**CS6362A SP21**

**Computer Security & Privacy**

**Project Topic: Digital Cash**

**By**

**Group Two**

**Anita Chidera Eluwa**

**Joel Trejo**

**Gerso Guillen**

**Luke Thurmond**

**Jose Garza**

**Department of Cybersecurity**

**School of Science, Engineering and Technology**

**St. Mary's University**

**April 26, 2021**

# Table of Contents

## Page Number

# CHAPTER ONE

# INTRODUCTION

## 1.1 BACKGROUND TO THE STUDY

Digital cash can be defined as any electronic system which allows for storage, transfer, and spending of electronic cash. These systems are mostly owned by private companies. In other words, anyone can use physical cash to buy digital credits. Then these credits can be stored in an electronic wallet and spent when required. Electronic cash is however different from mobile wallets. People can only use mobile wallets if the counterparty also uses the same wallet but this is not the case with digital cash. Digital cash is meant to be just like cash. This means that the counterparty does not need to have any systems in place to accept digital cash. There is no special hardware or software that needs to be installed to use digital cash. If such a requirement does exist, the arrangement can no longer be called digital cash. It can instead be classified as a mobile wallet.

Computerized cash acts much like genuine money, with the exception of that it's not on paper. A cryptographic money is an advanced or virtual cash intended to fill in as a medium of trade. It utilizes cryptography to secure and check exchanges and also to control the making of new units of a specific digital currency. Basically, digital forms of money are constrained sections in a database that nobody can change unless particular conditions are satisfied. Cryptocurrencies, such as Bitcoin, are a prominent example of digital currencies.

There have been numerous endeavors at making a computerized cash amid the 90s tech blast, with frameworks like Flooz, Beenz and DigiCash rising available yet definitely falling flat. There were various purposes behind their failures, for example, misrepresentation, money related issues and even contacts between organizations' workers and their supervisors. Prominently, those frameworks used a Trusted Third-Party approach, implying that the organizations behind them checked and encouraged the exchanges. Because of the disappointment of these organizations, the making of an advanced money framework was viewed as an act of futility for quite a while.

At that point, in mid 2009, a mysterious software engineer or a gathering of developers under a name Satoshi Nakamoto presented Bitcoin. Satoshi portrayed it as a 'distributed electronic framework'. It is totally decentralized, which means there are no servers included and no focal controlling specialist. The idea nearly takes after distributed systems for record sharing.

There are many reasons to exchange money for both goods or services online and there are also several banking and exchange options available for digital cash. An example of an online service which is mainly used for exchanging money is PayPal. While a third-party online service to trade currency is often the most practical solution for managing resources, there are also instances where the costs of these options have significant consequences and can even prevent the transaction. Frequently, online money exchange services are designed with specific time delays to prevent the misuse, though this processing time can also prevent the sale of a product.

Creating an option to exchange digital cash is a worthwhile project as this would allow the organization or group who owns this service control over both the privacy and policy considerations associated with it. Managing and overseeing both the records of transactions and the market specifications with digital cash would be an effective approach for any organization to further protect customers and to resolve any exchange discrepancies without needing to coordinate with additional organizations.

# CHAPTER TWO

# PROJECT SPECIFICATIONS

The goal of this project is to implement an electronic cash system where customer privacy is subjected to the policy of the institution using this application and all digital cash transactions are individually secured. To create this system there will be three significant parties or categories of users involved. The customers, merchants, and bank. Each of the three categories of users will require different accesses, rules, and permissions.

## 2.1 ELECTRONIC CASH

Electronic cash will also be referenced as e-cash. Each e-cash exchange will include additional information that will be required for the purposes of system management. Each money order will require the four following fields of information:

1. The amount of the transaction involved.
2. A uniqueness string number.
3. Identity strings which contain the identity of the customer (this information remains secret unless the customer tries to use the e-cash illicitly – more than once)
4. Bank's signature (before the customer can use the e-cash).

## 2.2 CUSTOMER

The customer accounts will need to be able to start the process for the exchange of e-cash. This will require customer accounts to exchange information with both the merchant and bank. From a customer account the e-cash order needs to be generated and assigned a different random uniqueness string number. When an e-cash transaction is started, a tracking number is provided by the bank that will be a combination of the customer account information (name, email, account number) along with the merchant information indicating where the funds are going. This exchange will require an authentication signature from the bank and merchant.

## 2.3 MERCHANT

The merchants will provide basic product or service listings that will not be included in this project. Marketing, advertising, and communicating with the customers can be done in a variety of ways. First, the merchants will be required to generate a random number or string,

that will be used by the customer to produce the transfer request. Second, the merchants will need to authenticate the bank's signature prior to receiving a transfer.

### 2.4 BANK

The bank will provide a random number that is used for tracking each money order, along with a way to ensure that no money order is used more than once. The bank will also maintain records of each of the transactions in a database, with the database the bank can prevent the transfer of funds that a customer does not have.

# CHAPTER THREE

# DESIGN

**3.1** The design of this project was centered on three (3) major parts:

- The database Architecture
- The Blockchain Architecture
- The relationship tables

The below architecture illustrates a process flow or logical analysis of the functions of digital cash.

