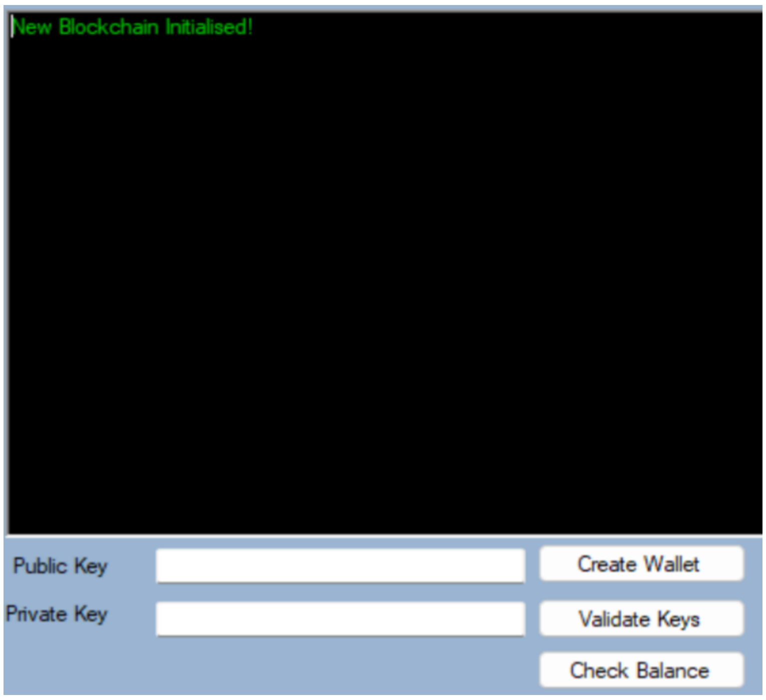
Module Code: CS3BC20
Assignment report Title: Blockchain Coursework Assignment

Student Number (e.g., 25098635): 29007149
Date (when the work completed): 06/03/2023
Actual hrs spent for the assignment: 34

https://csgitlab.reading.ac.uk/hg007149/blockchain

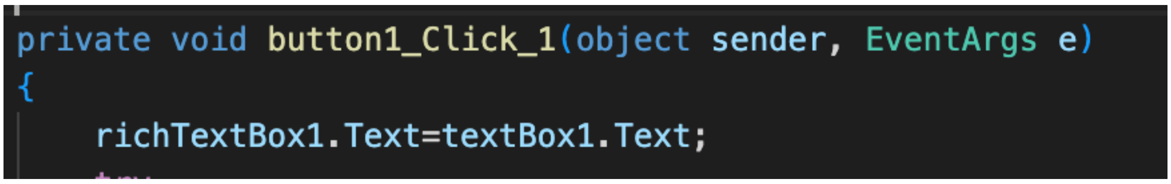
Project Setup

This project makes use of Visual Studio and its practical design elements in order to build a functional and understandable UI in which users can interact with the application. Initially the project starts as a blank screen, in which the designer is used to add some elements onto the screen which will then be attached to functions, giving them a purpose and interactivity when pressed. This can include making the program execute a task, drop down lists or inputting text.



```
New Blockchain Initialised!
```

This image provides an example of some of the design elements I was able to place on screen. As you can see buttons are placed with clear labels indicating their purpose, guiding the user on which options to select based on their needs. Each of these elements on screen is given a handler or some form of code to handle the interaction from the user, resulting in an execution.

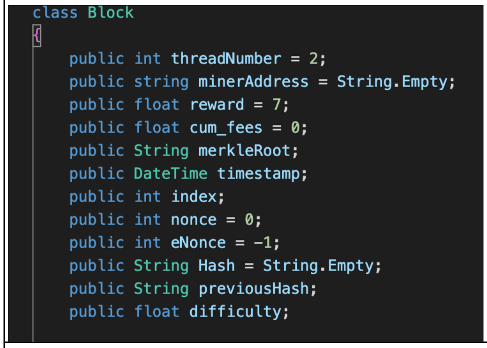


```
private void button1_Click_1(object sender, EventArgs e)
{
    richTextBox1.Text=textBox1.Text;
```

The code snippet above displays an event handler being applied to the design element. This will be called upon once the button is pressed, resulting in the relevant execution in the program. Further code and functionality is added further in the program to assign this button, and all others alike it a bespoke purpose.

