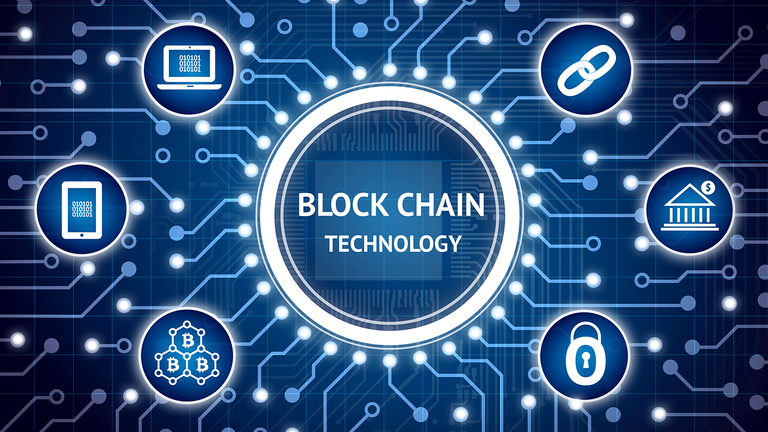
27016005 - CS3BC20

**BlockChain Computing**

Implementation of an offline CryptoCurrency 

horizontal line

Date of Completion: 19/03/2021

Actual hours spent on assignment: 40

Assignment Evaluation:

1. Assignment was interesting, fun, informative, engaging. Probably the best assignment throughout this degree. Lecturer put in an amazing amount of effort to make sure the students were understanding content and requirements.
2. Resources supplied were detailed, thorough, useful and applicable. Lecture content was related, helpful, interesting and applicable to the coursework. Detailed mark scheme was extremely useful and helpful. Lecturer was thorough and
3. Assignment was well defined, related to the course theory content, lecture content, practical content and did brilliantly bridging it all together in a way which effectively demonstrated the importance of the concepts.

Cannot fault this module. Taught amazingly, interesting, engaging, and the most effort from any lecturer in all the time at university. Thoroughly appreciated.

[**Part 1 - Project Setup & Preamble:**](#_nsliwbor150q) **3**

[**Part 2 - Blocks & the Blockchain**](#_fvvy1aka7w8s) **4**

[Hashing & Creation of Genesis Block:](#_j9sluswa4k0w) 5

[Hashing:](#_8dd7a769f0qr) 5

[Printing block to UI :](#_rdyzhsk5eony) 6

[**Part 3 - Transactions & Digital Signatures**](#_cum4y299ykbr) **8**

[Wallet Creation:](#_i3ikyycytlj3) 8

[Transactions:](#_ig2zl5lh7053) 8

[**Part 4 - Consensus algorithms**](#_a991aoc9kjad) **10**

[Generating new blocks:](#_ymv1rymygx8j) 10

[Adding Transactions to blocks:](#_69ht8351k2qz) 11

[Proof-of-Work:](#_m012ilh3vzmo) 11

[Rewards & Fees:](#_cc548u66kt5t) 12

[Evidence:](#_8smbynnkbdp4) 12

[**Part 5 - Validation**](#_n3akktxb30zc) **13**

[Blockchain Continuity Validation:](#_k8iuhr7911il) 13

[Block & MerkleRoot Validation](#_5sikplwh1c1s) 13

[Transaction Verification:](#_gbic7vorzpvl) 14

[**Part 6 - Assignment Tasks**](#_th65xnhmb3fz) **16**

[**Threading During Proof-of-Work**](#_6q43b64cu5pj) **16**

[**Adjusting the Difficulty in Proof-of-Work**](#_bk7p4hm2fxq8) **18**

[**Mining settings**](#_dtyccawdno3u) **19**

# 

# 

# Part 1 - Project Setup & Preamble:

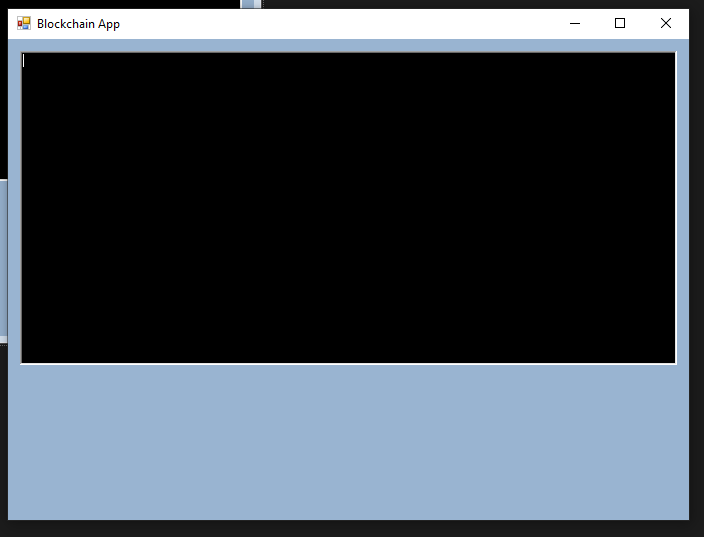
This Project was created in C#

Microsoft Visual Studio Community 2019 Version 16.2.5

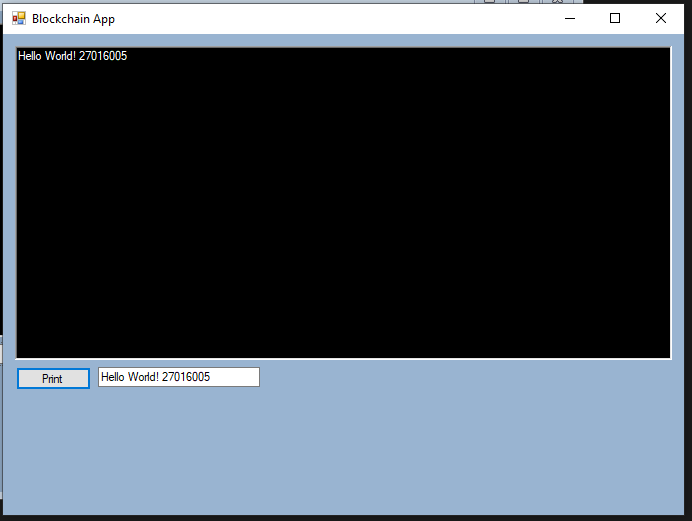
Using the .NET Framework Version 4.8.04084 - 2019

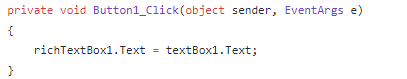
This code was maintained in a GitHub Repository: <https://github.com/Dannybrush/BlockChainProject>

Then ported to a CSGitLab Repository for submission.

A Foundation and Skeleton for the code was provided by the University of Reading Computer Science Department. This is available on the GitHub Repository: <https://github.com/Dannybrush/BlockChainProject/tree/main/Uni-Resources/CODE/Extracted/Practical_Exercises_Implementation_Environment>

This code errored when first downloaded, these errors were then corrected and the code was successfully run to produce this output:

A textbox and a button were then added to the User Interface (UI). Code was then implemented in such a manner that whatever was entered into the textbox at the time which the button was pressed would be printed to the “RichTextBox” (Large black output area). 

This was achieved using event handlers: 

Resulting in a Basic Hello World Program.