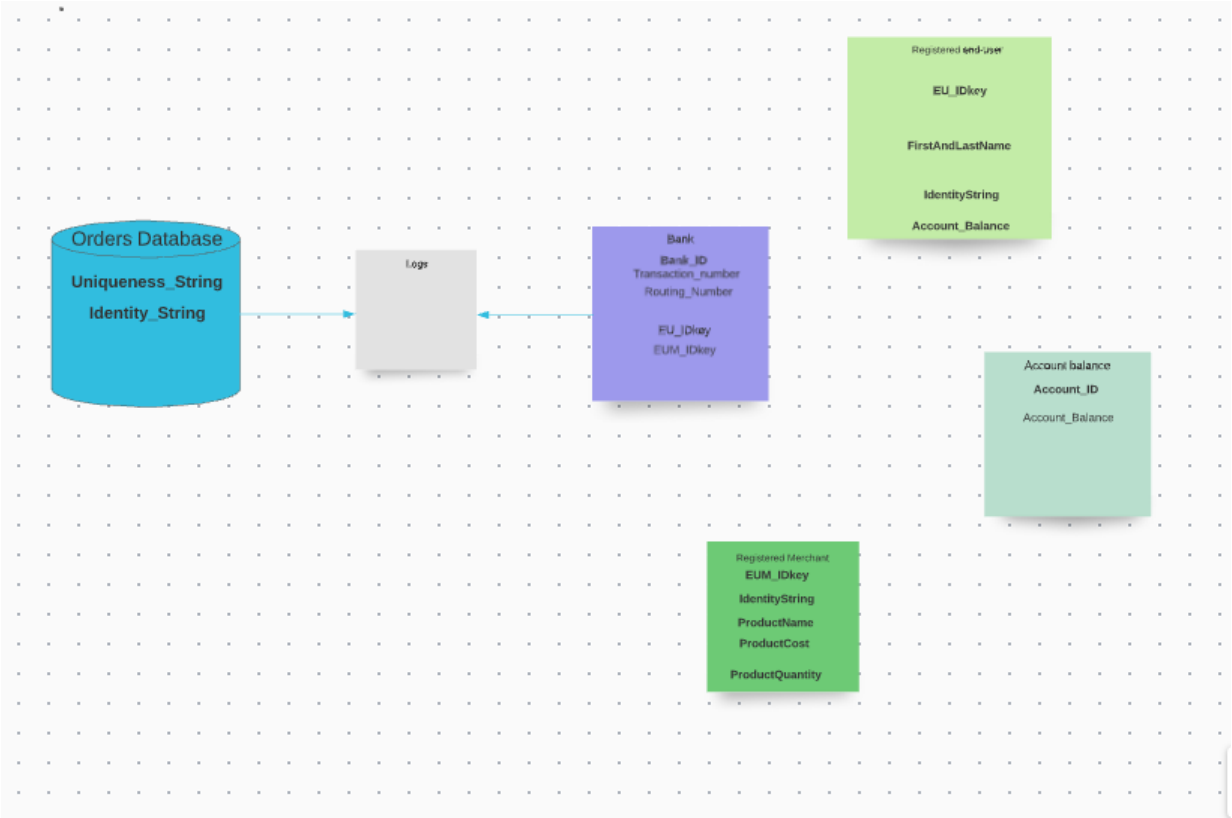



**FIGURE 3.1:** The Customer and Merchant will both access the bank with verification. The bank manages transactions and retains all information in the data base according to the organizations policies.