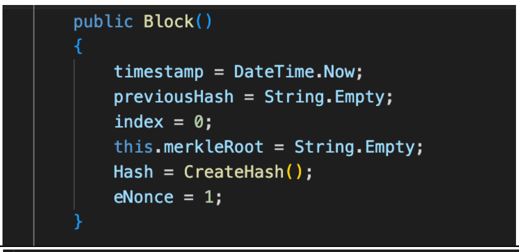
Blocks and The Blockchain

The Blockchain is a shared, immutable ledger that facilitates the process of recording transactions and tracking assets in a business network. Each block in the blockchain has a unique tag that allows them to be identified. These are known as “hash”. A hash is generated with the SHA-256 algorithm, creating a unique string made up of a combination of letters and numbers in order to create an encrypted and completely unique hash. Blocks are made up of a combination of elements, which include a timestamp of the creation of the block, the hash of the previous block, the number of the block, the nonce, the difficulty target and the code created by transactional data.



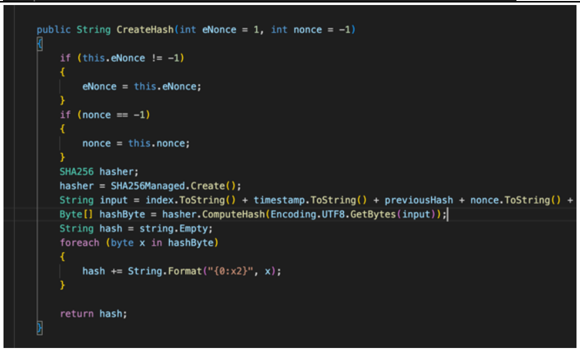
```
class Block
{
    public int threadNumber = 2;
    public string minerAddress = String.Empty;
    public float reward = 7;
    public float cum_fees = 0;
    public String merkleRoot;
    public DateTime timestamp;
    public int index;
    public int nonce = 0;
    public int eNonce = -1;
    public String Hash = String.Empty;
    public String previousHash;
    public float difficulty;
```

This code snippet displays some of the attributes referred to above being programmed for the blocks. You can see the nonce, hash, previous hash being set.



```
    public Block()
    {
        timestamp = DateTime.Now;
        previousHash = String.Empty;
        index = 0;
        this.merkleRoot = String.Empty;
        Hash = CreateHash();
        eNonce = 1;
    }
```

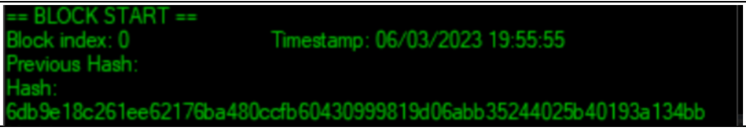
This code displays the constructor for the block class, where the index is set to 0 as it is the first block, and previous hash is set to empty due to no prior hashes existing.



```
public String CreateHash(int eNonce = 1, int nonce = -1)
{
    if (this.eNonce != -1)
    {
        eNonce = this.eNonce;
    }
    if (nonce == -1)
    {
        nonce = this.nonce;
    }
    SHA256 hasher;
    hasher = SHA256Managed.Create();
    String input = index.ToString() + timestamp.ToString() + previousHash + nonce.ToString() +
    Byte[] hashByte = hasher.ComputeHash(Encoding.UTF8.GetBytes(input));
    String hash = string.Empty;
    foreach (byte x in hashByte)
    {
        hash += String.Format("{0:x2}", x);
    }

    return hash;
}
```

This is the create hash function, which makes use of the SHA-256 algorithm in order to generate and concatenate all of the string values to make up a unique hash for the block. After being concatenated, it is processed as a byte array which is then converted into hex via a loop.

