```java
package blockchaintask;

import java.nio.charset.StandardCharsets;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.sql.Timestamp;
import java.util.ArrayList;
import java.util.Scanner;

/**
 * class BlockChain
 * The class is used to create a chain of blocks
 * To see how chain gets corrupted, validate a chain and to repair a chain
 * We also check the time a particular function takes on the chain
 */
public class BlockChain {
    ArrayList<Block> blocks;
    private String chainHash;
    int hashesPerSecond;
    private String errorMessage = "";

    /**
     * Constructor BlockChain()
     * The constructor is used to initialize the list of blocks that will be connected to one another
     * The hash of the latest block and the hashes per second when there are 1 million hashes being computed
     */
    BlockChain()
    {
        this.blocks = new ArrayList<>();
        this.chainHash = "";
        this.hashesPerSecond = 0;
    }

    /**
     * function getTime
     * The get time function is used to get the
```

```
36 current time in milliseconds
37      * @return timestamp (time)
38      */
39     public Timestamp getTime()
40     {
41         return new Timestamp(System.currentTimeMillis
   ());
42     }
43 
44     /**
45      * function getLatestBlock
46      * the function is used to get the latest block
   of the chain. Recently added
47      * @return
48      */
49     public Block getLatestBlock()
50     {
51         return blocks.get(blocks.size() - 1);
52     }
53 
54     /**
55      * function getChainSize
56      * The function gets the size of the blockchain
57      * @return
58      */
59     public int getChainSize()
60     {
61         return blocks.size();
62     }
63 
64     /**
65      * function computeHashesPerSecond
66      * The function is used the compute 1 million
   hashes and check how many hashes the system can
   compute in one second
67      * @throws NoSuchAlgorithmException
68      */
69     public void computeHashesPerSecond() throws
   NoSuchAlgorithmException {
70         Timestamp startTime = getTime();
71         String hash =  "00000000";
```

```
 72          for(int i = 0; i < 1000000; i++) {
 73              MessageDigest digest;
 74              digest = MessageDigest.getInstance("SHA-
    256");
 75              //encoding with SHA-256
 76              byte[] encodedhash = digest.digest(
 77                      hash.getBytes(StandardCharsets.
    UTF_8));
 78          }
 79          Timestamp endTime = getTime();
 80          long totalTime = endTime.getTime() -
    startTime.getTime();
 81          this.hashesPerSecond = (int) ((1000000*1000
    )/totalTime);
 82      }
 83
 84      /**
 85       * function get hashesPerSecond
 86       * return the hashes per second to display to
    the user
 87       * @return hashes per second
 88       */
 89      public int getHashesPerSecond()
 90      {
 91          return hashesPerSecond;
 92      }
 93
 94      /**
 95       * function addBlock
 96       * It adds the new block provided to the user,
    after performing certain checks on it
 97       * @param newBlock a new block to be added to
    the chain
 98       * @throws NoSuchAlgorithmException
 99       */
100      public void addBlock(Block newBlock) throws
    NoSuchAlgorithmException {
101          //To check whether the block is valid or not
     before adding the block
102          StringBuilder sb = new StringBuilder();
103          //String builder to get the number of 0s
```

```
103 compared to difficulty
104         for(int i = 0; i < newBlock.getDifficulty
    (); i++)
105         {
106             sb.append("0");
107         }
108         //Calculate the hash of the new block
    created
109         String hash = newBlock.calculateHash();
110         //If the substring till the difficulty of
    the hash if equal to the string builder's number of
    0s
111         //and the previous hash of the new block is
    equal to the lastest block's hash only then add the
    block
112         //and assign the chainHash the new block's
    hash
113         if(hash.substring(0,newBlock.getDifficulty
    ()).equals(sb.toString()) && newBlock.
    getPreviousHash().equals(getLatestBlock().
    calculateHash())) {
114             blocks.add(newBlock); //add the block
115             this.chainHash = hash; //chainHash's new
     hash is the latest block's hash
116         }
117         else
118         {
119             //If verification fails
120             System.out.println("Addition failed due
    to failed verification or wrong previous hash.");
121         }
122     }
123
124     /**
125      * function toString
126      * The function is used to print the blockchain
    in the string
127      * @return
128      */
129     public String toString()
130     {
```

```
131         //sb to append the entire blockchain in a
    String
132         StringBuilder sb = new StringBuilder();
133         sb.append("{" + "\"ds_chain{\" : [ ");
134         //appending the block's to string to the
    block chain
135         for(int i = 0; i < blocks.size(); i++)
136         {
137             sb.append(getBlock(i).toString()+"\n");
138         }
139         sb.append("],\"chainHash\":\"" + chainHash
     + "\"}");
140         //return the string
141         return sb.toString();
142     }
143 
144     /**
145      * function getBlock
146      * To get the block at ith position
147      * @param i the position of the block
148      * @return the block
149      */
150     public Block getBlock(int i)
151     {
152         return blocks.get(i);
153     }
154 
155     /**
156      * function getTotalDifficulty
157      * Get the total difficulty of the entire block
    chain
158      * @return the total difficulty
159      */
160     public int getTotalDifficulty()
161     {
162         int totalDifficulty = 0;
163         //get the difficulty of every block and add
    it
164         for(int i = 0; i < blocks.size(); i++)
165         {
166             totalDifficulty += getBlock(i).