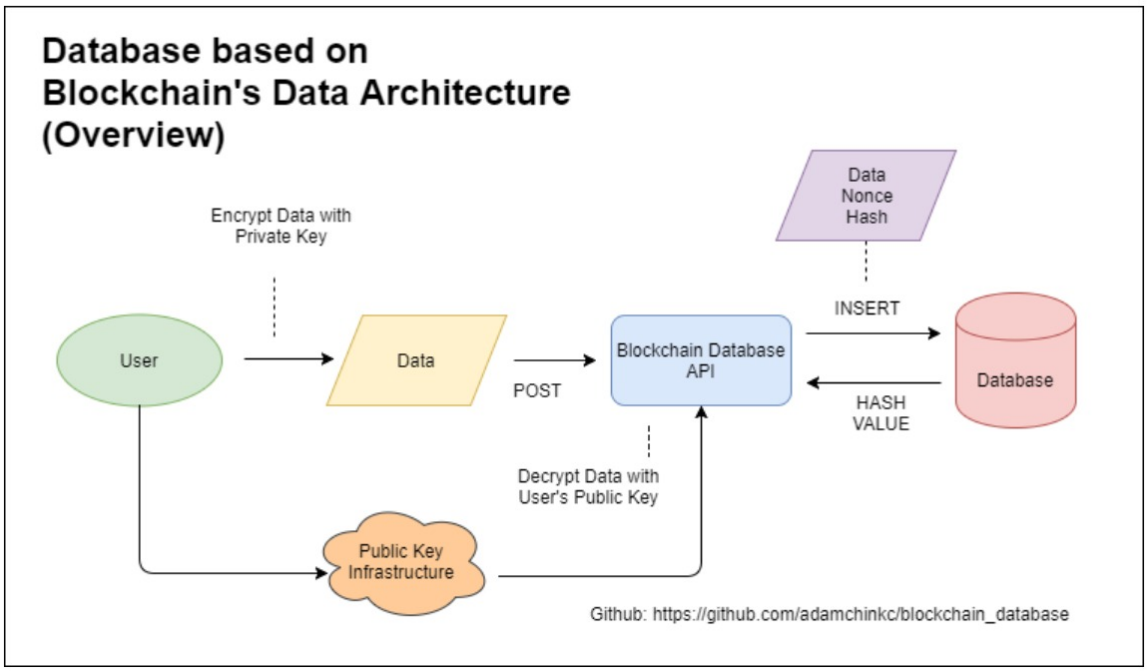



**FIGURE 3.2:** The above figure shows the Blockchain Architecture where our data was based on.

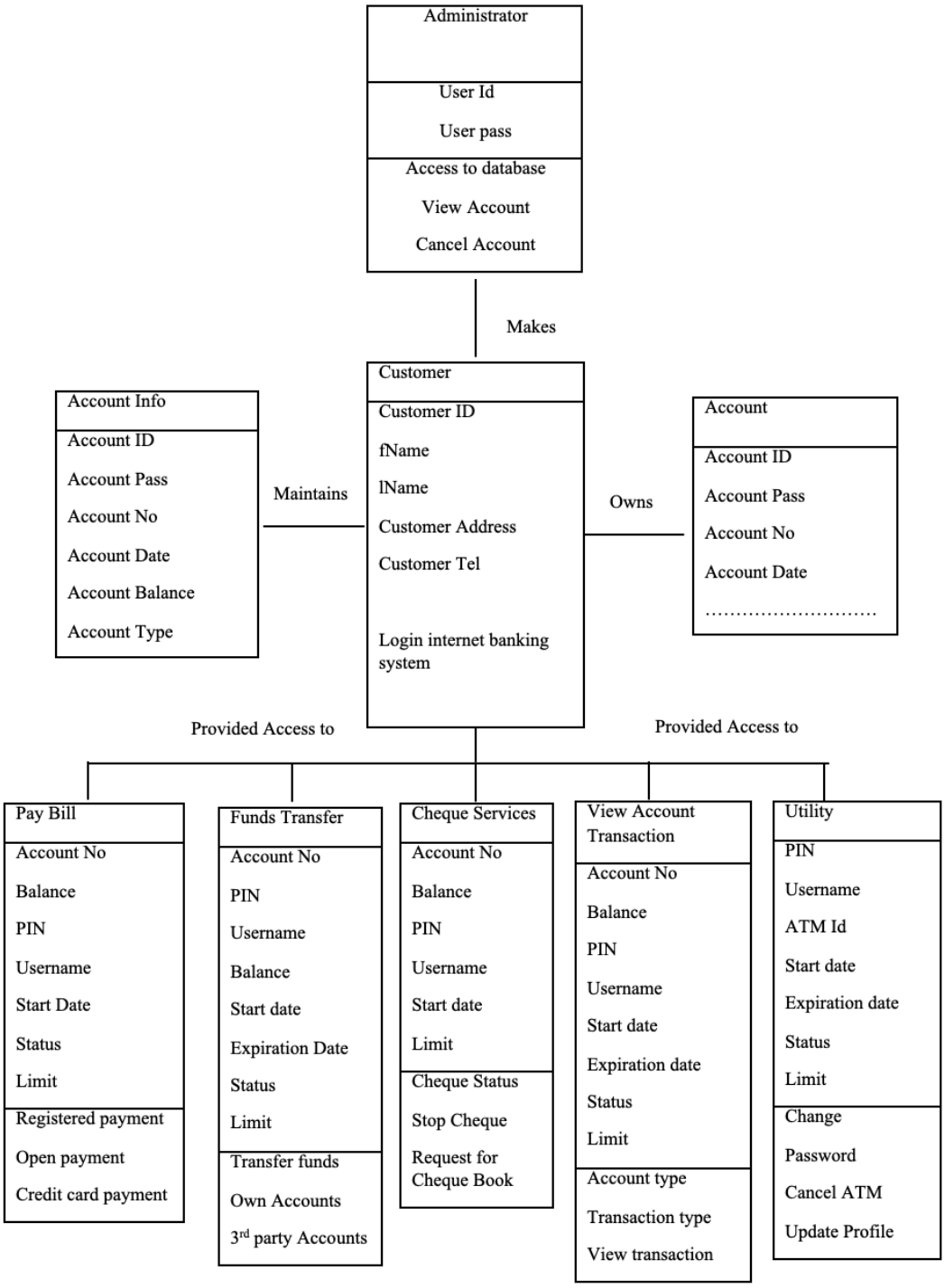



**FIGURE 3.3:** This is an example of the information that will be exchanged between each party.

# CHAPTER FOUR

# ALGORITHMS

## 4.1 ALGORITHMS

The main algorithm that our group used for our project was SSH encryption. SSH encryption uses symmetric and/or asymmetric encryption and hashing to secure the transfer of information between the customer, bank, and merchants. Using symmetric or asymmetric encryption, the bank can unmask a user if they are attempting to reuse DigiCash. With the hashing, the bank can use the hashes to compare if the current transaction is unique or not.

## 4.2 TOOLS

The main tools that our group used to develop the application were: Claris Filemaker, a virtual MacBook mini, and Microsoft SQL server. We used Claris file maker to build the front end of the application and we used Microsoft SQL server to build the back-end database used by the application. The virtual MacBook mini was used to host the server that the application would run on and provide the SSH encryption capabilities.

# CHAPTER FIVE

# USER MANUAL

The user manual of this project is divided into 4 categories as shown below.

## 5.1 ST MARY'S TX BLOCKCHAIN

Our first step was to set up our blockchain and to do that, we installed Java, JDK installer and the IDE used was Eclipse. And the class used for our blockchain was STMARYTXCHAIN. Please see below how that was implemented. Please note that all the source codes will also be provided in the code listing section of this documentation.

```
public class StmarytxChain {

        public static void main(String[] args) {

                Block genesisBlock = new Block("Hi im the first block", "0");
                System.out.println("Hash for block 1 : " + genesisBlock.hash);

                Block secondBlock = new Block("simple second block",genesisBlock.hash);
                System.out.println("Hash for block 2 : " + secondBlock.hash);

                Block thirdBlock = new Block("Another simple third block",secondBlock.hash);
                System.out.println("Hash for block 3 : " + thirdBlock.hash);


        }
}
```

- BLOCK

```
import
java.util.Date;

        public class Block {

                public String hash;
                public String previousHash;
                private String data; //our data will be a simple message.
                private long timeStamp; //as number of milliseconds since
        1/1/1970.

                //Block Constructor.
                public Block(String data,String previousHash ) {
                        this.data = data;
                        this.previousHash = previousHash;
                        this.timeStamp = new Date().getTime();
                }
```

```
}
```

- STRINGUTIL

```
import java.security.MessageDigest;

public class StringUtil {
	//Applies Sha256 to a string and returns the result.
	public static String applySha256(String input){

		try {
			MessageDigest digest = MessageDigest.getInstance("SHA-256");
			//Applies sha256 to our input,
			byte[] hash = digest.digest(input.getBytes("UTF-8"));
			StringBuffer hexString = new StringBuffer(); // This will contain hash as hexidecimal
			for (int i = 0; i < hash.length; i++) {
				String hex = Integer.toHexString(0xff & hash[i]);
				if(hex.length() == 1) hexString.append('0');
				hexString.append(hex);
			}
			return hexString.toString();
		}
		catch(Exception e) {
			throw new RuntimeException(e);
		}
	}
}
```

- CALCULATEHASH

```
public String calculateHash()
{
		String calculatedhash = StringUtil.applySha256(
			previousHash +
			Long.toString(timeStamp) +
			data
			);
		return calculatedhash;
	}
```

- BLOCK (CONTRUCTOR)

```
public Block(String data,String previousHash ) {
        this.data = data;
        this.previousHash = previousHash;
        this.timeStamp = new Date().getTime();
        this.hash = calculateHash(); //Making sure
    we do this after we set the other values.
        }
```

- STORING OUR BLOCKS IN AN ARRAYLIST AND IMPORTING WITH GSON

```
import
java.util.ArrayList;
        import com.google.gson.GsonBuilder;

        public class StmarytxChain {

            public static ArrayList<Block> blockchain = new
        ArrayList<Block>();

            public static void main(String[] args) {
                //add our blocks to the blockchain ArrayList:
                blockchain.add(new Block("Hi im the first block",
        "0"));
                blockchain.add(new Block("Yo im the second
        block",blockchain.get(blockchain.size()-1).hash));
                blockchain.add(new Block("Hey im the third
        block",blockchain.get(blockchain.size()-1).hash));

                String blockchainJson = new
        GsonBuilder().setPrettyPrinting().create().toJson(blockchain);

                System.out.println(blockchainJson);
            }

        }
```

**FIGURE 5.1**: The above images show the implementation process on how our blockchain was set up.

After this, the next step we implemented was to check the integrity of our blockchain. To do that, please see below.

## 5.12 CHECKING OUR BLOCKCHAIN INTEGRITY

By creating an ischainvalid() Boolean method in the STMARYTXCHAIN class, we can loop through all blocks and compare the hash. Please see how this was implemented.

```
public static Boolean isChainValid() {
        Block currentBlock;
        Block previousBlock;

        //loop through blockchain to check hashes:
        for(int i=1; i < blockchain.size(); i++) {
                currentBlock = blockchain.get(i);
                previousBlock = blockchain.get(i-1);
                //compare registered hash and calculated hash:
                if(!currentBlock.hash.equals(currentBlock.calculateHash()) ){
                        System.out.println("Current Hashes not equal");

                        return false;
                }
                //compare previous hash and registered previous hash
                if(!previousBlock.hash.equals(currentBlock.previousHash) ) {
                        System.out.println("Previous Hashes not equal");
                        return false;
                }
        }
        return true;
}
```

**FIGURE 5.12:** The above image shows the implementation process on how to check the integrity of our blockchain.

## 5.13 MINING BLOCKS

The next thing was to mine our blockchain. Our chainblock is easily invalidated because it so small but we know that once the blockchain is much longer it won't be as easy to break the chain. Please see below.

```
import
java.util.Date;

                public class Block {

                        public String hash;
                        public String previousHash;
                        private String data; //our data will be a simple message.
                        private long timeStamp; //as number of milliseconds since
                1/1/1970.
                        private int nonce;

                        //Block Constructor.
                        public Block(String data,String previousHash ) {
                                this.data = data;
                                this.previousHash = previousHash;
                                this.timeStamp = new Date().getTime();

                                this.hash = calculateHash(); //Making sure we do this
                after we set the other values.
                        }

                        //Calculate new hash based on blocks contents
                        public String calculateHash() {
                                String calculatedhash = StringUtil.applySha256(
                                                previousHash +
                                                Long.toString(timeStamp) +
                                                Integer.toString(nonce) +
                                                data
                                                );
                                return calculatedhash;
                        }

                        public void mineBlock(int difficulty) {
                                String target = new String(new
                char[difficulty]).replace('\0', '0'); //Create a string with difficulty
                * "0"
                                while(!hash.substring( 0, difficulty).equals(target)) {
                                        nonce ++;
                                        hash = calculateHash();
                                }
                                System.out.println("Block Mined!!! : " + hash);
                        }
```

**FIGURE 5.13:** Mining our Blockchain

The next step is to update our blockchain to trigger MineBlock() for each block as well as have the Ischainvalid() also check to see if blockchain is solved. Please see below.

