**Module Code:** CS3BC20

**Assignment report Title:** Blockchain Coursework Assignment

**Student Number (e.g. 25098635): 27016428**

**Date (when the work completed): 12/03/2021**

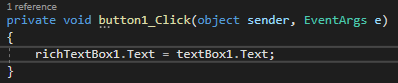
**Actual hrs spent for the assignment: 30hrs**

**Github link to code: https://github.com/hyounas/27016428\_BlockChain.git**

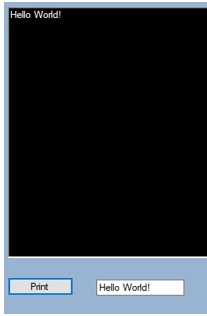
**Part 1 – Project Setup**

In Part 1, the objective is to setup the blockchain project on Visual Studios and implement changes to the UI which was empty, these changes include, buttons, textboxes etc. The main components of the project were located in BlockchainAssignment.csproj which included the ‘BlockchainApp.cs’, ‘Hashcode’ and ‘Wallet’ folder. The code is runnable and the UI can be accessed by default by clicking on the BlockchainApp.cs.

Since the code is mostly empty, our first task was to implement a button which was taken from the toolbox window and added into our UI. Later, we had to rename the button to ‘Print’ from its default name. This will be used to print out text on the large textbox within the UI. Heres the code on how to implement these two together:



Once this was done, we can now input our text, “HelloWorld” in the textbox which was dragged and dropped from the toolbox menu and click the print button to output text in the large textbox as shown:



**Part 2 – Blocks and the Blockchain**

**Implementing classes and more**

Firstly, we created two classes named Blockchain.cs and Block.cs. The ‘Blockchain’ class will consist of a chain of blocks, in order to do this, a list of variables is implemented into it which will in turn hold the blocks generated from the ‘Block’ class. Later on, we initialise the Blockchain class.

Once that is done, we can now add necessary variables in the Block.cs class such as timestamp, index, hash and hash of the previously known Block.

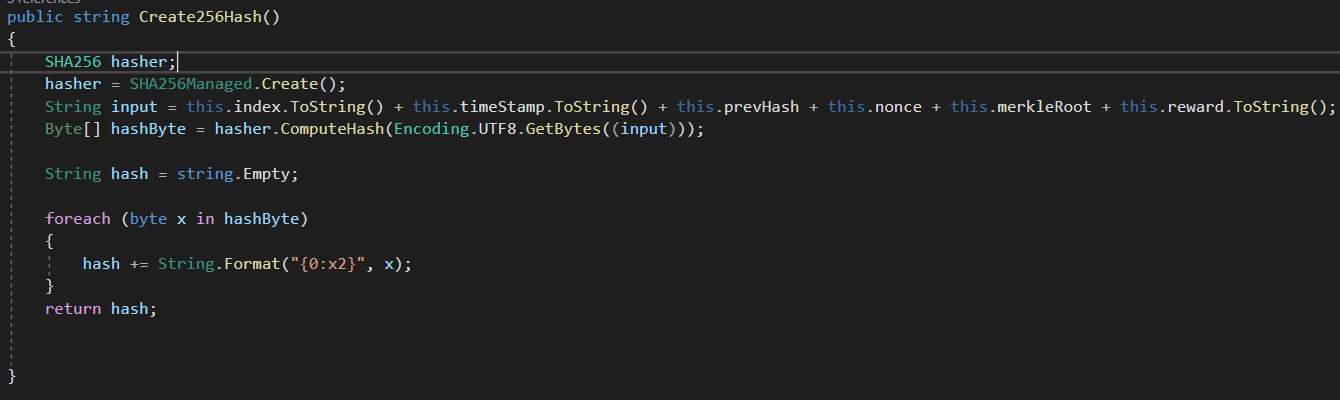
* Timestamp: This will help us indicate when a Block is made.
* Index: Position of the Block in the Blockchain.
* Hash: A Hash is basically a reference number for the Block. This Hash variable will show us the reference number for the current block.
* Previous Hash: Hash (identity) of the previous block.

Firstly, we create a constructor to implement these variables in the class. It has two arguments, hash of the prior block and index of the prior block.

**Hashing and creating the Genesis Block**

Genesis block is known as the very first block a block chain and that block is numbered 0 in latest versions where as previously, it was numbered block 1.