# Part 2 - Blocks & the Blockchain

The code used for part 1 contains the classes:

HashTools.cs

Wallet.cs

BlockchainApp.cs (+BlockchainApp.Designer.cs)

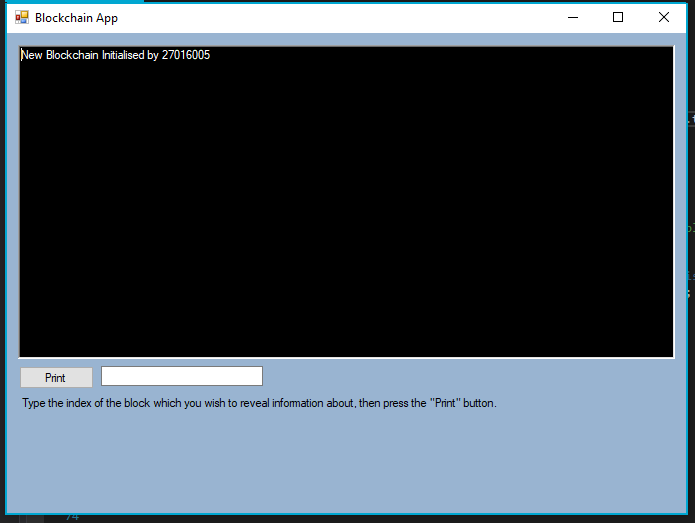
program.cs

More Classes were added for Part 2:

Blockchain.cs

Block.cs

A list of type Block was added to the Blockchain class to create a way to store the “Chain” of Blocks. The Blockchain was then initialised.



Fields were then added to the Block.cs class. In most programming languages this would take form of a variety of Public, Private, Protected, and possibly even global variables, C# utilises the ability of creating properties. This has been implemented for the following properties in the Block Class:

TimeStamp -> an indicator of when the block was initialised

Used for storing time, and uses DateTime.Now function which returns a DateTime

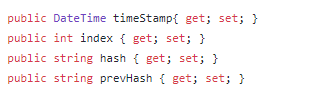
Index -> the position of the block in the blockchain

Position therefore always integer → type int

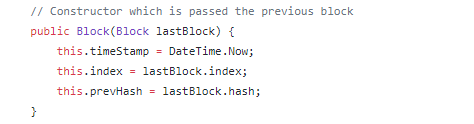
Hash -> the hash of the block

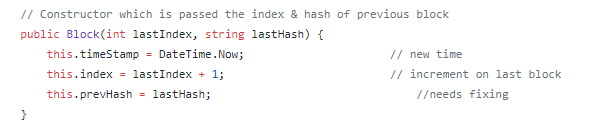
PrevHash -> the hash of the previous block

64 character Hexadecimal string produced by hashing function



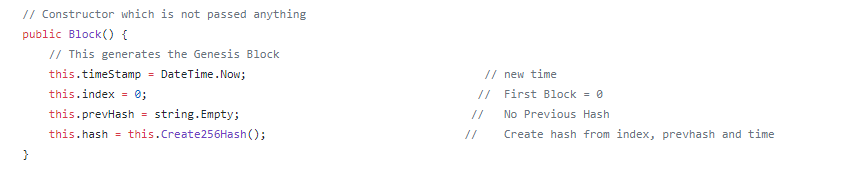
Three Constructors have then been created for the Block class. One Constructor which takes no input parameters, one which is overloaded with the input of the previous block, and one which just takes the index and hash of the previous block. Implementation of both of these latter two constructors is possibly redundant. The “DateTime.Now” function retrieves the current time.





### Hashing & Creation of Genesis Block:

A genesis block is the initial block in a blockchain, often quoted as “Block 0”



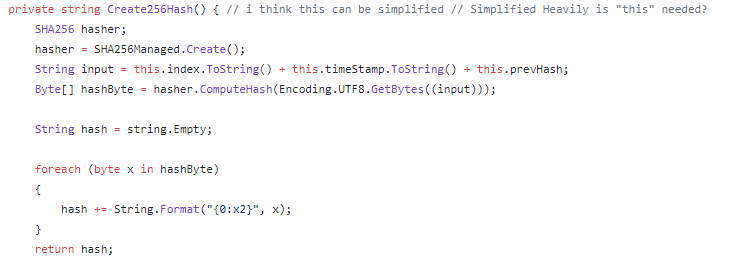
With the Genesis block being the first block in the blockchain, it is not possible for the last block to be passed into the constructor, therefore the index must be set to 0 and the hash must be created.

#### Hashing:

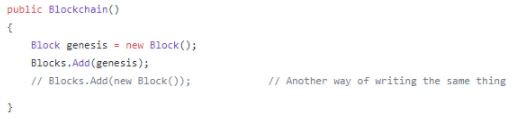
The hashing algorithm used, used SHA256 hashing, and is obtainable from

<https://emn178.github.io/online-tools/sha256.html>



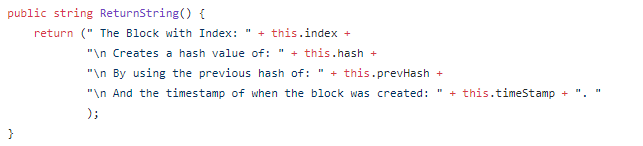
When implemented it looks like this: 

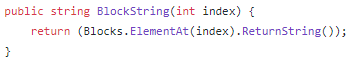
A default constructor was then added to the “Blockchain” class, this constructor calls the Genesis Block Constructor from the Block Class, and then the genesis block to the blockchain list created.



### Printing block to UI :

A function was then added to Block.cs which returns all the details about a given block in a string format. A further function was added to Blockchain.cs which prints the string to the output region given the strings index in the blockchain. The previously created buttons were repurposed for this effect.





This prints the details of the block with the index inputted into the UI Textbox.

# 

# Part 3 - Transactions & Digital Signatures

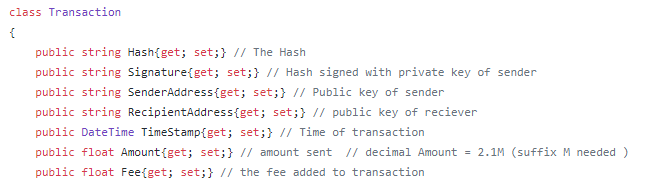
### Wallet Creation:

A new button is added to the UI, with the functionality of generating a new wallet.

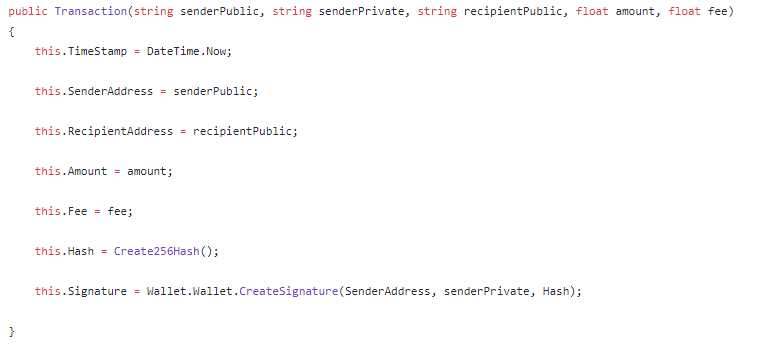
Two Textboxes are added which show the Public and Private keys on the UI, these also have labels to go with them. A “Validate keys” button was also added which confirms if the public and private keys in the respective textboxes are a matching pair. If the pair matches, the button turns green in colour. The methods implemented behind this functionality were provided in the original code skeleton as part of the wallet.cs class. The public key is 64 characters long, and the private key is 32 characters long.