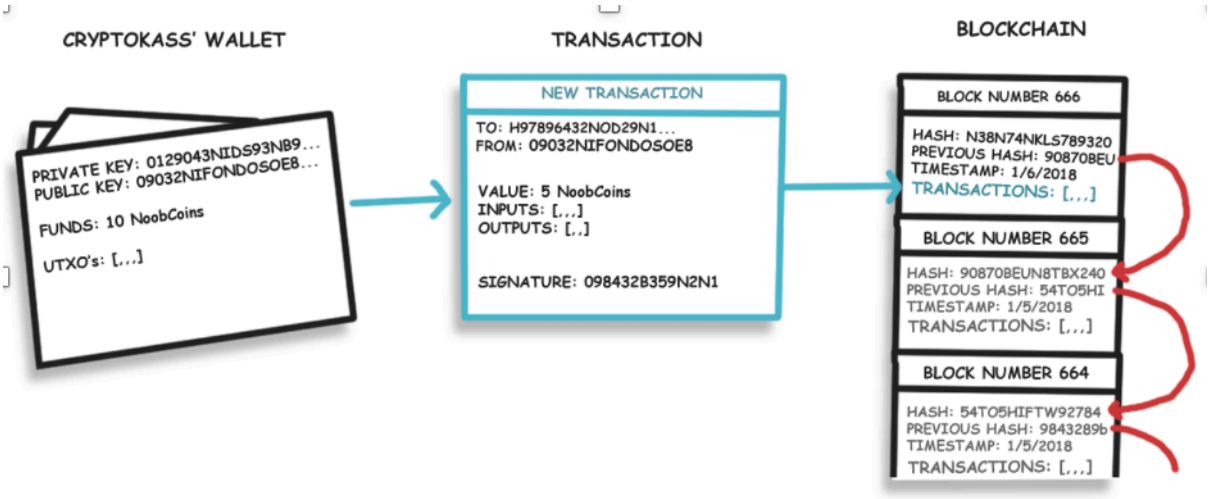
```
import
java.util.ArrayList;
                    import com.google.gson.GsonBuilder;

                    public class StmarytxChain {

                            public static ArrayList<Block> blockchain = new
                    ArrayList<Block>();
                            public static int difficulty = 5;

                            public static void main(String[] args) {
                                    //add our blocks to the blockchain ArrayList:

                                    blockchain.add(new Block("Hi im the first block",
                    "0"));
                                    System.out.println("Trying to Mine block 1... ");
                                    blockchain.get(0).mineBlock(difficulty);

                                    blockchain.add(new Block("Yo im the second
                    block",blockchain.get(blockchain.size()-1).hash));
                                    System.out.println("Trying to Mine block 2... ");
                                    blockchain.get(1).mineBlock(difficulty);

                                    blockchain.add(new Block("Hey im the third
                    block",blockchain.get(blockchain.size()-1).hash));
                                    System.out.println("Trying to Mine block 3... ");
                                    blockchain.get(2).mineBlock(difficulty);

                                    System.out.println("\nBlockchain is Valid: " +
                    isChainValid());

                                    String blockchainJson = new
                    GsonBuilder().setPrettyPrinting().create().toJson(blockchain);
                                    System.out.println("\nThe block chain: ");
                                    System.out.println(blockchainJson);
                            }
```

```
	public static Boolean isChainValid() {
		Block currentBlock;
		Block previousBlock;
		String hashTarget = new String(new
char[difficulty]).replace('\0', '0');

		//loop through blockchain to check hashes:
		for(int i=1; i < blockchain.size(); i++) {
			currentBlock = blockchain.get(i);
			previousBlock = blockchain.get(i-1);
			//compare registered hash and calculated
hash:

	if(!currentBlock.hash.equals(currentBlock.calculateHash())
){
				System.out.println("Current Hashes not
equal");
				return false;
			}
			//compare previous hash and registered
previous hash

	if(!previousBlock.hash.equals(currentBlock.previousHash) ) {
				System.out.println("Previous Hashes
not equal");
				return false;
			}
			//check if hash is solved
			if(!currentBlock.hash.substring( 0,
difficulty).equals(hashTarget)) {
				System.out.println("This block hasn't
been mined");
				return false;
			}
		}
		return true;
	}
}
```

**FIGURE 5.14:** Updating our Blockchain

## 5.2 OUR E-CASH BANK APP WALLET

### 5.21 PREPARING OUR WALLET




We first created a wallet class as shown below.

```
package Stmarytxchain;
import java.security.*;

public class Wallet {
	public PrivateKey privateKey;
	public PublicKey publicKey;
}
```

**FIGURE 5.21:** Creating a wallet class

## 5.22 USERS

In our e cash banking app, we allow for anyone to be able to sign up with their mobile number and a password. The hash between the two creates the public key for the user. The private key is two fields that only admin know which two fields are being used to generate this and is stored in the server hidden from anyone. Users can sign up by either going to the sign-up form hosted on the file server that is then served via webdirect or can be manually added by an admin.