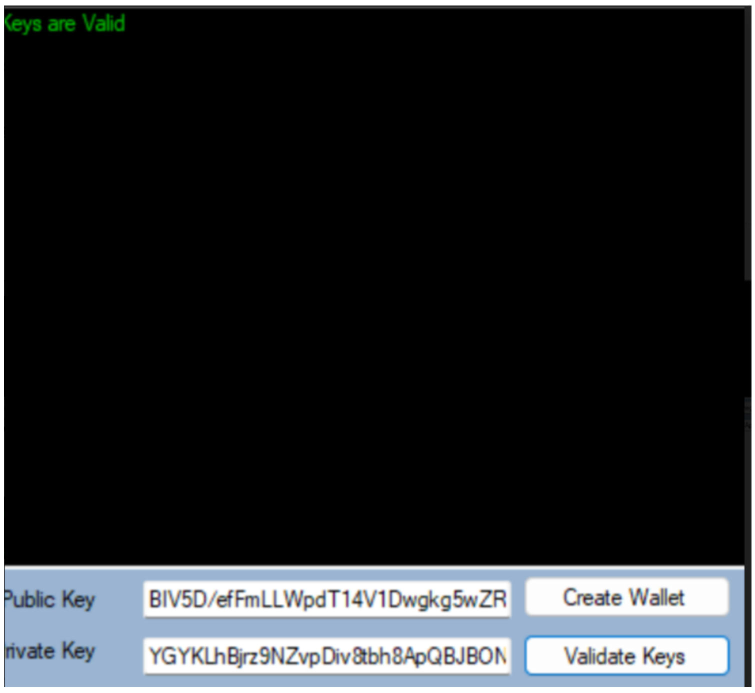


```
== BLOCK START ==
Block index: 0                    Timestamp: 06/03/2023 19:55:55
Previous Hash:
Hash:
6db9e18c261ee62176ba480ccfb60430999819d06abb35244025b40193a134bb
```

This is the output from the displayed codes execution upon being called.

Transactions and Digital Signatures

The wallets must have two keys which have a relation to each other. These are known as public keys and private keys. Public keys can be seen and accessed by all elements in the blockchain, hence it being “public”. The private key as you could imagine is the exact opposite, hidden and private from all entities, and should not be able to be deciphered, as this will typically hold access to any wallet funds, so a lack of security here compromises the entire wallets integrity. After programming this, two buttons are added to the UI and a display box which displays the public key and private key that are generated. You then have the ability to check the keys are valid by checking they have a relation to one another, ensuring the wallet functions correctly and is ready for transactions.



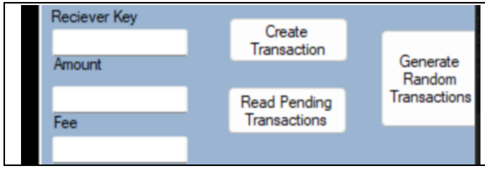
```
Keys are Valid
```

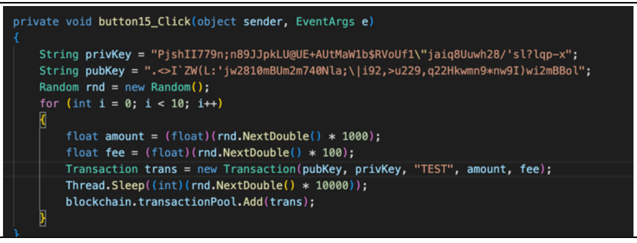
Public Key: BIV5D/efFmLLWpdT14V1Dwgkg5wZR
Private Key: YGYKLhBjrz9NZvpDiv8tbh8ApQBJBON

Setting up Transactions

Now that wallets can be generated, we need to have the ability to transfer funds between these wallets. In the transaction class we need to hold onto the hashes of the transaction with a private key, the public key from the sender, private key of the recipient, when it was made, the amount and the fee. These are all vital in allowing for transactions to be made between wallets. A hash is then generated and assigned with the sender’s private key using the constructor we make in this class.

After this we need to work on the UI. I placed buttons again using the design feature in order to allow for users to interact with the UI and generate transactions to provide this functionality.

