  
 sock = servSock.accept();  
 /\* Spawn off the Start Worker thread that reads the start message \*/  
 new StartWorker(sock).start();  
 }  
 }catch(IOException e){  
 /\* Printing Stack Trace \*/  
 e.printStackTrace();  
 }  
 }  
}