### Transactions:

From here we add transactions to the blockchain. This is done by first creating a transaction.cs class, with all of the required properties:



From here a constructor is created for the transaction class, accepting both public keys, the amount, the fee and the sender's key as input parameters. These are assigned to their relative properties, the timestamp is then generated, the transaction hash is then calculated and finally the hash is signed using a signature which can be created from the static method “CreateSignature()” in the wallet.cs class. Note: The senderPrivate key is not stored.



Two more textboxes and labels were then added to the UI: Amount and Fee respectively.

Another Textbox and label was added for the receiver address. A “Create Transaction” button was also added.

When the “Create transaction” button is clicked: Amount, Fee, Public Key, Private Key, and receiver key are passed through to create a new instance of transaction. The Transaction is generated, and then printed. The transaction is also added to a transaction pool, which is a list of pending transactions stored in the blockchain.

# 

# 

# 

# 

# 

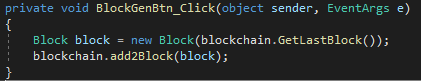
# 

# 

# Part 4 - Consensus algorithms

### Generating new blocks:

A button was added to the UI with the text “Generate New Block”, this button adds a new block to the blockchain, passing the previous block as the argument for the constructor.



Methods were implemented to print all the blocks in a block chain in its entirety. This was in addition to the functionality of printing a chosen specified block.

# 

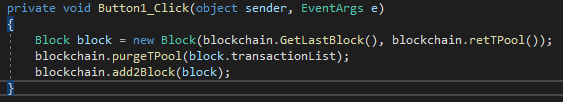
### Adding Transactions to blocks:

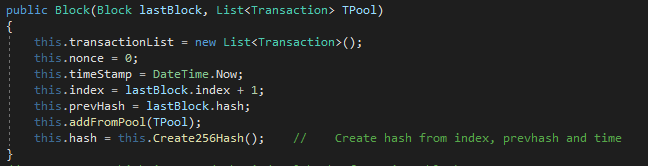
A List for holding transactions was adding to the block.cs class.

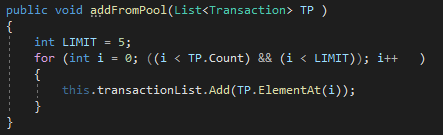


A function was then implemented which A new button was then again added to the UI to Generate Blocks which take transactions from the transaction pool. The blocks cannot take all the blocks so a finite limit was added - currently this is set to an arbitrary value of 5.

The Block class constructor was then adapted to accept a list of transactions as well as the previous block. The pending transaction pool is passed into the constructor, where the block retrieves a maximum of 5 transactions from the pool. These transactions are then purged from the pending transaction pool.





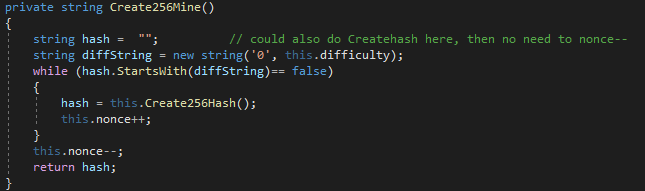


### Proof-of-Work:

Proof-of-work (PoW) is the standardised consensus algorithm for BitCoin and many other existing Cryptocurrencies. PoW dictates that for a block to be added to the blockchain it must have a hash which satisfies a given difficulty threshold. The chosen default difficulty level for this implementation was 4, this is because it could be calculated in a reasonable time while maintaining consistent results.

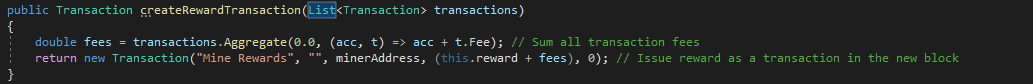
For the proof of work algorithm to work, the block must have a “nonce” field else every time the hash is created, it would be identical, by including a nonce (Number Only used oNCE) and changing this every time the hashing algorithm does not produce a hash of sufficient difficulty different hashes can be created.

Due to the hexadecimal nature of the hashing algorithm, each increase in difficulty level is an increase, factor 16 of workload. The Difficulty level is determined by a certain criteria in which the hash must meet. In this instance, leading “0” values have been chosen, thus a difficulty level of 4 means a hash must start with “0000” and difficulty level of 2 means it must start with “00”. Difficulty has been included in the hash composition.

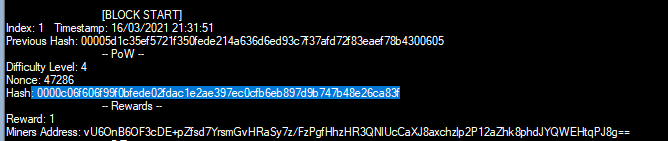
**Proof of work, Nonce Gen, Difficulty levels, Rewards and Fees. **

### Rewards & Fees:

After a block is successfully mined, the miner receives a reward. This is a flat reward given in the form of a transaction from a “Reward” wallet. Rewards are the most obvious motivation and incentive for blockmining. The implemented reward system is a hardcoded fixed value, however a similar option as bitcoin could be implemented where the reward changes with every x blocks mined. Alternatively rewards could be based on number of calculations to make a block, or the time taken per block.

****

#### Evidence:

****

**For a Series of screenshots showing the addition of blocks, creation of transactions and more etc see appendix.**

# 

# Part 5 - Validation

The concept of a blockchain network revolves around the idea of a “trustless” network where no node in the network can be trusted, and only the system can be trusted.

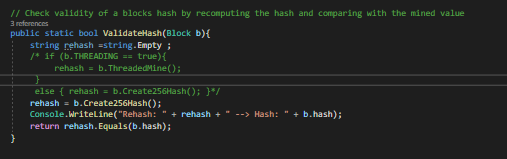
### Blockchain Continuity Validation:

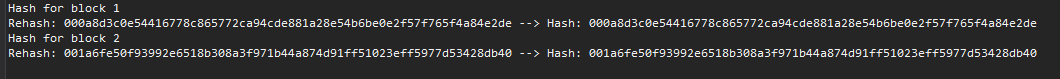
As the blockchain is a linear structure of connected blocks, it is possible to check the integrity by checking that for the entire blockchain, the previous hash is equal to the hash of the previous block. This iterates through the blockchain making sure that for the entire chain, the “PreviousHash” of the current block is equal to the “CurrentHash” of the previous block.

### Block & MerkleRoot Validation

Block Validation :

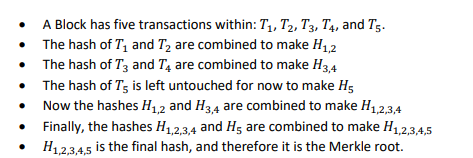
Rehashes the block using all of the properties, checks to see if the hash matches the original hash.

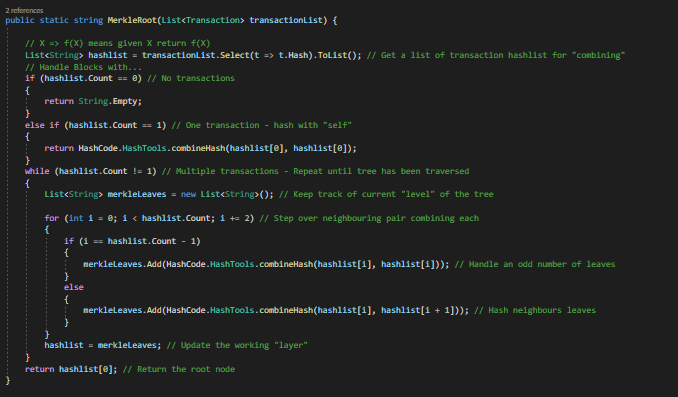




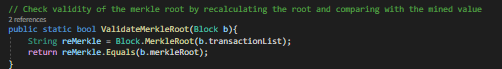
Merkle Root Validation:

The merkle root algorithm is used to iteratively combine the transaction hashes for a block until only one hash remains.