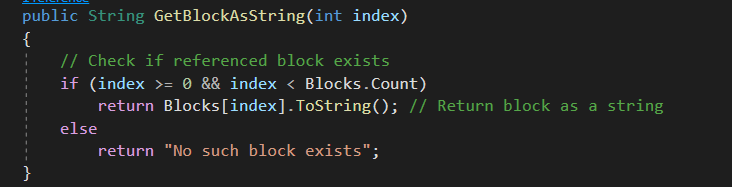
Firstly, we create a hash in order to finish assigning the variables timestamp, index, hash and previous hash in the class Block.cs. To do this, we add a new method called Create256Hash which will, as the methods named, create a hash (identity/reference number). This method will combine the index, timestamp, hash and previous hash.



Now, what left is to create a Genesis Block which will be used to generate blocks. To do this, a new constructor needs to be created in the Block.cs class. This constructor will take no arguments and will set the index to 0 and the previous hash to an empty string.

Once the Genesis Block has been created, new methods were added in Block.cs to return string containing the index, previous hash, hash and timestamp of the block. Later, a new method was implemented in the Blockchain.cs class that takes a Block index as an argument, calls the Create256Hash method and returns the output.

A button and textbox were added to the UI to print out the index on the large textbox.



**Part 3 – Transactions and Digital Signatures**

**Wallets & Private and Public Key Generation**

To create a wallet, we need a private key and a public key. A public key is a key that is available to all entities in a blockchain whereas a private is kept hidden from the public, as it gives control to the own of the wallet funds. Although public and private key pairs are mathematically linked, a private key cannot be solved (deciphered, decoded) solely through leveraging the public key's relationship. For the purpose of generating key pairs, there are many asymmetric encryption algorithms available and one of them is the Elliptic Curve Digital Signature Algorithm (ECDSA).

Firstly, new buttons were implemented in the UI, a button to create a new wallet, a button to create private and public keys and a button that validates keys.

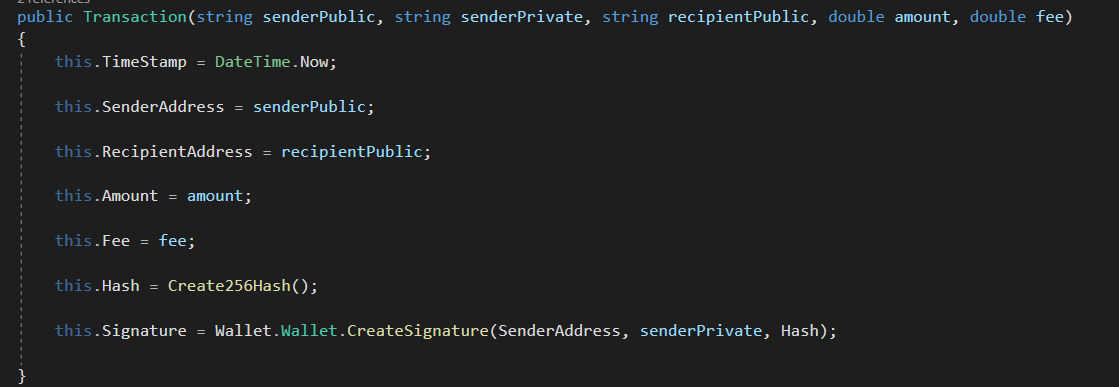


The button turns green when valid and then red if not.