 Reciever Key Amount Fee Create Transaction Read Pending Transactions Generate Random Transactions	These are the buttons displayed on the UI, which are each attached to event handlers in the code that provide them with functionality



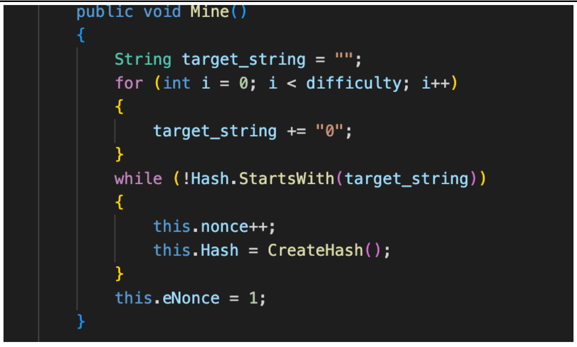
```
private void button15_Click(object sender, EventArgs e)
{
    String privKey = "PjshII779n;n89JJpkLU@UE+AUtMaW1b$RVoUf1\"jaiq8Uuwh28/'sl?lqp-x";
    String pubKey = ".<>I`ZW(L:'jw2810mBUm2m740Nla;\|i92,>u229,q22Hkwmn9*nw9I)wi2mBBol";
    Random rnd = new Random();
    for (int i = 0; i < 10; i++)
    {
        float amount = (float)(rnd.NextDouble() * 1000);
        float fee = (float)(rnd.NextDouble() * 100);
        Transaction trans = new Transaction(pubKey, privKey, "TEST", amount, fee);
        Thread.Sleep((int)(rnd.NextDouble() * 10000));
        blockchain.transactionPool.Add(trans);
    }
}
```

This code snippet displays the button functionality and the hashes.

Consensus Algorithm

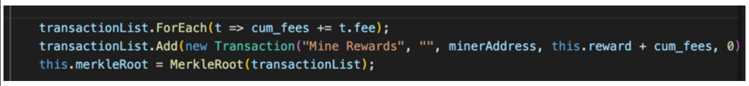
Firstly, we retrieve the last block in the blockchain. We do this as the last block will represent the current size of the blockchain.

We need to add blocks to the blockchain, this is the basic functionality. To add blocks to the blockchain, we need the ability of mining. Mining is the process of generating and adding new blocks on the blockchain by solving complex mathematical and computational problems. The consensus or proof of work algorithm defines the rules and difficulty for miners to mine on a blockchain.



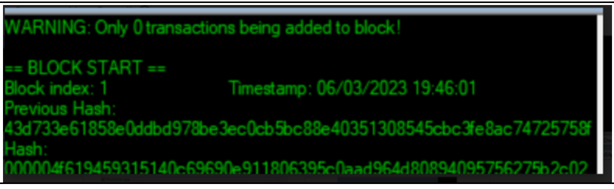
```
public void Mine()
{
    String target_string = "";
    for (int i = 0; i < difficulty; i++)
    {
        target_string += "0";
    }
    while (!Hash.StartsWith(target_string))
    {
        this.nonce++;
        this.Hash = CreateHash();
    }
    this.eNonce = 1;
}
```

This is the mine function, which makes use of the proof of work algorithm in order to find a hash that meets a specified number of zeros. The nonce is a randomly generated value that changes, making it harder to find a hash value that meets the difficulty target.



```
transactionList.ForEach(t => cum_fees += t.fee);
transactionList.Add(new Transaction("Mine Rewards", "", minerAddress, this.reward + cum_fees, 0)
this.merkleRoot = MerkleRoot(transactionList);
```

A reward system is made for mining blocks. Reward is equal to 7, and all new transactions are then added to the list, the fees and reward are then calculated as seen in the code snippet.