```

```
166 getDifficulty();
167         }
168         return totalDifficulty;
169     }
170 
171     /**
172      * function getTotalExpectedHAshes
173      * Get the total expected hashes of the
    blockchain
174      * @return the expected hashes
175      */
176     public double getTotalExpectedHashes()
177     {
178         double totalExpectedHashes = 0.0;
179         //Count the hashes based on the difficulty
    of every block and add them
180         for(int i = 0; i < blocks.size(); i++)
181         {
182             totalExpectedHashes += Math.pow(16,
    getBlock(i).getDifficulty());
183         }
184         //return the total expected hashes
185         return totalExpectedHashes;
186     }
187 
188     /**
189      * function isChainValid
190      * Check if the chain is valid or not.
191      * If it is corrupted at some point
192      * @return the boolean value of true or false
193      * @throws NoSuchAlgorithmException
194      */
195     public boolean isChainValid() throws
    NoSuchAlgorithmException {
196         //If there is ony 1 block in the chain
197         if(blocks.size() == 1)
198         {
199             //Get the block and create a string
    builder of 0s till its difficulty value
200             Block block = blocks.get(0);
201             StringBuilder sb = new StringBuilder();
```

```
202             for(int i = 0; i < block.getDifficulty
    (); i++)
203             {
204                 sb.append("0");
205             }
206             //calculate the hash of the block
207             String hash = block.calculateHash();
208             //If the substring of hash matchs the sb
    's zeros and the chainHash is equal to it's hash
    then it is valid
209             //Else the chain is not valid and return
     false with the 1st node being invalid
210             if(!(hash.substring(0,block.
    getDifficulty()).equals(sb.toString()) && chainHash.
    equals(hash))) {
211                 errorMessage = "..Improper hash at
    node 1. Does not begin with " + sb;
212                 return false;
213             }
214         }
215         //If the size of blockchain is more than 1
216         else
217         {
218             String hash = "";
219             //For every block in the chain
220             for(int i = 1; i < blocks.size(); i++)
221             {
222                 //Get the block and create a string
    builder of 0s till its difficulty value
223                 StringBuilder sb = new StringBuilder
    ();
224                 for(int j = 0; j < getBlock(i).
    getDifficulty(); j++)
225                 {
226                     sb.append("0");
227                 }
228                 //calculate the hash of the block
229                 hash = getBlock(i).calculateHash();
230                 //If the substring of hash matchs
    the sb's zeros and the previous hash of the block is
     equal to hash of the previous block
```

```
231                 //Else the chain is not valid and
    return false with the node that is invalid
232                 if(!(hash.substring(0,getBlock(i).
    getDifficulty()).equals(sb.toString()) && getBlock(i
    ).getPreviousHash().equals(blocks.get(i-1).
    calculateHash()))) {
233                     errorMessage = "..Improper hash
    at node" + i + ".Does not begin with " + sb;
234                     return false;
235                 }
236             }
237             //Also check if the hash that we get at
    the end of the loop is valid to the chain hash. If
    not, return false
238             if(!(hash.equals(chainHash)))
239             {
240                 errorMessage = "..Improper value of
    chainHash";
241                 return false;
242             }
243         }
244         return true; //return true if all cases pass
245     }
246 
247     /**
248      * function repairChain
249      * The function is to repair the chain when it
    is corrupted.
250      * A different message is set in place of the
    previous one for the block
251      * @throws NoSuchAlgorithmException
252      */
253     public void repairChain() throws
    NoSuchAlgorithmException {
254         //For every block
255         for(int i= 0; i< blocks.size(); i++)
256         {
257             //Get the block and create a string
    builder of 0s till its difficulty value
258             StringBuilder sb = new StringBuilder();
259             for(int j = 0; j < getBlock(i).
