Université d'Ottawa
Faculté de génie

École d'ingénierie et de
technologie de l'information

uOttawa
L'Université canadienne
Canada's university

University of Ottawa
Faculty of Engineering

School of Information
Technology and Engineering

Assignment 2 (9%~~5%~~) CSI2110

Submit to virtual campus by 23h55, October 22 ~~(19),~~ 2018

This is the initial handout; further information may be posted in virtual campus; it is your responsibility to follow further postings.

Assignments are INDIVIDUAL and the code must be your own independent creation; you need to interact with peers for output verification as specified in the testing section.

Programming a BlockChain

Bitcoin is a cryptocurrency created in 2008 used to perform online payments without requiring a trusted central authority. It is based on a technology that uses a distributed peer-to-peer system in order to record the transactions. This means that it is the Bitcoin community (the Bitcoin 'miners') that validates all the transactions using consensus across a very large network of peer-to-peer nodes. In exchange for their participation in the validation process, the miners receive Bitcoins. The data structure that maintains the ordered list of transactions is called a *blockchain* which basically is an immutable linked list of transactions. In this assignment, we will ask you to program your own blockchain. We have made several simplifications in the scheme to fit the scope of the assignment but it still covers many key concepts of blockchains.

Recording the transactions

The blockchain is the database used in order to record all transactions that occurred. In the case of the Bitcoin chainblock, each block contains a set of transactions. In our case, we will simply assume that a block contains only one transaction. This transaction is simply one person giving a certain amount of bitcoins to another person. Here is the simple class representing a transaction:

```
public class Transaction {

  private String sender;
  private String receiver;
  private int amount;

  …
  public String toString() {
   return sender + ":" + receiver + "=" + amount;
  }
}
```

Here `sender` is the username of the person giving money and `receiver` is the username of the person receiving the money; `amount` is the number of bitcoins involved in the transaction (here we assume that this is an integer value). Note that in reality, Bitcoin blockchain never stores the names of the sender and receiver, it rather use the concept of public and private keys in order to keep the transactions anonymous as well as the concept of digital signatures to ensure the sender was the person who created the transaction. This is however outside the scope of this assignment, for this reason, we will use simple usernames.

Once a transaction has been validated, a block is created and added to the end of the block chain. The format of a block is as follows:

```
public class Block {

  private int index;                  // the index of the block in the list
  private java.sql.Timestamp timestamp;  // time at which transaction
                                         // has been processed
  private Transaction transaction;  // the transaction object
  private String nonce;             // random string (for proof of work)
  private String previousHash;   // previous hash (set to "" in first block)
  //(in first block, set to string of zeroes of size of complexity "00000")
  private String hash;  // hash of the block (hash of string obtained
                           from previous variables via toString() method)

  …

  public String toString() {
   return timestamp.toString() + ":" + transaction.toString()
              + "." + nonce+ previousHash;
  }
}
```

Your blockchain will be contained in an `ArrayList` inside class `BlockChain`, so the `index` attribute should correspond to the position of the block in the `ArrayList`. The `timestamp` confirms the time at which the submitted transaction has been processed. It is obtained as follows when the block is constructed:

```
timestamp = new Timestamp(System.currentTimeMillis());
```

The last three attributes will be explained below, but before, we need to explain the concept of cryptographic hashing.

Cryptographic hashes

A one-way cryptographic algorithm is a function that is applied to an input, often a long string of characters and that returns as output a sequence of bits, called the hash code. A fundamental property of a cryptographic hash function is that it should be extremely difficult to invert the function as to find the input string that has produced the given output. The algorithm should also be such that it is very rare that two strings produce the same hash code (collision

resistance); in addition, any minor change in the input string (like changing just one character) should produce a completely different hash code.

Blockchains use cryptographic hashing in order to produce an immutable chain of blocks, i.e. a chain of block that would be very difficult to modify. More specifically, the Bitcoin blockchain uses a popular cryptographic algorithm called SHA-256 (Secure Hashing Algorithm). In this assignment, you will use a more primitive version of this algorithm, called SHA-1. This one produces a 160-bit long hash code (generally displayed as 40 hex characters). Note however that SHA-1 is today completely unsafe to use as it has been broken (meaning that it could be possible to get back the input string from the output code). For this reason it should never be used in real-life applications. We use it here again for simplification and to illustrate the functioning of a blockchain.