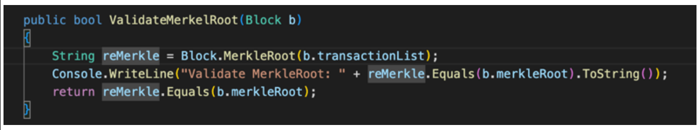
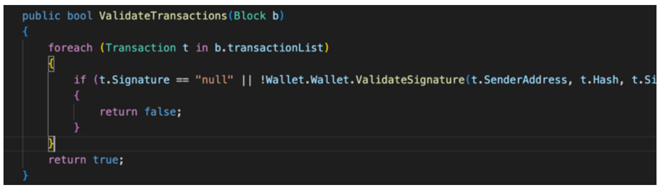


```
WARNING: Only 0 transactions being added to block!

== BLOCK START ==
Block index: 1                    Timestamp: 06/03/2023 19:46:01
Previous Hash:
43d733e61858e0ddbd978be3ec0cb5bc88e40351308545cbc3fe8ac74725758f
Hash:
000004f619459315140c69690e911806395c0aad964d80894095756275b2c02
```

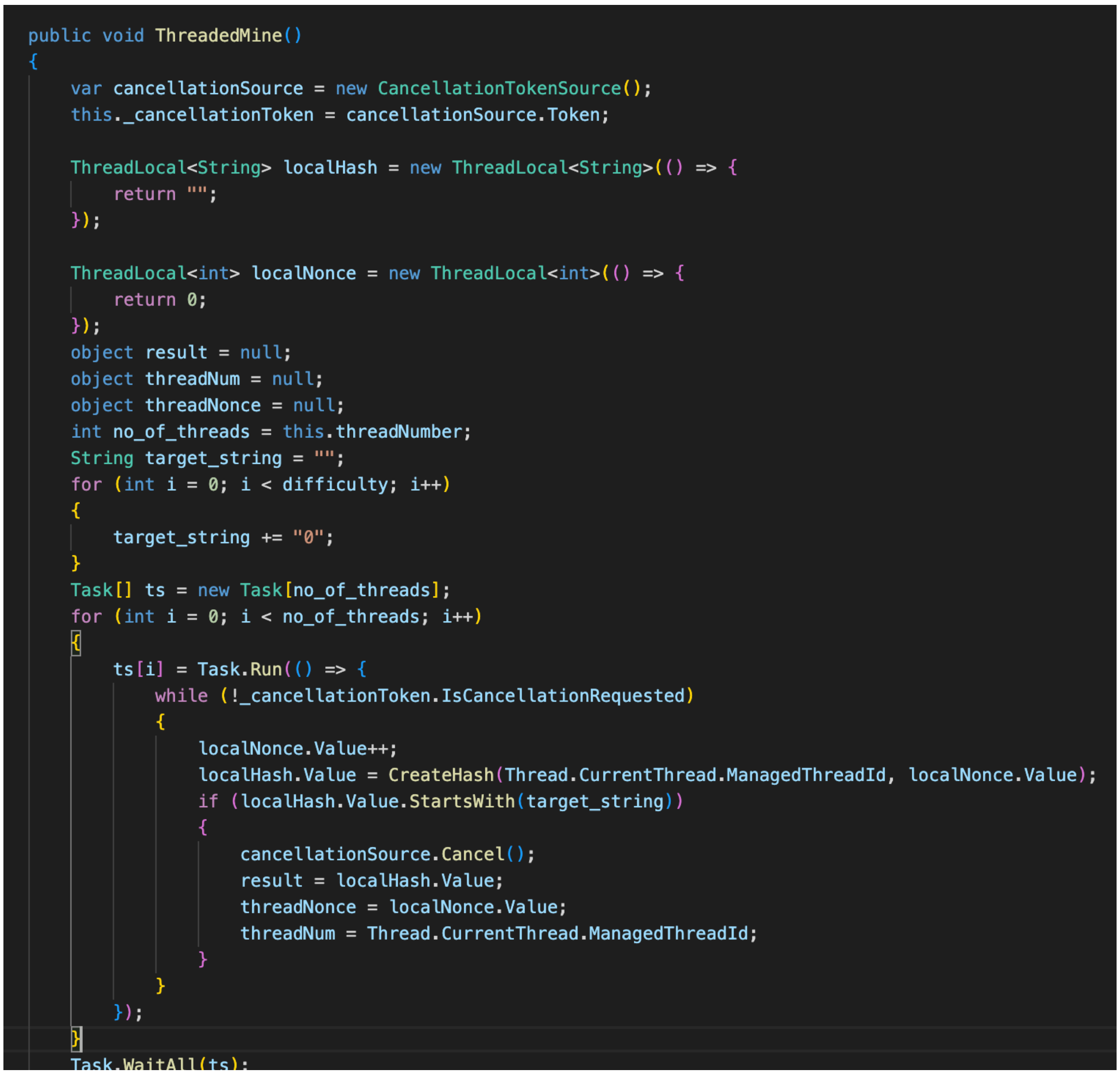
Validation

Validation is important as we need to be able to verify that the blockchain is performing tasks as intended and there is no suspicious activity on the blockchain. This comes as wallets are supposed to hold a level of integrity and protection, without validation we cannot ensure these wallets and funds are safe.

 `public bool ValidateMerkelRoot(Block b)` `{` `    String reMerkle = Block.MerkleRoot(b.transactionList);` `    Console.WriteLine("Validate MerkleRoot: " + reMerkle.Equals(b.merkleRoot).ToString());` `    return reMerkle.Equals(b.merkleRoot);` `}`	By checking the hash of the blocks matches the previous block we are able to validate the hashes are correct. If the blockchain only has one block, there is no block to compare to, so it checks the Merkle root. This means the hash of the transaction is checked and validated.
 `public bool ValidateTransactions(Block b)` `{` `    foreach (Transaction t in b.transactionList)` `    {` `        if (t.Signature == "null" || !Wallet.Wallet.ValidateSignature(t.SenderAddress, t.Hash, t.Si` `        {` `            return false;` `        }` `    }` `    return true;` `}`	We validate the transactions by looping through the transaction list to see if the wallet signature is true or false. If the signature is null or false, the system displays the signature is false, if it is true, the system feeds that back to the user. If all transactions are valid, this is returned to the user. This is validated by comparing the hashes and returning the validity of the signature.