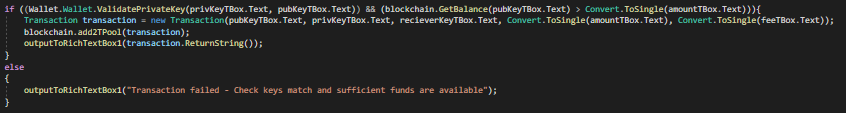


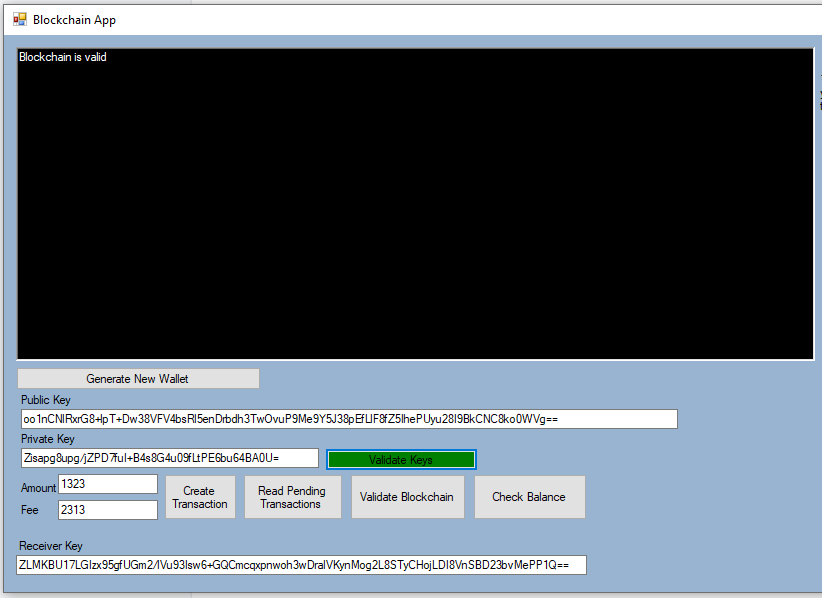
This is then validated upon clicking the “Validate” button. Where it recalculates the merkle root and checks the two match.

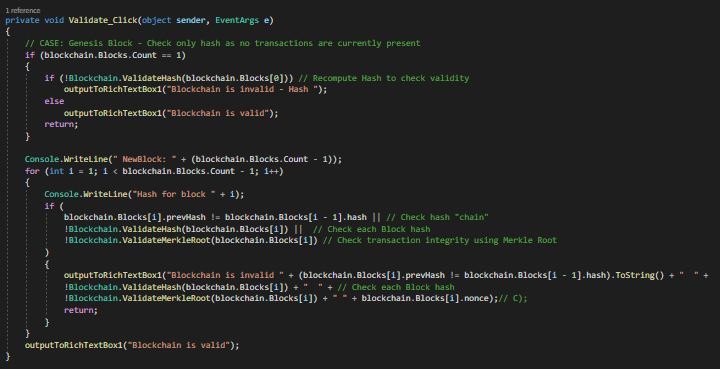


### Transaction Verification:

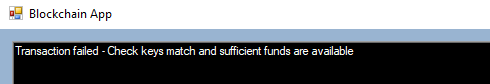
Validation has also been applied to the Transactions, so that when transactions are created, the keys must match, and the wallet must have sufficient funds to support the transaction.

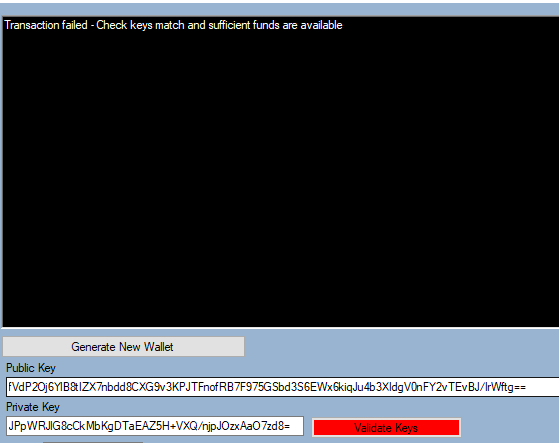






Also note the Validate keys is green confirming that the keys match.





# 

# 

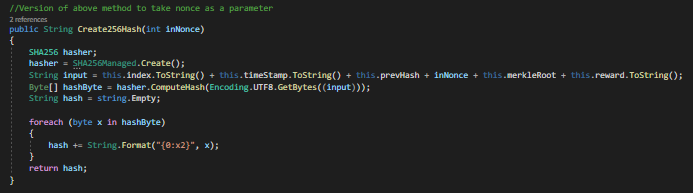
# Part 6 - Assignment Tasks

# Threading During Proof-of-Work

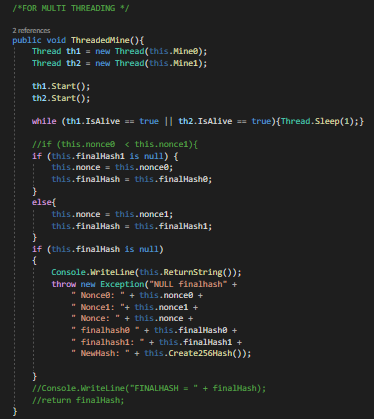
By adding a property to the block to say if threaded mining was activated or not.



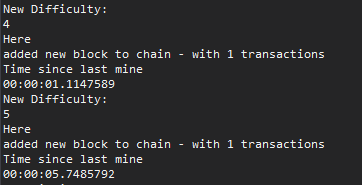
Adapting the mining function to accept an overload value containing the nonce:



Creating two threads, which split the workload in half, one working with odd number Nonce values and one working with even:



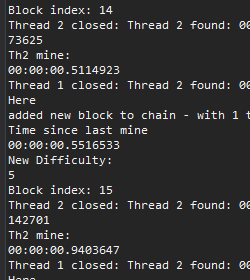
Two separate Nonce values were established, one for each thread. One thread handled all even value nonce, while the other handles all odd valued nonces, this means both threads could increment by two each time, halving the workload and preventing any duplication of work. When one thread found a solution, both threads were killed.



For difficulty 4 and 5 the non-threaded version produced these values:

4: 01.1147589 Seconds

5: 05.7485792 Seconds



The threaded versions produce these values:

4: 00.5114923

5: 00.9403647

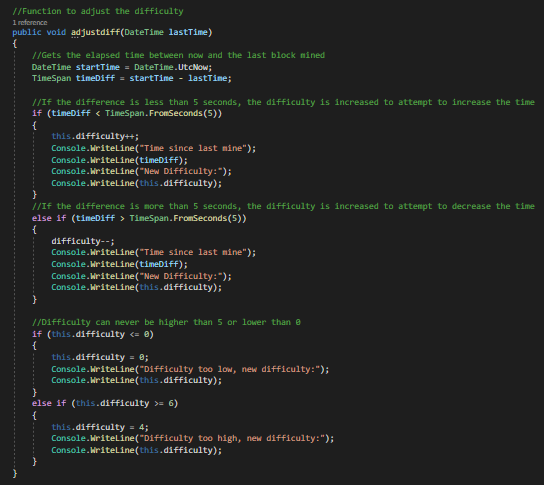
This shows that threading has a major effect on the result and time taken for the hashing process.