Here is for example, the hash code that corresponds to the string “csi2110”:
`99cd2c14 48ad356e 932283c0 94dcf288 68e99163`
And here is the one corresponding to “csi2510”:
`c63357ff 9bacb6ec 2203571a 77cba807 a038b1c7`
As you can see, even if the two strings only differ by one character, their respective hash codes are completely different.

In a blockchain, a hash is computed for every block and this hash value is used to link the blocks together. This hash is computer from the attributes of the block, which includes the hash code of the previous block. This is how the blockchain is made immutable. Indeed, if a malicious user would modify a value in one of the blocks in the chain (e.g. change the amount of money he received), then this chain would change also the associated hash code. As a result, the hash code of the next block would also change and so on. Consequently, changing one value in the chain would impose also changing all the subsequent blocks. By making sure all hash values in each block are valid, it becomes then possible to guarantee the integrity of the chain.

In this assignment, you will use the `toString` representation of a block in order to generate a hash code. Note that this representation includes a special attribute called the *nonce*. This is a random string that is used to make the creation of a block a more time-consuming task and thus making the blockchain more difficult to hack. The nonce is used to produce what is called a *proof of work*.

Proof of work

We just explained how a block can be created from a transaction and added to the chain by computing a hash value using a cryptographic algorithm. This is quick and easy to do. But how can we make this process more difficult? One way is to impose some conditions on the hash code associated to a block. Bitcoin imposes that a valid hash code must start with a certain number of zeros. How can we get such hash value? Remember that there is one attribute you can play with, the nonce. You therefore have to find a value for the nonce that will result in a hash code that will meet the requirement (here, number of leading zeros). And the only way to find it is by trial and error. The first miner who finds the solution, gets the privilege of having its

created block added to the chain and, in addition, gets a few bitcoins in exchange (this is how new bitcoins are added to the system).

Bitcoin adjusts the difficulty of this requirement as new computational power is added to the network. This is a way to make sure no one will control the chain by just having the most powerful farm of servers. Also, by making computationally intensive the task of adding new blocks, it makes even more difficult for someone to modify a block in the chain as this person would have to also recreate all the other blocks and this before another competitor inserts a new block! In fact, to take control of the block chain, a miner should have more computational power than the rest of the network combined.

More specificly in this assignment, the nonce is a string to be discovered by random trial, until the hash of the block has the desired number of zeros. The selection of nonce will be done after the other 5 variables in the block have been initialized. Because you will be printing on a textfile please limit the characters in the nonce to be visible characters (ASCII code that are integers in [33,126]). A pseudocode is given next:

```
Repeat
   Set block nonce to <next random string of visible characters>
Until ( SHA-1(block.toString()) starts with '00000' in hexa )
```

How to select a random string (second line above) is part of your algorithm design. We are not specifying string size but you should ponder about it. Long strings take space on the file, but if you insist on short strings the code above may run forever. As an extreme example of the latter observation, if you are adamant to only try strings of length 1character, you would be drawing a random string from a set with only 126-32=94 choices; it is very unlikely that one of these choices would yield a Sha-1 output with 5 leading zeroes in hexadecimal notation as required. You may be creative, but you need to be wise.

Instructions

Your task is to write a program that will insert new transactions in a blockchain. You therefore have to write the classes: `Transaction`, `Block` and `BlockChain`. Some details about clases `Transaction` and `Block` are given earlier in this handout; note that these details must be closely followed as they affect the validity of blocks that must be consistent for all the users (students) be able to check each other's blockchains. The class `BlockChain` will store the blockchain and contain whatever methods needed for its manipulation. One important requirement for the class `BlockChain` is to have a `main` method that will initiate the following sequence of execution.

`main` method of class `BlockChain`:

1. **Reading a blockchain from a given file:**
   Your `BlockChain` class should have methods that will read and write the block chain to a txt file. You are given a file named `blockchain.txt` that currently contains 3 blocks. The first step is therefore to read this file and add the blocks into an `ArrayList`. The methods must be as follows:

```
public static BlockChain fromFile(String fileName);
public void toFile(String fileName);
```

2. **Validating the blockchain:**
   You must then validate the blockchain by checking all the hashes, making sure they correspond to the values in the corresponding block. You must also check that all the `index` and `previousHash` attributes are consistent. In addition, we also ask you to validate all the transactions to make sure that noone spent bitcoins they do not have (this validation is similar to the validation described in step 3). Note that in the real blockchain, it is not possible to have invalid past transactions because these ones have been validated through consensus. Your `BlockChain` class should provide method:

   ```
   public boolean validateBlockchain();
   ```

3. **Prompting the use for a new transaction and verifying it:**
   If the blockchain is valid, then you ask a user to enter a new transaction by specifying a sender, a receiver and a bitcoin amount. To accept the transaction, you must verify that the sender has enough money to proceed with the transaction. To do this, you must go through all the blocks and accumulate the bitcoins assigned to this user minus the ones that have been spent by this user. To help you accomplishing this, you need to write a `getBalance(username)` in the `BlockChain` class.