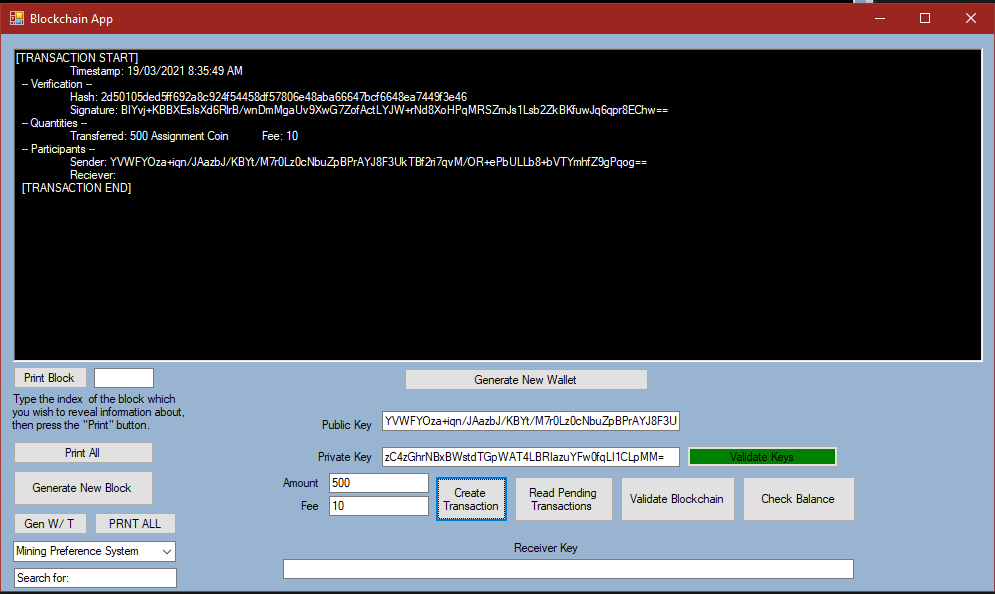
**Setting up Transactions**

Now whats left is creating transactions and to do this, we created a Transaction.cs class that will transfer fund between them. In this class, the following variables were implemented that is typically seen in a Blockchain transaction.

* Hash – hash of transaction and its contents
* Signature – hash of transaction signed with private key of sender.
* SenderAddress – the public key of receiver
* Amount – amount of currency being sent to the receiver.
* Fee – fee added to the transaction.



The number, Cost, Public Key, Private Key, and Receiver Key fields are passed through to a new instance of Transaction when the user clicks the ‘Create Transaction' button, and the Transaction is created. The transaction is then printed to the user interface's big text box as shown:



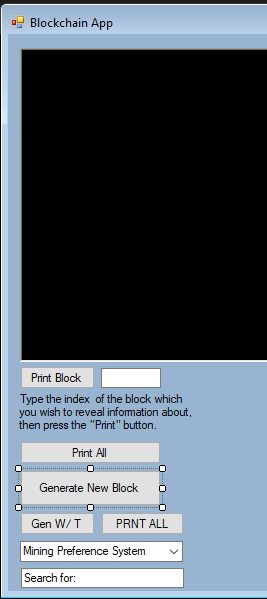
When a transaction is applied to the Blockchain, it is not verified or considered legitimate. Even so, it is not always approved right away; some services that recognise cryptocurrency as payment need a certain number of confirmations. When a transaction is first added to the Blockchain, it receives one confirmation; when the next block is added, it receives two confirmations. Each successive block adds a new confirmation. To reduce the effect of forks and attacks on the Blockchain, transactions are not accepted until they exceed a certain number of confirmations.

Therefore, a pending transaction list was added in the Blockchain.cs class that stores transactions that are pending.

**Part 4 – Consensus Algorithm (Proof – of – Work)**

**Generating new Blocks**

In this part, more blocks are needed to create a Blockchain. To do this, we added a new button called ‘Generate New Block’ in the UI which will allow us to add in new blocks. This button should call the Block constructor and pass it the variables from the previous block. Once a block is generated, it gets added in with the previous blocks which in turn creates a blockchain.



‘Generate new block’ creates a new block. ‘Print All’ prints all the blocks that have been generated.

**Adding transactions into Blocks**

The transaction pool that we built earlier will now be used. Transactions should be carried out inside blocks. To do this, a new data field is implemented in Block.cs class called ‘transactionList, which will contain transactions. A new Block should be followed by transactions from the transaction pool when it is formed.

It is important to know that blocks cannot take an infinite number of transactions from the pool.

**Proof-of-Work**

Proof-of-Work notes that a new Block must have a hash that reaches a certain complexity level before it can be added to the chain. Nodes compete in Proof-of-Work to build a Block that reaches this complexity threshold in the shortest time possible. Now, in order to implement Proof-of-Work, a new property needs to be added in the Block.cs class known as nonce. By default, nonce is set to 0 and a global difficulty threshold is setup that will be satisfied. The difficulty level is proportional to the number of 0’s at the beginning of the hash that constitute a valid block. Therefore, if the number of 0’s increases, the time it takes to mine will increase as well.

Characters in hex format are represented by SHA256, which means each character can be one of 16 different characters.

As a consequence, each additional zero in Proof-of-Work raises the complexity by a factor of 16.