6.1

As the proof of work algorithm does not maximise efficiency in terms of processing, we can begin to look at the process of multithreading. Multithreading is the process of using multiple processors in order to break up a large and complex task and process parts concurrently and concatenate them to form the final result faster.

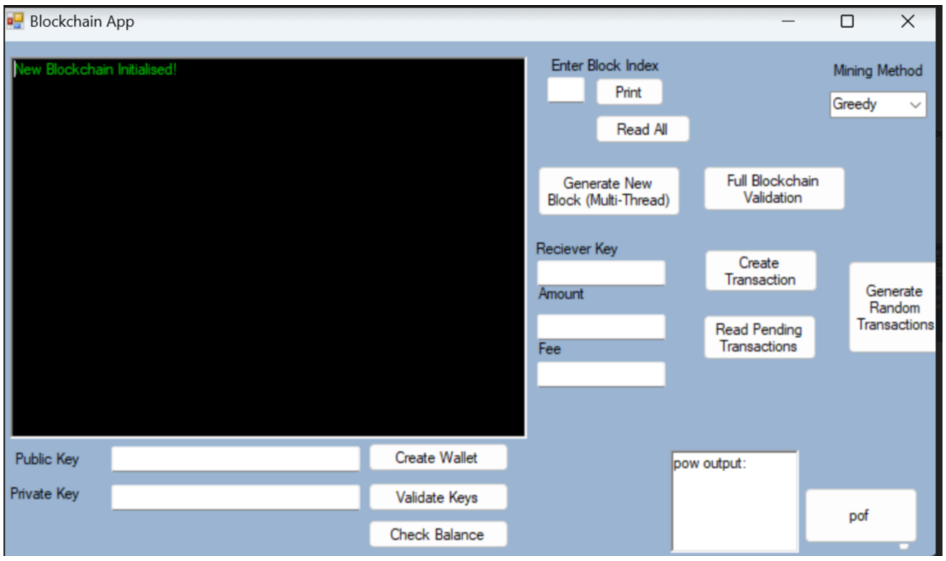


```
public void ThreadedMine()
{
    var cancellationSource = new CancellationTokenSource();
    this._cancellationToken = cancellationSource.Token;

    ThreadLocal<String> localHash = new ThreadLocal<String>(() => {
        return "";
    });

    ThreadLocal<int> localNonce = new ThreadLocal<int>(() => {
        return 0;
    });
    object result = null;
    object threadNum = null;
    object threadNonce = null;
    int no_of_threads = this.threadNumber;
    String target_string = "";
    for (int i = 0; i < difficulty; i++)
    {
        target_string += "0";
    }
    Task[] ts = new Task[no_of_threads];
    for (int i = 0; i < no_of_threads; i++)
    {
        ts[i] = Task.Run(() => {
            while (!_cancellationToken.IsCancellationRequested)
            {
                localNonce.Value++;
                localHash.Value = CreateHash(Thread.CurrentThread.ManagedThreadId, localNonce.Value);
                if (localHash.Value.StartsWith(target_string))
                {
                    cancellationSource.Cancel();
                    result = localHash.Value;
                    threadNonce = localNonce.Value;
                    threadNum = Thread.CurrentThread.ManagedThreadId;
                }
            }
        });
    }
    Task.WaitAll(ts);
```