```

```
259 getDifficulty(); j++)
260             {
261                 sb.append("0");
262             }
263             //calculate the hash of the block
264             String hash = getBlock(i).calculateHash
    ();
265             //If the substring of hash matches the
    sb's zeros then the block needs no repairing,
266             // if not then calculate the proof of
    work of the block all over again
267             if(!(hash.substring(0,getBlock(i).
    getDifficulty()).equals(sb.toString()))) {
268                 getBlock(i).proofOfWork(); //
    Calculating the proof of work
269                 if(i+1 != blocks.size()) { // If it
    is not the last block then set the next block's
    previous hash as the hash just calculated
270                     blocks.get(i + 1).
    setPreviousHash(getBlock(i).calculateHash());
271                 }
272                 else
273                 {
274                     //If it is the last block then
    chainHash should be the block's hash
275                     chainHash = getBlock(i).
    calculateHash();
276                 }
277             }
278         }
279     }
280 
281     /**
282      * function setChainHash
283      * The function sets the chain hash of the block
     chain
284      * @param chainHash String
285      */
286     public void setChainHash(String chainHash)
287     {
288         this.chainHash = chainHash;
```

```
289     }
290 
291     /**
292      * getChainHash
293      * The function reti=urns the chain hash of the
    block chain
294      * @return String chain hash
295      */
296     public String getChainHash()
297     {
298         return chainHash;
299     }
300 
301     /**
302      * function getErrorMessage()
303      * sets the error message to the point of
    failure if the chain is valid or not
304      * @return the error message of the chain being
    valid or not
305      */
306     public String getErrorMessage()
307     {
308         return errorMessage;
309     }
310 
311     /**
312      * function setErrorMessage
313      * Setting the error message as back to empty
    once the chain verification failure point is known
314      * @param errorMessage the message if the chain
    is valid or not
315      */
316     public void setErrorMessage(String errorMessage)
317     {
318         this.errorMessage = errorMessage;
319     }
320 
321     /**
322      * function main
323      * The main function gives user the choices in
    order to manipulate the blockchain
```

```
324      * A genisis block is created and the user is
    then given options to add a transaction
325      * corrupt a chain or repair a chain
326      * If the chain is valid and display the chain
    along with the time taken by each function
327      * @param args
328      */
329     public static void main(String[] args) {
330         //Taking the user input
331         Scanner input = new Scanner(System.in);
332         //Creating a blockchain object
333         BlockChain blockChain = new BlockChain();
334         //Creating a genesis block wth difficulty of
     2
335         Block block = new Block(0, blockChain.
    getTime(), "Genesis", 2);
336         //Setting the previous hash of 1t block to 0
337         block.setPreviousHash("");
338         try {
339             //Computing the proof of work of the 1st
     block
340             block.proofOfWork();
341             //Adding the block to the chain
342             blockChain.blocks.add(block);
343             //Computing a million hashes per second
    and displaying the number of hashes done by system
    in a second
344             blockChain.computeHashesPerSecond();
345             //Setting the value of chainHash
    variable to the 1st block's hash
346             blockChain.setChainHash(block.
    calculateHash());
347         } catch (NoSuchAlgorithmException e) {
348             e.printStackTrace();
349         }
350
351         while(true)
352         {
353             //Menu choices for the user to choose
    from
354             System.out.println("\n0. View basic
```

```
354 blockchain status");
355             System.out.println("1. Add a transaction
     to blockchain.");
356             System.out.println("2. Verify the
    blockchain.");
357             System.out.println("3. View the
    blockchain.");
358             System.out.println("4. Corrupt the chain
    .");
359             System.out.println("5. Hide the
    corruption by repairing the chain");
360             System.out.println("6. Exit");
361 
362             int choice = Integer.parseInt(input.
    nextLine());
363 
364             switch(choice)
365             {           //Displaying the data of the
     blockchain. The chain size, difficulty of the
    latest block, total difficulty
366                         //the hashes per second
    computed above, total expected hashes by computing
    expected hashes of each block
367                         //Nonce of the latest block
    and the chainHash i.e the hash of the latest block
368                 case 0: System.out.println("Current
    size of the chain: "+ blockChain.getChainSize());
369                         System.out.println("
    Difficulty of the most recent block: "+ blockChain.
    getLatestBlock().getDifficulty());
370                         System.out.println("Total
    difficulty for all blocks: "+ blockChain.
    getTotalDifficulty());
371                         System.out.println("
    Approximate hashes per second on this machine: "+
    blockChain.getHashesPerSecond());
372                         System.out.println("Expected
     total hashes required for the whole chain: "+
    blockChain.getTotalExpectedHashes());
373                         System.out.println("Nonce
    for the most recent block: "+ blockChain.