Here a list of steps taken to implement the Proof-of-Work in the Block.cs class:

1. Nonce added in the Create256Hash() method, so that it is part of the hash composition process.
2. A new method Mine() was created that runs the Create256Hash() method within the while loop, which runs until the given hash starts with 0 correlating to the difficulty.
3. Next, the creat256Hash() method was renamed to Mine().
4. Nonce value was increased over and over if the hashing attempt failed.
5. Finally, Read() method was implemented to return the last value of nonce and the difficulty level at which the block was mined.

**Rewards & Fees**

After every block is mined, the user receives a reward for that mining in the form of virtual currency that is generated. This transaction is sent to the mining node address and the node that is sending this address is unknown, this is because the currency is generated at the time when the receiver mined.

On the other hand, Fees are a small amount applied to each transaction to encourage mining nodes to select their transaction for the next Block. The higher the fee, the more likely it will be selected by the node. When a node mines a Block successfully, they earn a flat reward as well as the accumulated fees from all transactions in the Block.

Here are the steps taken to implement rewards and fees:

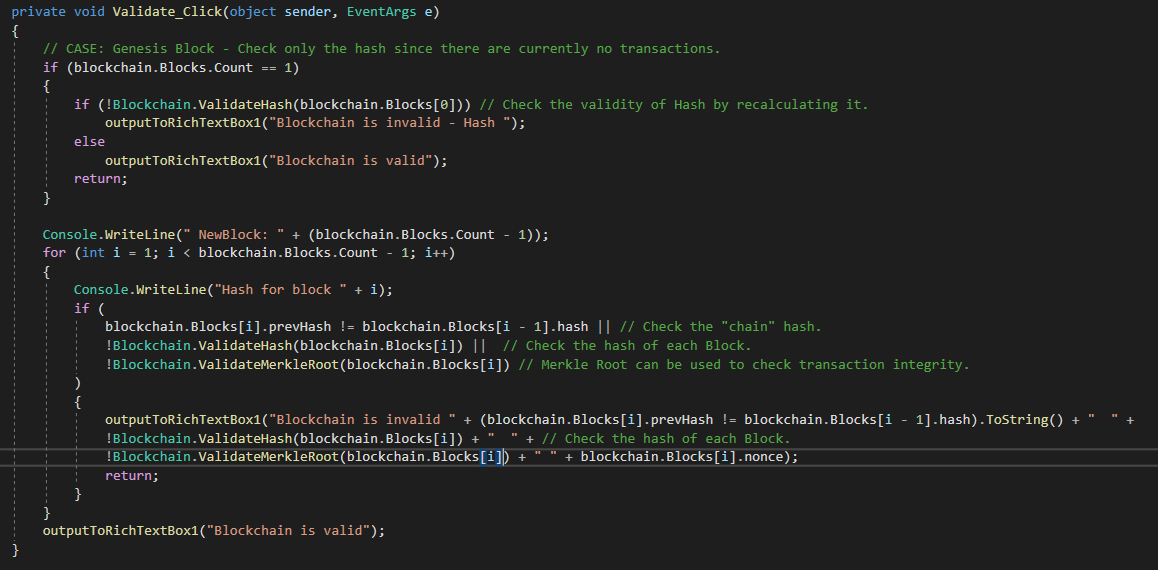
1. Added a fixed logic for rewards
2. The next move was to change the UI code such that the miner's public address is passed to the Block constructor.
3. Implemented new transactions to the list of transactions used by Block.t. Moreover, created a new method that calculates the reward and total fees owed and outputs the transaction in the large box.
4. Later, further code was updated to add the new transaction to the transaction list in the Block.
5. The reward was added to the hash composition used in Create256Hash()