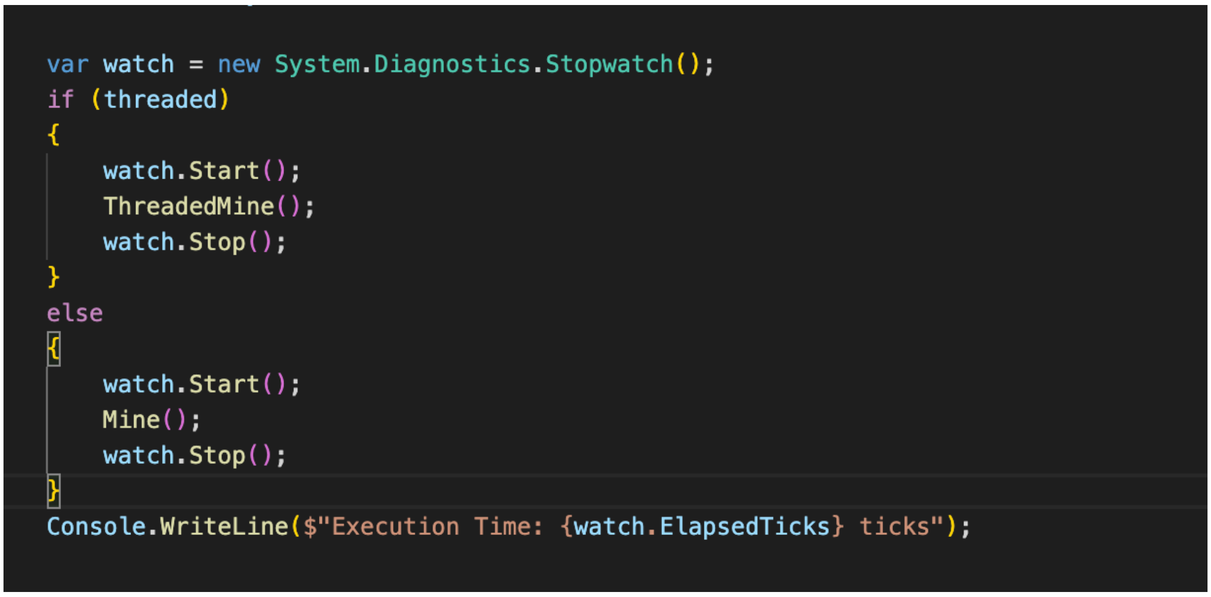
Each task in the array is looped through and utilised to create hashes until a cancellation token is called upon to halt this. If the hash meets the difficulty target the token is not used. The user is then given the option to perform a threaded mine, where they may see increased performance.



Blockchain App

New Blockchain Initialised!