## BY GOING TO FILE>MANAGE>SECURITY

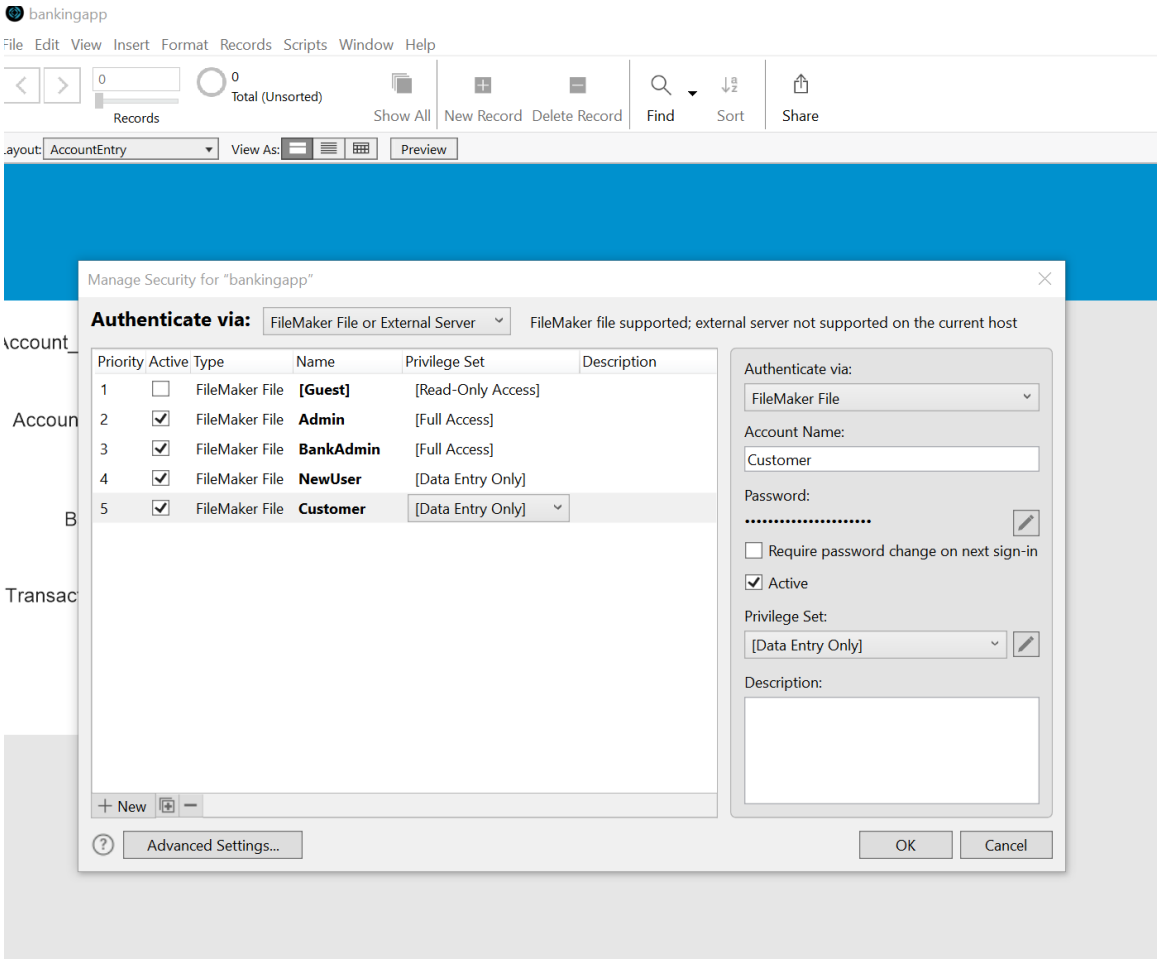



**FIGURE 5.22:** Manage Security User Interface of the Application

We generated our private and public keys in a KeyPair and appended that to our Wallet class:

```
Package
Stmarytxchain;
                import java.security.*;

                public class Wallet {

                        public PrivateKey privateKey;
                        public PublicKey publicKey;

                        public Wallet(){
                                generateKeyPair();
                        }

                        public void generateKeyPair() {
                                try {
                                        KeyPairGenerator keyGen =
                KeyPairGenerator.getInstance("ECDSA","BC");
                                        SecureRandom random =
                SecureRandom.getInstance("SHA1PRNG");
                                        ECGenParameterSpec ecSpec = new
                ECGenParameterSpec("prime192v1");
                                        // Initialize the key generator and generate a
                KeyPair
                                        keyGen.initialize(ecSpec, random);   //256 bytes
                provides an acceptable security level
                                        KeyPair keyPair = keyGen.generateKeyPair();
                                        // Set the public and private keys from the keyPair
                                        privateKey = keyPair.getPrivate();
                                        publicKey = keyPair.getPublic();
                                }catch(Exception e) {
                                        throw new RuntimeException(e);
                                }
                        }

                }
```

**FIGURE 5.23:** Private and Public keys generation

**Note:** This all gets generated in the FileServer using Java and stored in the FileMaker server which only knows to check hash and not personal information as to not be bias.

## 5.3 HOW TRANSACTIONS GET SIGNED

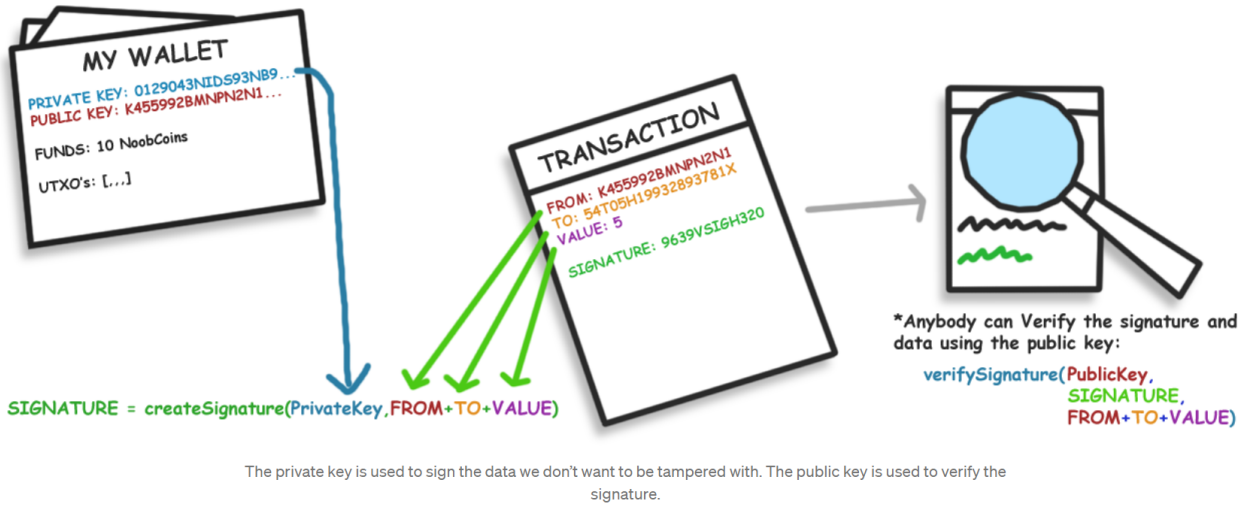



**FIGURE 5.3:** How transactions get signed.

### 5.31 TRANSACTIONS

Each transaction will have a public key of sender, public key of receiver, amount of funds to get transferred, inputs to validate, output that shows addresses received in transactions, and a cryptographic signature.

This class was created with the below:

```
import
java.security.*;
                import java.util.ArrayList;

                public class Transaction {

                        public String transactionId; // this is also the hash of the
                transaction.
                        public PublicKey sender; // senders address/public key.
                        public PublicKey reciepient; // Recipients address/public key.
                        public float value;
                        public byte[] signature; // this is to prevent anybody else from
                spending funds in our wallet.

                        public ArrayList<TransactionInput> inputs = new
                ArrayList<TransactionInput>();
                        public ArrayList<TransactionOutput> outputs = new
                ArrayList<TransactionOutput>();

                        private static int sequence = 0; // a rough count of how many
                transactions have been generated.

                        // Constructor:
                        public Transaction(PublicKey from, PublicKey to, float value,
                ArrayList<TransactionInput> inputs) {
                                this.sender = from;
                                this.reciepient = to;
                                this.value = value;
                                this.inputs = inputs;
                        }

                        // This Calculates the transaction hash (which will be used as
                its Id)
                        private String calulateHash() {
                                sequence++; //increase the sequence to avoid 2 identical
                transactions having the same hash
                                return StringUtil.applySha256(
                                                StringUtil.getStringFromKey(sender) +
                                                StringUtil.getStringFromKey(reciepient) +
                                                Float.toString(value) + sequence
                                                );
                        }
                }
```

**FIGURE 5.31:** How the class was created.

## 5.32 WHY SIGNATURES

Signatures perform two very important tasks on our blockchain: They allow only the owner to spend their coins, secondly, they prevent others from tampering with their submitted transaction before a new block is mined (at the point of entry). Private key is used to sign the data.