```

```
getLatestBlock().getNonce());
                        System.out.println("Chain hash: "+ blockChain.getChainHash());
                        break;
                        //Adding a transaction to the block, asking the user for the difficulty and the transaction data
                        //For adding a block with 2 difficulty, the computation is fast, and it takes 0 milliseconds.
                        //For adding a block with difficulty of 3 it takes about 5 milliseconds
                        //The block with difficulty of 4 takes about 173 milliseconds
                        //The block with difficulty of 5 took 232 milliseconds
                        //The block with difficulty of 6 took 68888 milliseconds, we can see an increase in computation of time
                        // And for difficulty 7 it took 758503 milliseconds, indicating that as we increase the difficulty, the computation time increases exponentially
                case 1: System.out.println("Enter difficulty > 0");
                        int difficulty = Integer.parseInt(input.nextLine());
                        System.out.println("Enter transaction");
                        String data = input.nextLine();
                        Timestamp startTime = blockChain.getTime();
                        Block block1 = new Block(blockChain.getChainSize(), blockChain.getTime(), data, difficulty);
                        block1.setPreviousHash(blockChain.getChainHash());
                        try {
                            block1.proofOfWork();
                            blockChain.addBlock(
```

```
392 block1);
393                         } catch (
    NoSuchAlgorithmException e) {
394                             e.printStackTrace();
395                         }
396                         Timestamp endTime =
    blockChain.getTime();
397                         System.out.println("Total
    execution time to add this block was " + (endTime.
    getTime() - startTime.getTime()) + " milliseconds");
398                         break;
399                         //If the chain is valid then
     the time taken by the isChainValid function to
    verify is 0 milliseconds
400                         //The chain verification if
    the chain is invalid also takes 0 milliseconds,
    because as soon as the
401                         //chain finds a failure
    point it returns false and let's us know the point
    of failure.
402                         //The computation time is
    the same
403                 case 2: System.out.println("
    Verifying the entire chain");
404                         Timestamp startTimeValid =
    blockChain.getTime();
405                         try {
406                             boolean isValid =
    blockChain.isChainValid();
407                             if(blockChain.
    getErrorMessage().equals("")) {
408                                 System.out.println("
    Chain verification: " + isValid);
409                             }
410                             else
411                             {
412                                 System.out.println(
    blockChain.getErrorMessage());
413                                 System.out.println("
    Chain verification: " + isValid);
414                                 blockChain.
```

```
414 setErrorMessage("");
415                             }
416                         } catch (
    NoSuchAlgorithmException e) {
417                             e.printStackTrace();
418                         }
419                         Timestamp endTimeValid =
    blockChain.getTime();
420                         System.out.println("Total
    execution time to verify the chain was " + (
    endTimeValid.getTime() - startTimeValid.getTime
    ()) + " milliseconds");
421                         break;
422                         //This case is used to print
     the entire blockchain in the json format
423                 case 3: System.out.println("View the
     blockchain");
424                         System.out.println(
    blockChain);
425                         break;
426                         //The case helps in
    corrupting the data by asking the user the block
    that it wants to corrupt and
427                         //the new data for that
    block
428                 case 4: System.out.println("Corrupt
    the Blockchain");
429                         System.out.println("Enter
    the block ID of block to corrupt");
430                         int id = Integer.parseInt(
    input.nextLine());
431                         System.out.println("Enter
    new data for block " + id);
432                         String corruptData = input.
    nextLine();
433                         blockChain.getBlock(id).
    setData(corruptData);
434                         System.out.println("Block "
    + id +" now holds " + corruptData);
435                         break;
436                         //Repairing the block of the
```

```
436  chain with the difficulty of 3 when the data is corrupt takes about 248 milliseconds
437                          //It takes more time to repair the chain if the difficulty is greater, for difficulty 5 it takes about 2142 milliseconds
438                          //This indicates that the repair time depends on the block's difficulty and time taken to calculate the proof of work again.
439                          //The time taken also depends on the number of blocks that are corrupt and  need repairing
440                  case 5: System.out.println("Repairing the entire chain");
441                          Timestamp startTimeRepair = blockChain.getTime();
442                          try {
443                              blockChain.repairChain();
444                          } catch (NoSuchAlgorithmException e) {
445                              e.printStackTrace();
446                          }
447                          Timestamp endTimeRepair = blockChain.getTime();
448                          System.out.println("Total execution time required to repair the chain was " + (endTimeRepair.getTime() - startTimeRepair.getTime()) + " milliseconds");
449                          break;
450                          //Exiting the system if it is case 6
451                  case 6: System.exit(0);
452                          break;
453              }
454          }
455      }
456
457
458 }
459
```