Mining Method: Greedy

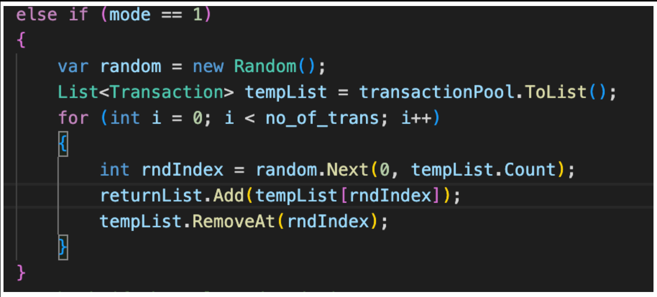


```csharp
var watch = new System.Diagnostics.Stopwatch();
if (threaded)
{
    watch.Start();
    ThreadedMine();
    watch.Stop();
}
else
{
    watch.Start();
    Mine();
    watch.Stop();
}
Console.WriteLine($"Execution Time: {watch.ElapsedTicks} ticks");
```

This code snippet displays the creation of the stopwatch tool, which is used to compare the times between threaded and non-threaded mining.

6.3

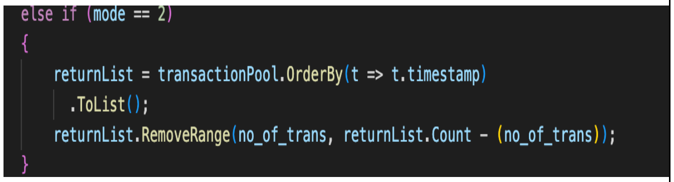
In this task, I implemented four alternative mining methods for the user to select from.



```csharp
else if (mode == 1)
{
    var random = new Random();
    List<Transaction> tempList = transactionPool.ToList();
    for (int i = 0; i < no_of_trans; i++)
    {
        int rndIndex = random.Next(0, tempList.Count);
        returnList.Add(tempList[rndIndex]);
        tempList.RemoveAt(rndIndex);
    }
}
```

Random Mining

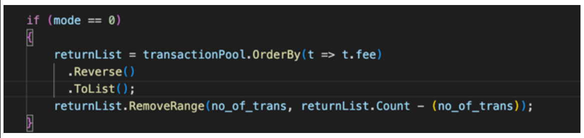
Random mining is the process of randomly selecting transactions from the transaction pool by creating an index for each transaction and then removing it from a duplicate pool ensuring this cannot be selected again. The code snippet here displays the implementation.



```csharp
else if (mode == 2)
{
    returnList = transactionPool.OrderBy(t => t.timestamp)
      .ToList();
    returnList.RemoveRange(no_of_trans, returnList.Count - (no_of_trans));
}
```

Altruistic mining

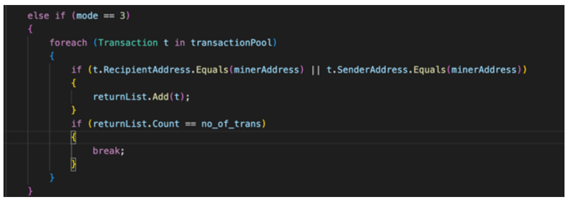
This is the process of selecting transactions solely based on the longest wait time from the pending transactions pool.



```csharp
if (mode == 0)
{
    returnList = transactionPool.OrderBy(t => t.fee)
      .Reverse()
      .ToList();
    returnList.RemoveRange(no_of_trans, returnList.Count - (no_of_trans));
}
```

Greedy Mining

This selects transactions with the highest fee first. This filters transactions from the pool from highest to lowest.



```
else if (mode == 3)
{
    foreach (Transaction t in transactionPool)
    {
        if (t.RecipientAddress.Equals(minerAddress) || t.SenderAddress.Equals(minerAddress))
        {
            returnList.Add(t);
        }
        if (returnList.Count == no_of_trans)
        {
            break;
        }
    }
}
```

Address specific Mining

This selects transactions based off address preferences to prioritise transactions from specific users, sort of like a fast track/vip list. The code selects transactions with specific addresses using loops.