   ```
   public int getBalance(String username);
   ```

4. **Adding the transaction into the blockchain:**
   If the transaction is found to be valid, then a block containing this transaction is added to the block chain.
   a. As proof of work, the hash of each inserted block chain must start with **5 zeros** (in hexadecimal notation). Your program must therefore find (by trial and error) a nonce that will make the hash code to meet this condition.

   A method to add a block to the block chain must be present:

   ```
   public void add(Block block);
   ```

5. **Asking for more transactions and back to 3.**
   Ask the user if he/she wants to add another transaction. If yes, then go back to Step 3.

6. **Saving the blockchain to a file with specific filename**
   Once the session is ended, you then save the updated blockchain into a text file (even if no new block has been created). To identify your file, we ask you to use a 'miner' id made of your uottawa email username. You then append to the filename your id preceded by an '_'. For example if the filename is `blockchain.txt` and your uottawa email is `jsmit001@uottawa.ca`, then the saved filename will be `blockchain_jsmit001.txt`

Requirements and advices:

- We provide a code for SHA-1 in class `Sha1`. Use the given method to avoid any unforeseen incompatibilities.
- We provide file `blockchain.txt` with initial 3 transactions.
- You must have classes `Transaction`, `Block` and `BlockChain` as specified. In particular you must follow precisely the same methods for toString() for `Transaction` and `Block` as described in the handout, as they affect hashes produced and are essential for the blockchain verification process.
- The TA must be able to run the `main` method of `BlockChain` class which must follow the sequence of steps above.
- You may create any extra methods you seem fit in these classes and are free to create new classes if needed.

Testing and Submission:

You also have to submit a blockchain text file that will include 10 transactions of your choice (added to the initial 3 provided in `blockchain.txt`). Your code should keep statistics of number of hash trials for obtaining each poof-of-work (nonce), which should be included in your report.

You also have to submit 3 other text files produced by three colleagues of your class. These additional three files will confirm that your blockchain is valid. For example, if you are miner `jsmit001` and you ask miner `ihack023` to verify your blockchain, then this one should produce a file called `blockchain_jsmit001_ihack023.txt`. The fact that a blockchain filename has been appended by a miner id means that this miner has found this blockchain to be valid.

In addition, you should also submit one blockchain file produced by a colleague in your class and that you have successfully validated. For example, if you have validated the blockchain of miner `jtrude173` then you would submit the file `blockchain_jtrude173_jsmit001.txt`

Your must submit a short typed report (suggested max 2 pages) in file `report.pdf` that contains:

- Your miner id (your uottawa email username; for example, `lmoura@uottawa.ca` has miner id `lmoura` and `laganier@uottawa.ca` has miner id `laganier`.
- Brief description of your classes and methods, and explanation of proof of work algorithm (nonce generation).
- A table that contains the statistics on hash trials for each of the 10 transactions you generated in the first text file (number of hash trials per transaction, and their average)
- Optional: any additional information to be seen by the marker.

We ask you to submit a zip file containing:

- your code: all `.java` files required to run your program
- a file `report.pdf` containing the report in PDF format (please specify miner id)
- the 5 blockchain text files specified.

Marking breakdown (20 points):

[3 pt] Code (all classes, well commented)
[3 pt] Report (classes description, proof-of-work description, table): suggested max 2 pages.
[4 pt] A blockchain `blockchain_<yourusername>.txt` created by your program containing original 3 transactions plus 10 additional transactions is valid.
[3 pt] 3 blockchain files from 3 different users confirming the validity of your blockchain have been submitted (format `blockchain_<your username>_<friend username>.txt`)
[3 pt] A validated file from another user has been submitted. You get the points only if the blockchain is indeed valid and your program is able to validate it
(format `blockchain_<some friend username>_<your username>.txt`)
[4 pt] Your blockchain class is able to detect all invalid blockchains in our tester class.
.

HAVE FUN, AND CHECK FURTHER INFORMATION/UPDATES IN BRIGHTSPACE AREA FOR ASSIGNMENT#2.