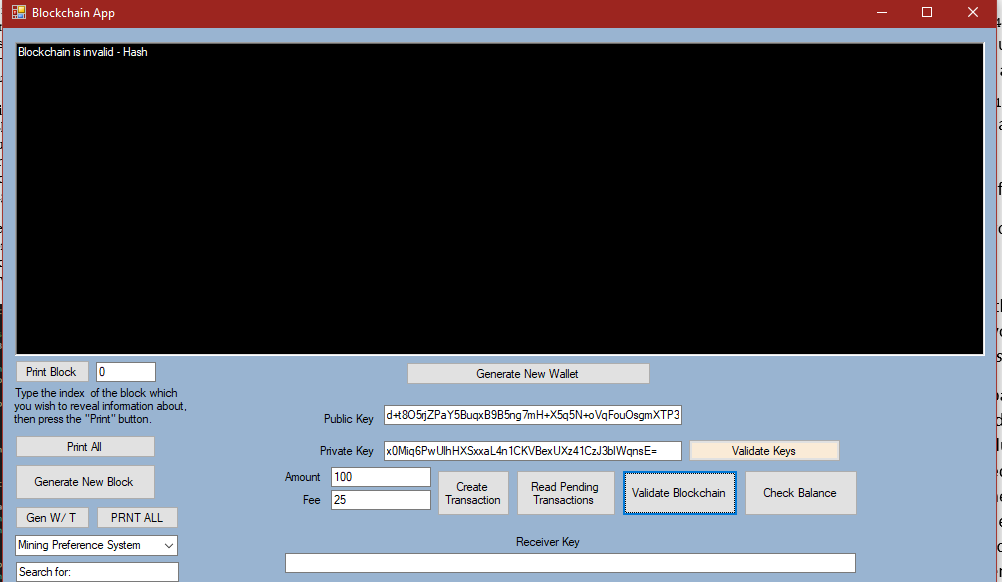
**Part 5 – Validation**

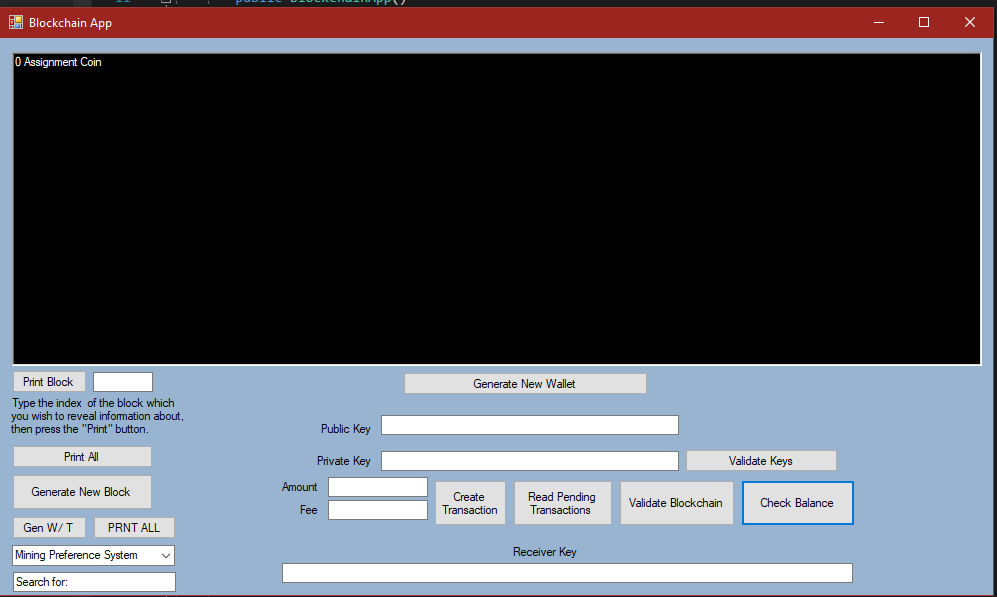
**Validating the Blockchain Structure**

A Blockchain is usually hosted through peer-2-peer networks, which is basically from user to user. Therefore, these type of nodes in a peer-2-peer cannot be trusted at all. To fix this problem, validation is required. We need to validate whether the Blocks are properly connected by seeing whether each Block is connected to the hash of the previous Block (Block Coherence, Contiguity).

Furthermore, a new button has been added to the UI that will validate the entire Blockchain to see if the connections are secure. This is done by looping through the blocks in the chain and seeing whether the hashes of each block matches with its previous block. Once all Blocks pass, the method returns a message, “Blockchain is valid – Hash”, otherwise, it returns “Blockchain is invalid – Hash”.







As shown in the above screenshot, a ‘Check Balance’ has been added to check the balance of a wallet and include further checks within the transaction. This is done to make sure that the sender has enough funds in the wallet before the transaction is allowed to proceed.

**Part 6 – Assignment Tasks**

**Task 1 – Threading**

The ThreadedMine() function in Block.cs has been used to implement threading.The Stopwatch feature was also used, as requested.

**Task 2 – Adjusting the Difficulty Level in Proof-of-Work**

The difficulty checkboxes visible in the UI gives us a boolean variable which triggers a difference in difficulty. If dynamic difficulty is chosen, an if loop in the class Block.cs is activated which alters the difficulty considering how much time it has taken.

**Task 3 – Mining Settings**

The UI has been setup for this but the actual code has not been implemented in the Block.cs class.