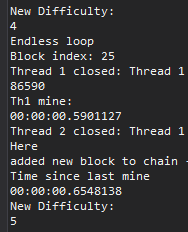
# Adjusting the Difficulty in Proof-of-Work

Variable difficulty was implemented by creating a desired block time; in this case 5 seconds was chosen. If the previous block took more than 5 seconds to be generated, the difficulty decreases, if the previous block took less than 5 seconds to be generated the difficulty increases. For testing purposes, this was limited to difficulty values of 0-6 due to the hexadecimal exponential nature of the increase in time taken with each increase in difficulty level.

A further option could be to implement a hexadecimal level of difficulty where instead of just adding another “0” to increase the difficulty it could add a threshold based on the hexadecimal value of the string. This would allow for more extreme fine tuning and [[1]](#footnote-0)consistency between block generation periods.

For example, target = 0000 -> decrease difficulty → target <= 0001 etc. this gradually adjusts the difficulty with a 16x higher accuracy. This is similar to how BitCoin & Ethereum handle difficulty. (See links below)





This shows the program successfully finding a result in quicker than the expected time, and thus increasing the difficulty: this is debugging output and not available to the user.

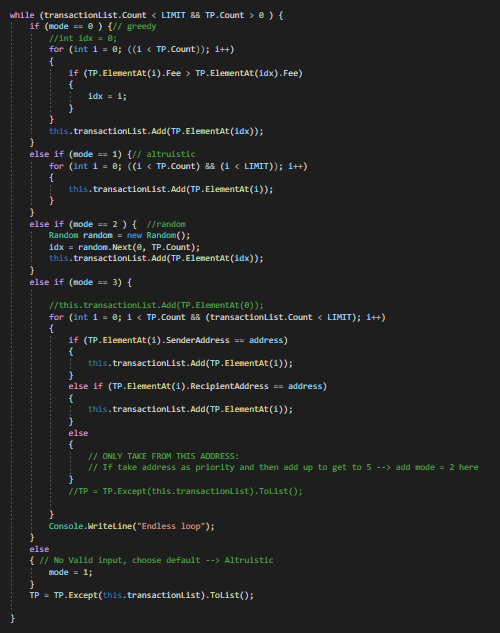
# Mining settings

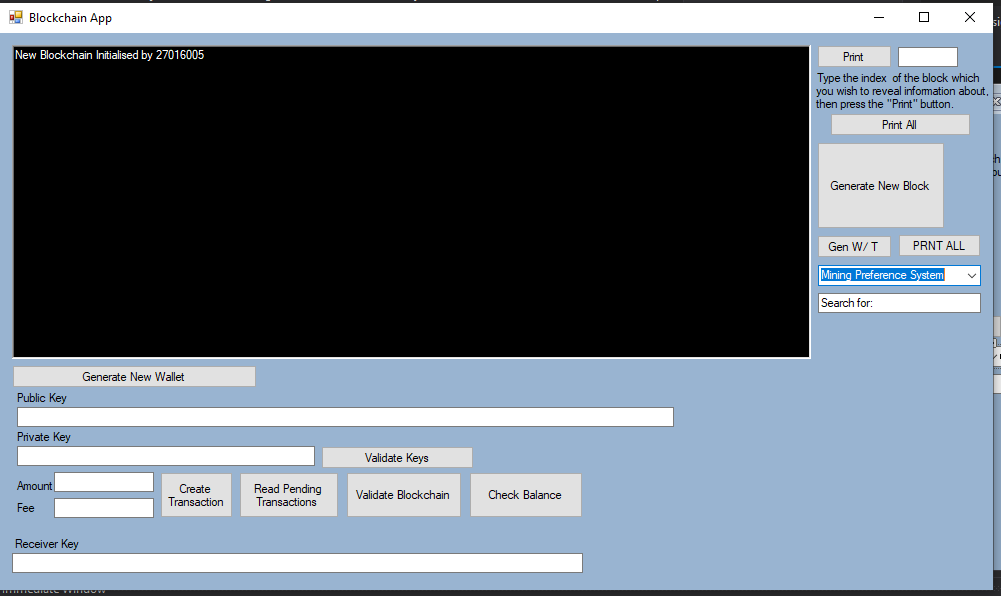
A combo-box was implemented with 4 options of how to prioritise the order of transactions from the pending pool. These were: Greedy; Altruistic; Random; Addressed based.

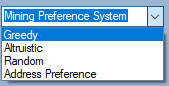
Greedy method prioritises the transactions with the highest fees.

Altruistic method operates on a first in- first out style methodology.

Random selected a random order.

Addressed based takes any transactions to or from a given address. 

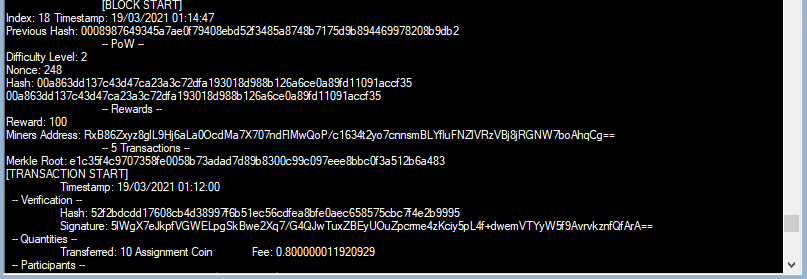




By Default, altruistic is chosen:

Here is proof of greedy:

For Transactions of 10 coins for the from the same wallet, to the same wallet, with only the fee changing -> between 1 and 0 at 0.1 increments randomly ordered.



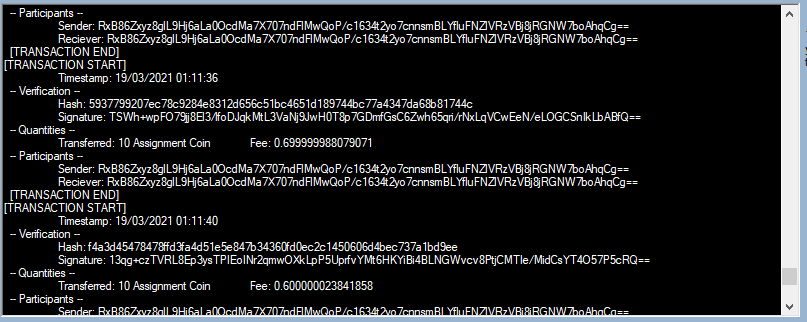
Use Cases:

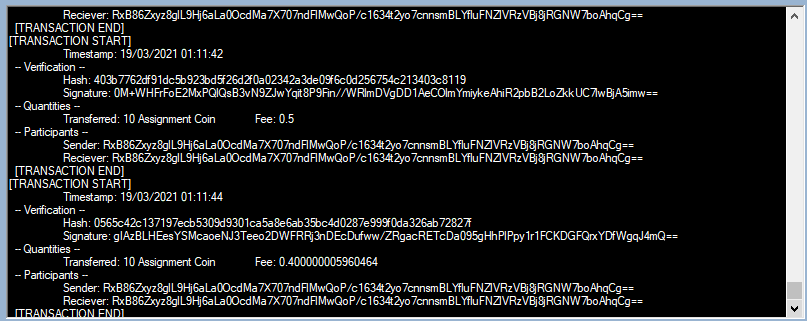
**Greedy**: Prioritise highest fees = most profit → business model

**Altruistic**: Fi-Fo → customer satisfaction / fair model. No Competition between miners resulting in more stability. Predictable.