## 5.33 HELPER FUNCTIONS

We also created some helper functions for example the StringUtil class:

```
//Applies ECDSA Signature and returns the result ( as bytes ).
		public static byte[] applyECDSASig(PrivateKey privateKey, String
input) {
		Signature dsa;
		byte[] output = new byte[0];
		try {
			dsa = Signature.getInstance("ECDSA", "BC");
			dsa.initSign(privateKey);
			byte[] strByte = input.getBytes();
			dsa.update(strByte);
			byte[] realSig = dsa.sign();
			output = realSig;
		} catch (Exception e) {
			throw new RuntimeException(e);
		}
		return output;
	}

	//Verifies a String signature
	public static boolean verifyECDSASig(PublicKey publicKey, String data,
byte[] signature) {
		try {
			Signature ecdsaVerify = Signature.getInstance("ECDSA",
"BC");
			ecdsaVerify.initVerify(publicKey);
			ecdsaVerify.update(data.getBytes());
			return ecdsaVerify.verify(signature);
		}catch(Exception e) {
			throw new RuntimeException(e);
		}
	}

	public static String getStringFromKey(Key key) {
		return Base64.getEncoder().encodeToString(key.getEncoded());
	}
```

**FIGURE 5.33:** Helper Functions

As well as we created a generateSignature() and verifySignature() like in this example:

```
//Signs all the data we dont wish to be tampered with.
public void generateSignature(PrivateKey privateKey) {
	String data = StringUtil.getStringFromKey(sender) +
StringUtil.getStringFromKey(reciepient) + Float.toString(value)	;
	signature = StringUtil.applyECDSASig(privateKey,data);
}
//Verifies the data we signed hasnt been tampered with
public boolean verifiySignature() {
	String data = StringUtil.getStringFromKey(sender) +
StringUtil.getStringFromKey(reciepient) + Float.toString(value)	;
	return StringUtil.verifyECDSASig(sender, data, signature);
}
```

We then test this all with new variables and replace our content of our main method:

```
import
java.security.Securi
ty;
                    import java.util.ArrayList;
                    import java.util.Base64;
                    import com.google.gson.GsonBuilder;

                    public class StmarytxChain {

                    	public static ArrayList<Block> blockchain = new
                    ArrayList<Block>();
                    	public static int difficulty = 5;
                    	public static Wallet walletA;
                    	public static Wallet walletB;

                    	public static void main(String[] args) {
                    		//Setup Bouncey castle as a Security Provider
                    		Security.addProvider(new
                    org.bouncycastle.jce.provider.BouncyCastleProvider());
                    		//Create the new wallets
                    		walletA = new Wallet();
                    		walletB = new Wallet();
                    		//Test public and private keys
                    		System.out.println("Private and public keys:");
```

```
        System.out.println(StringUtil.getStringFromKey(walletA.priv
ateKey));

        System.out.println(StringUtil.getStringFromKey(walletA.publ
icKey));
                //Create a test transaction from WalletA to walletB
                Transaction transaction = new
Transaction(walletA.publicKey, walletB.publicKey, 5, null);
                transaction.generateSignature(walletA.privateKey);
                //Verify the signature works and verify it from the
public key
                System.out.println("Is signature verified");
                System.out.println(transaction.verifiySignature());

        }
```

**FIGURE 5.33:** Generate Signature and Verify Signature

### 5.34 WHO OWNS STMARYSCOINS?

Your wallets balance is the sum of all the unspent transactions outputs addressed to you.

So, we created a TransactionInput Class:

```
public class TransactionInput {
	public String transactionOutputId; //Reference to TransactionOutputs ->
transactionId
	public TransactionOutput UTXO; //Contains the Unspent transaction output

	public TransactionInput(String transactionOutputId) {
		this.transactionOutputId = transactionOutputId;
	}
}
```

As well as a TransactionOutput Class:

```
import java.security.PublicKey;

public class TransactionOutput {
	public String id;
	public PublicKey reciepient; //also known as the new owner of these coins.
	public float value; //the amount of coins they own
	public String parentTransactionId; //the id of the transaction this output was
created in

	//Constructor
	public TransactionOutput(PublicKey reciepient, float value, String
parentTransactionId) {
		this.reciepient = reciepient;
		this.value = value;
		this.parentTransactionId = parentTransactionId;
		this.id =
StringUtil.applySha256(StringUtil.getStringFromKey(reciepient)+Float.toString(value)+
parentTransactionId);
	}
```

```
	//Check if coin belongs to you
	public boolean isMine(PublicKey publicKey) {
		return (publicKey == recipient);
	}

}
```

We also added a collection of all UTXOs:

```
public class StmarytxChain {

	public static ArrayList<Block> blockchain = new ArrayList<Block>();
	public static HashMap<String,TransactionOutputs> UTXOs = new
HashMap<String,TransactionOutputs>(); //list of all unspent transactions.
	public static int difficulty = 5;
	public static Wallet walletA;
	public static Wallet walletB;

	public static void main(String[] args) {
```

Then we put all this together with a boolean method in our transaction:

```
//Returns
true if new
transaction
could be
created.

	public boolean processTransaction() {

		if(verifiySignature() == false) {
			System.out.println("#Transaction Signature failed to
	verify");
			return false;
		}

		//gather transaction inputs (Make sure they are unspent):
		for(TransactionInput i : inputs) {
			i.UTXO =
	StmarytxChain.UTXOs.get(i.transactionOutputId);
```

```
		}

		//check if transaction is valid:
		if(getInputsValue() < StmarytxChain.minimumTransaction) {
			System.out.println("#Transaction Inputs to small: " +
getInputsValue());
			return false;
		}

		//generate transaction outputs:
		float leftOver = getInputsValue() - value; //get value of
inputs then the left over change:
		transactionId = calulateHash();
		outputs.add(new TransactionOutput( this.reciepient,
value,transactionId)); //send value to recipient
		outputs.add(new TransactionOutput( this.sender,
leftOver,transactionId)); //send the left over 'change' back to sender


		//add outputs to Unspent list
		for(TransactionOutput o : outputs) {
			StmarytxChain.UTXOs.put(o.id , o);
		}

		//remove transaction inputs from UTXO lists as spent:
		for(TransactionInput i : inputs) {
			if(i.UTXO == null) continue; //if Transaction can't be
found skip it
			StmarytxChain.UTXOs.remove(i.UTXO.id);
		}

		return true;
	}

//returns sum of inputs(UTXOs) values
	public float getInputsValue() {
		float total = 0;
		for(TransactionInput i : inputs) {
			if(i.UTXO == null) continue; //if Transaction can't be
found skip it
			total += i.UTXO.value;
		}
		return total;
```

```
	}

//returns sum of outputs:
	public float getOutputsValue() {
		float total = 0;
		for(TransactionOutput o : outputs) {
			total += o.value;
		}
		return total;
	}
```