**Random**: Fair model.

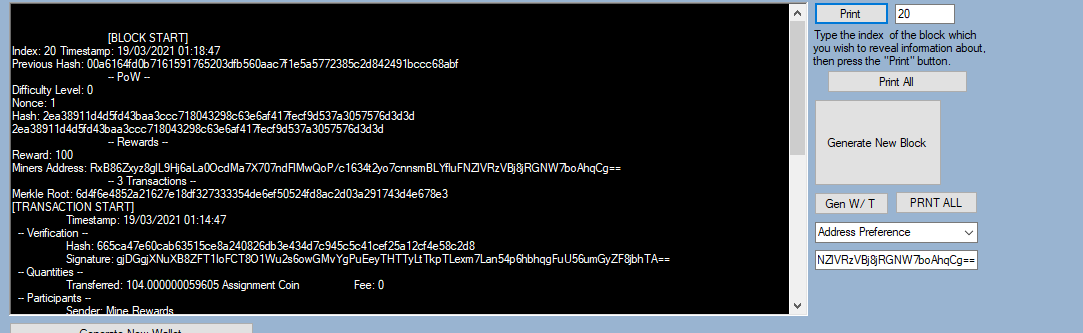
**Address Preference:** Analytics / Statistics, personal tracking





This shows that no matter which order the transactions are created in, they can be processed with whichever method is selected.

Here is proof with the Address preference functionality:



References:

<https://en.bitcoin.it/wiki/Difficulty>

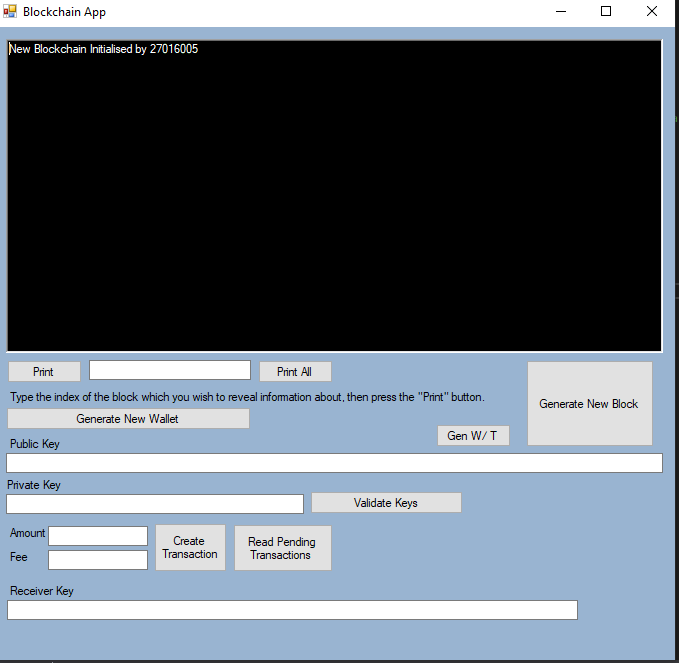
<https://en.bitcoinwiki.org/wiki/Difficulty_in_Mining>

<https://medium.com/coinmonks/part-5-implementing-blockchain-and-cryptocurrency-with-pow-consensus-algorithm-a7f8853d23dc>

APPENDIX:

Showing Pending pool → Pre-extension tasks.

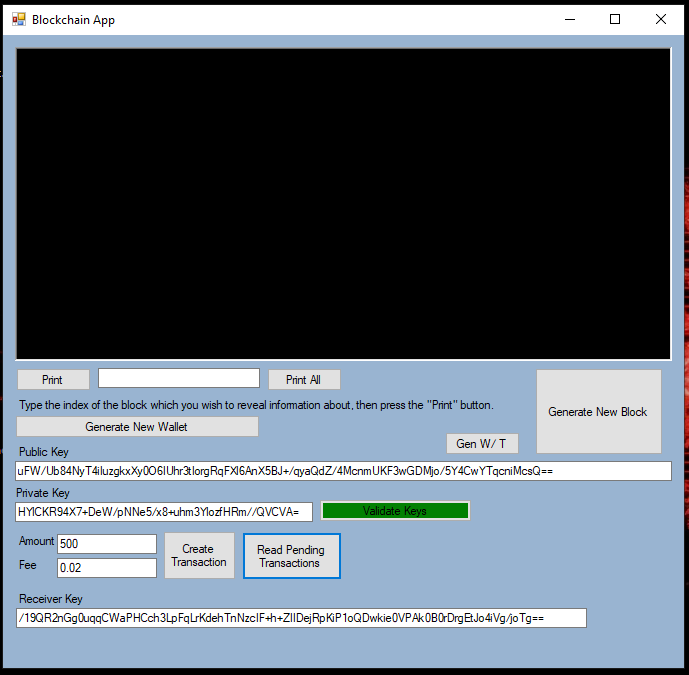
Initialise:



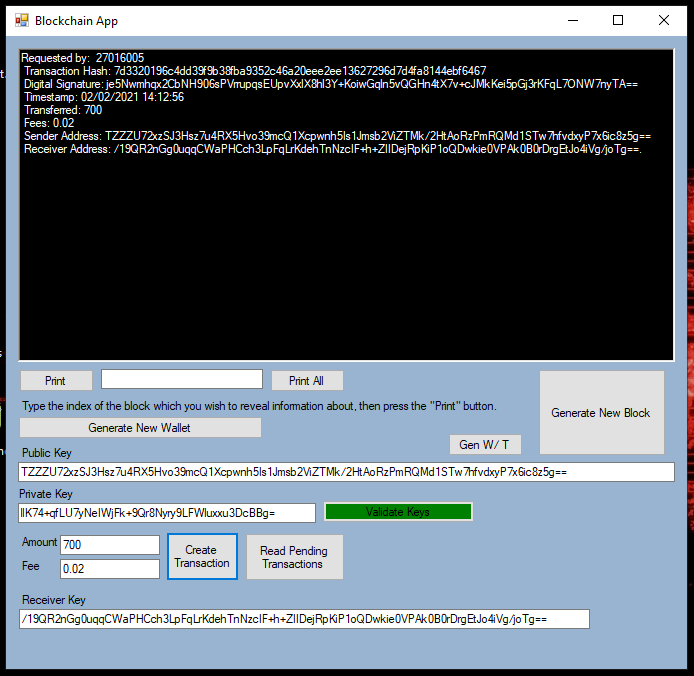
Create 3 (plus genesis) blocks (Printall)