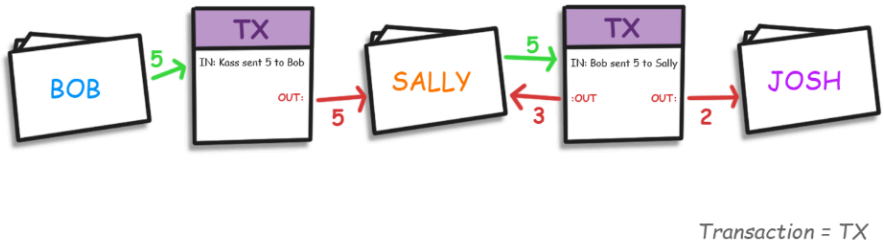



**FIGURE 5.34:** The above images show how we created the transaction input class.

We then updated our wallet to:

```
import
java.security.*;
		import java.security.spec.ECGenParameterSpec;
		import java.util.ArrayList;
		import java.util.HashMap;
		import java.util.Map;

		public class Wallet {

			public PrivateKey privateKey;
			public PublicKey publicKey;
```

```
	public HashMap<String,TransactionOutput> UTXOs = new
HashMap<String,TransactionOutput>(); //only UTXOs owned by this wallet.

	public Wallet() {...

	public void generateKeyPair() {...

  //returns balance and stores the UTXO's owned by this wallet in
this.UTXOs
	public float getBalance() {
		float total = 0;
	 for (Map.Entry<String, TransactionOutput> item:
NoobChain.UTXOs.entrySet()){
		TransactionOutput UTXO = item.getValue();
	    if(UTXO.isMine(publicKey)) { //if output belongs to me ( if
coins belong to me )
		UTXOs.put(UTXO.id,UTXO); //add it to our list of unspent
transactions.
		total += UTXO.value ;
	    }
	 }
		return total;
	}
	//Generates and returns a new transaction from this wallet.
	public Transaction sendFunds(PublicKey _recipient,float value )
{
		if(getBalance() < value) { //gather balance and check
funds.
			System.out.println("#Not Enough funds to send
transaction. Transaction Discarded.");
			return null;
		}
    //create array list of inputs
		ArrayList<TransactionInput> inputs = new
ArrayList<TransactionInput>();

		float total = 0;
		for (Map.Entry<String, TransactionOutput> item:
UTXOs.entrySet()){
			TransactionOutput UTXO = item.getValue();
			total += UTXO.value;
			inputs.add(new TransactionInput(UTXO.id));
			if(total > value) break;
```

```
                }

                Transaction newTransaction = new Transaction(publicKey,
        _recipient , value, inputs);
                newTransaction.generateSignature(privateKey);

                for(TransactionInput input: inputs){
                        UTXOs.remove(input.transactionOutputId);
                }
                return newTransaction;
        }

}
```

**FIGURE 5.35:** Updated wallet

Then we added transactions to our block by generating the merkleroot in StringUtils:

```
//Tacks in
array of
transactions
and returns
a merkle
root.
        public static String getMerkleRoot(ArrayList<Transaction> transactions) {
                int count = transactions.size();
                ArrayList<String> previousTreeLayer = new
        ArrayList<String>();
                for(Transaction transaction : transactions) {
                        previousTreeLayer.add(transaction.transactionId);
                }
                ArrayList<String> treeLayer = previousTreeLayer;
                while(count > 1) {
                        treeLayer = new ArrayList<String>();
                        for(int i=1; i < previousTreeLayer.size(); i++) {

                treeLayer.add(applySha256(previousTreeLayer.get(i-1) +
        previousTreeLayer.get(i)));
                        }
                        count = treeLayer.size();
                        previousTreeLayer = treeLayer;
                }
                String merkleRoot = (treeLayer.size() == 1) ?
        treeLayer.get(0) : "";
                return merkleRoot;
        }
```

**FIGURE 5.36:** Transactions added to our block.

Then we finally implemented our changes to our Block Class:

```
import
java.util.ArrayList;
                    import java.util.Date;

                    public class Block {

                            public String hash;
                            public String previousHash;
                            public String merkleRoot;
                            public ArrayList<Transaction> transactions = new ArrayList<Transaction>();
                    //our data will be a simple message.
                            public long timeStamp; //as number of milliseconds since 1/1/1970.
                            public int nonce;

                            //Block Constructor.
                            public Block(String previousHash ) {
                                    this.previousHash = previousHash;
                                    this.timeStamp = new Date().getTime();

                                    this.hash = calculateHash(); //Making sure we do this after we set the
                    other values.
                            }

                            //Calculate new hash based on blocks contents
                            public String calculateHash() {
                                    String calculatedhash = StringUtil.applySha256(
                                                    previousHash +
                                                    Long.toString(timeStamp) +
                                                    Integer.toString(nonce) +
                                                    merkleRoot
                                                    );
                                    return calculatedhash;
                            }

                            //Increases nonce value until hash target is reached.
                            public void mineBlock(int difficulty) {
                                    merkleRoot = StringUtil.getMerkleRoot(transactions);
                                    String target = StringUtil.getDificultyString(difficulty); //Create a
                    string with difficulty * "0"
                                    while(!hash.substring( 0, difficulty).equals(target)) {
                                            nonce ++;
                                            hash = calculateHash();
                                    }
                                    System.out.println("Block Mined!!! : " + hash);
                            }

                            //Add transactions to this block
                            public boolean addTransaction(Transaction transaction) {

                                    //process transaction and check if valid, unless block is genesis block
                    then ignore.
                                    if(transaction == null) return false;
                                    if(previousHash != "0") {
                                            if((transaction.processTransaction() != true)) {
                                                    System.out.println("Transaction failed to process.
                    Discarded.");
                                                    return false;
                                            }
                                    }
                                    transactions.add(transaction);
                                    System.out.println("Transaction Successfully added to Block");
                                    return true;
                            }

                    }
```