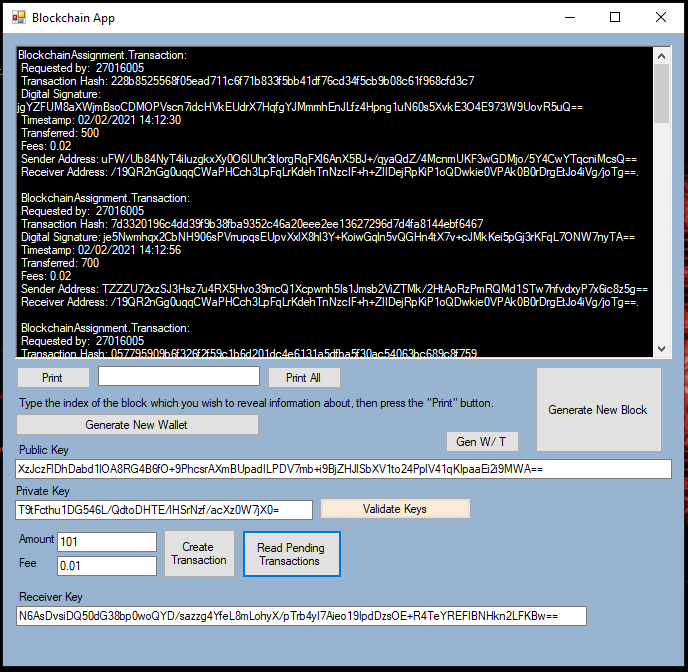
# 

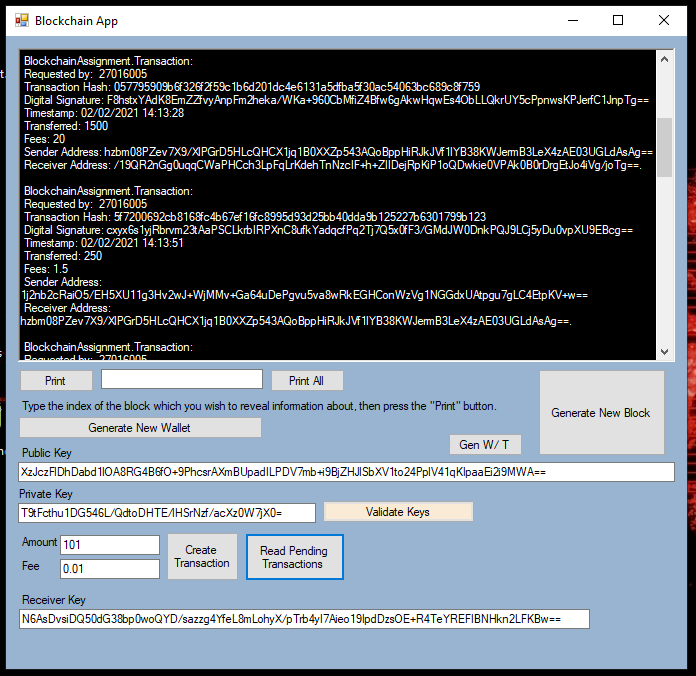
View transactions:

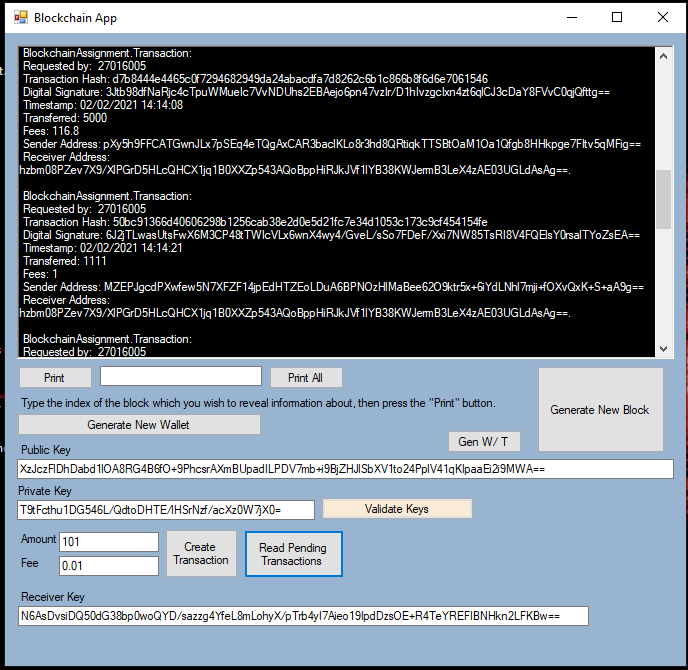


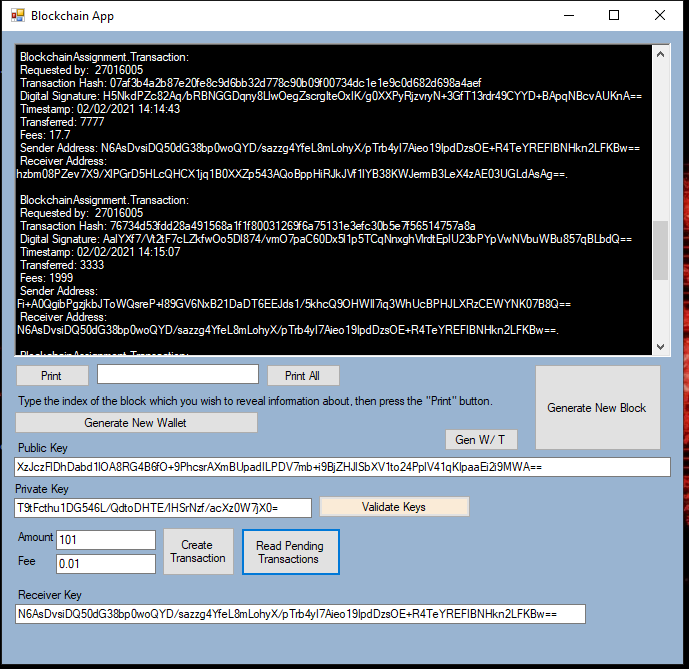
Val keys + Add 7 Transactions

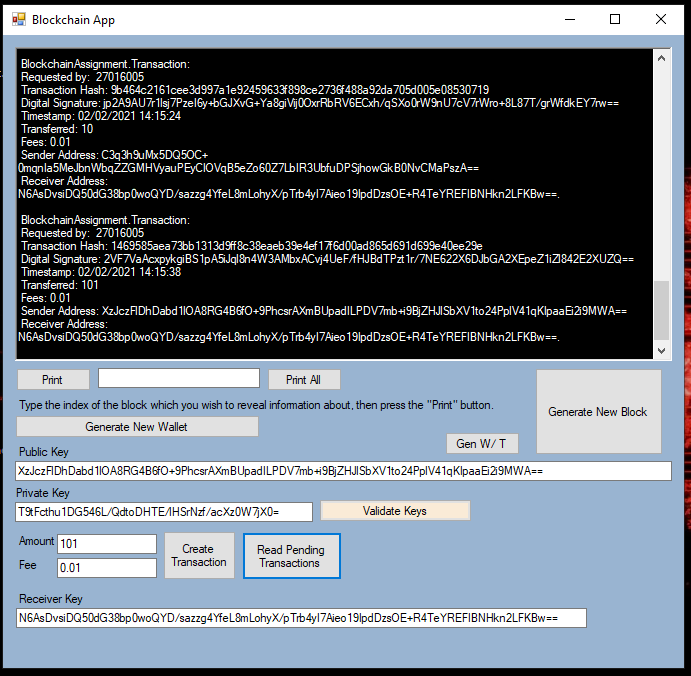


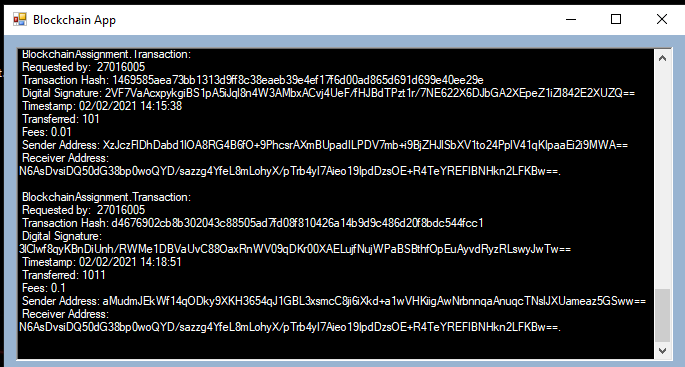




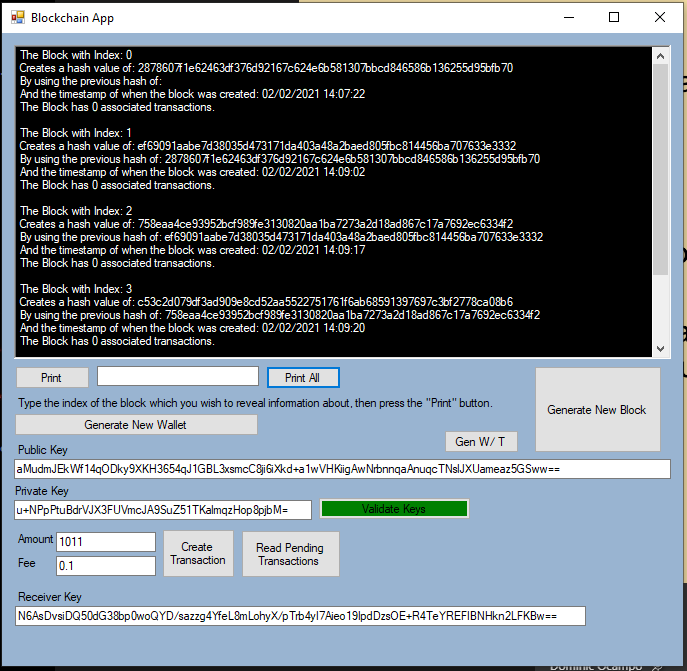


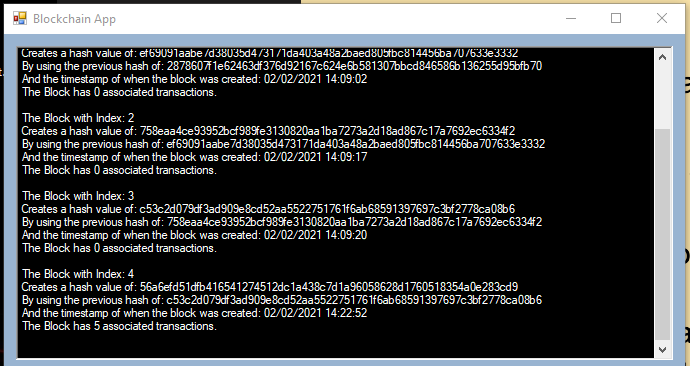


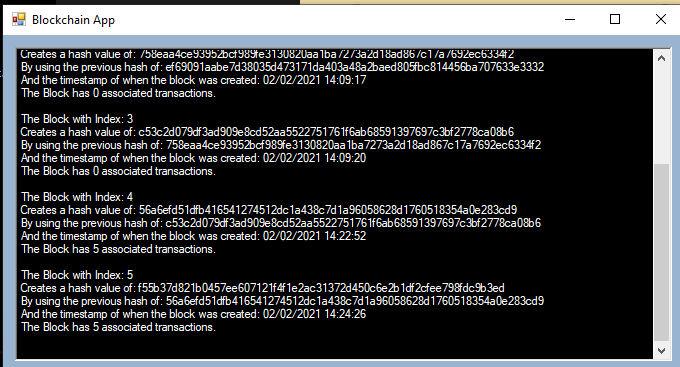


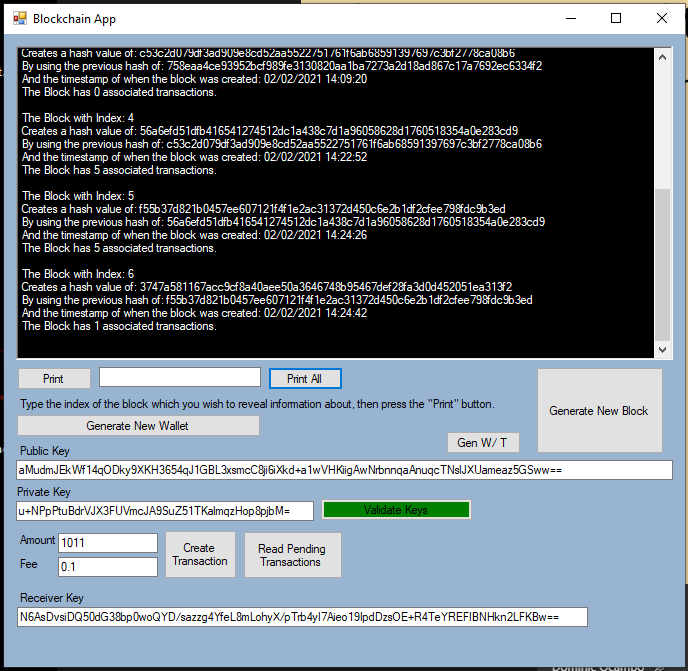


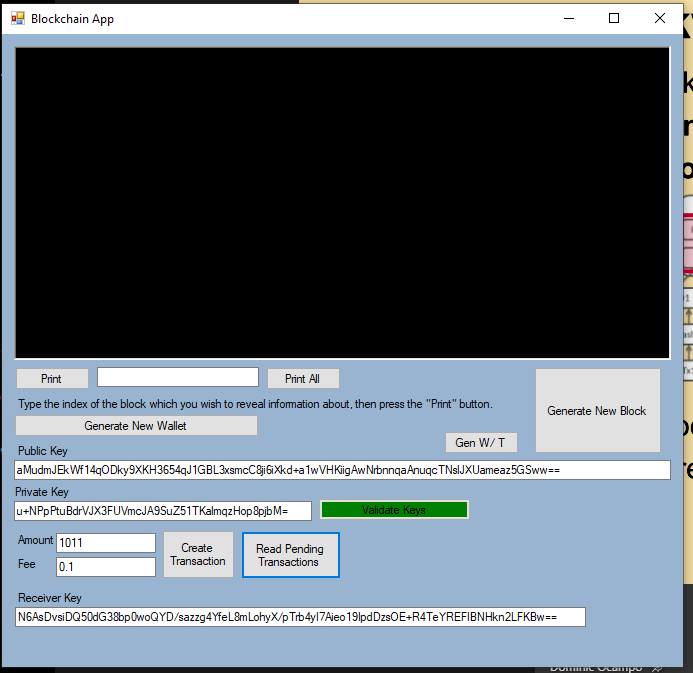
Generate a new block (with transactions)







shows +5 +5 + 1

Pending: 

1. <https://en.bitcoin.it/wiki/Difficulty> <https://en.bitcoinwiki.org/wiki/Difficulty_in_Mining>

   <https://medium.com/coinmonks/part-5-implementing-blockchain-and-cryptocurrency-with-pow-consensus-algorithm-a7f8853d23dc> [↑](#footnote-ref-0)