**FIGURE 5.37:** Implementing our changes to our Block class.

## 5.4 SUCCESS

### 5.41 TRANSACTIONS CONFIRMED

After we got everything said and done, we then tested our final code on our server machine and to do that, please see implementation below:

```
public class StmarytxChain {

	public static ArrayList<Block> blockchain = new ArrayList<Block>();
	public static HashMap<String,TransactionOutput> UTXOs = new
HashMap<String,TransactionOutput>();

	public static int difficulty = 3;
	public static float minimumTransaction = 0.1f;
	public static Wallet walletA;
	public static Wallet walletB;
	public static Transaction genesisTransaction;

	public static void main(String[] args) {
		//add our blocks to the blockchain ArrayList:
		Security.addProvider(new
org.bouncycastle.jce.provider.BouncyCastleProvider()); //Setup Bouncey castle as a
Security Provider

		//Create wallets:
		walletA = new Wallet();
		walletB = new Wallet();
		Wallet coinbase = new Wallet();

		//create genesis transaction, which sends 100 StmarytxCoin to
walletA:
		genesisTransaction = new Transaction(coinbase.publicKey,
walletA.publicKey, 100f, null);
		genesisTransaction.generateSignature(coinbase.privateKey); //manually
sign the genesis transaction
		genesisTransaction.transactionId = "0"; //manually set the
transaction id
		genesisTransaction.outputs.add(new
TransactionOutput(genesisTransaction.reciepient, genesisTransaction.value,
genesisTransaction.transactionId)); //manually add the Transactions Output
		UTXOs.put(genesisTransaction.outputs.get(0).id,
genesisTransaction.outputs.get(0)); //its important to store our first transaction
in the UTXOs list.
```

```
		System.out.println("Creating and Mining Genesis block... ");
		Block genesis = new Block("0");
		genesis.addTransaction(genesisTransaction);
		addBlock(genesis);

		//testing
		Block block1 = new Block(genesis.hash);
		System.out.println("\nWalletA's balance is: " +
walletA.getBalance());
		System.out.println("\nWalletA is Attempting to send funds (40) to
WalletB...");
		block1.addTransaction(walletA.sendFunds(walletB.publicKey, 40f));
		addBlock(block1);
		System.out.println("\nWalletA's balance is: " +
walletA.getBalance());
		System.out.println("WalletB's balance is: " + walletB.getBalance());

		Block block2 = new Block(block1.hash);
		System.out.println("\nWalletA Attempting to send more funds (1000)
than it has...");
		block2.addTransaction(walletA.sendFunds(walletB.publicKey, 1000f));
		addBlock(block2);
		System.out.println("\nWalletA's balance is: " +
walletA.getBalance());
		System.out.println("WalletB's balance is: " + walletB.getBalance());

		Block block3 = new Block(block2.hash);
		System.out.println("\nWalletB is Attempting to send funds (20) to
WalletA...");
		block3.addTransaction(walletB.sendFunds( walletA.publicKey, 20));
		System.out.println("\nWalletA's balance is: " +
walletA.getBalance());
		System.out.println("WalletB's balance is: " + walletB.getBalance());

		isChainValid();

	}

	public static Boolean isChainValid() {
		Block currentBlock;
		Block previousBlock;
```

```
		String hashTarget = new String(new char[difficulty]).replace('\0',
'0');
		HashMap<String,TransactionOutput> tempUTXOs = new
HashMap<String,TransactionOutput>(); //a temporary working list of unspent
transactions at a given block state.
		tempUTXOs.put(genesisTransaction.outputs.get(0).id,
genesisTransaction.outputs.get(0));

		//loop through blockchain to check hashes:
		for(int i=1; i < blockchain.size(); i++) {

			currentBlock = blockchain.get(i);
			previousBlock = blockchain.get(i-1);
			//compare registered hash and calculated hash:
			if(!currentBlock.hash.equals(currentBlock.calculateHash()) ){
				System.out.println("#Current Hashes not equal");
				return false;
			}
			//compare previous hash and registered previous hash
			if(!previousBlock.hash.equals(currentBlock.previousHash) ) {
				System.out.println("#Previous Hashes not equal");
				return false;
			}
			//check if hash is solved
			if(!currentBlock.hash.substring( 0,
difficulty).equals(hashTarget)) {
				System.out.println("#This block hasn't been mined");
				return false;
			}

			//loop thru blockchains transactions:
			TransactionOutput tempOutput;
			for(int t=0; t <currentBlock.transactions.size(); t++) {
				Transaction currentTransaction =
currentBlock.transactions.get(t);

				if(!currentTransaction.verifiySignature()) {
					System.out.println("#Signature on Transaction("
+ t + ") is Invalid");
					return false;
				}
				if(currentTransaction.getInputsValue() !=
currentTransaction.getOutputsValue()) {
```

```
                                System.out.println("#Inputs are note equal to
outputs on Transaction(" + t + ")");
                                return false;
                        }

                        for(TransactionInput input: currentTransaction.inputs)
{
                                tempOutput =
tempUTXOs.get(input.transactionOutputId);

                                if(tempOutput == null) {
                                        System.out.println("#Referenced input on
Transaction(" + t + ") is Missing");
                                        return false;
                                }

                                if(input.UTXO.value != tempOutput.value) {
                                        System.out.println("#Referenced input
Transaction(" + t + ") value is Invalid");
                                        return false;
                                }

                                tempUTXOs.remove(input.transactionOutputId);
                        }

                        for(TransactionOutput output:
currentTransaction.outputs) {
                                tempUTXOs.put(output.id, output);
                        }

                        if( currentTransaction.outputs.get(0).reciepient !=
currentTransaction.reciepient) {
                                System.out.println("#Transaction(" + t + ")
output reciepient is not who it should be");
                                return false;
                        }
                        if( currentTransaction.outputs.get(1).reciepient !=
currentTransaction.sender) {
                                System.out.println("#Transaction(" + t + ")
output 'change' is not sender.");
                                return false;
                        }
```


**FIGURE 5.41:** The above figures show the implementation of our final codes on our server.

# CHAPTER SIX

## SUMMARY

In the previous chapters, we have seen the step-by-step process on how this project was implemented and all the tools used. In summary the main functionality of the application is as follows:

When the customer information is inputted, the information is encrypted, and the data runs through a blockchain database API before getting to the database. The database then hashes the information and sends it back to the blockchain API which then uses the public key of the user to decrypt the hashed value. The hash value of the user's phone number and email are used to prevent account duplication and to ensure only reliable records are being used. After which the bank is then responsible for connecting to the database by releasing the bank records of each transaction.

With the implementation of our application, we have been able to apply most of the necessary security features and requirements we learnt in this course. It has also strongly tested our skills and given us a better understanding on how to apply the skills of implementation, debugging and testing. This experience has prepared us for our future in the Information Technology world.

## FUTURE WORK

As we have developed this application for the exchange of transactions between parties, one way the project can be improved is by implementing automated sorting and balancing of records. This will provide a more accurate and less stressful way of sorting and generating transaction statements.

Also, as this application was designed specifically for our institution St. Mary's a customization of the user interface was eliminated and will need to be included to improve the appearance and uniqueness of the application.

# REFERENCES

[1] Keshavan, N., Keshavan, N., Technologies, C., Technologies, C. and Technologies, C., 2021. Testing Online Banking Applications: A Hypothesis. [online] Software Testing Blog by Cigniti Technologies. Available at: <https://www.cigniti.com/blog/testing-online-banking-applications-hypothesis/> [Accessed 12 February 2021].

[2] 2021. [online] Available at: <https://www.researchgate.net/figure/Class-Diagram-of-Internet-Banking-System-5-DATA-FLOW-DIAGRAM_fig8_45919149> [Accessed 12 February 2021].

# CODE LISTING

```
public class StmarytxChain {

        public static void main(String[] args) {

                Block genesisBlock = new Block("Hi im the first block", "0");

                System.out.println("Hash for block 1 : " + genesisBlock.hash);

                Block secondBlock = new Block("simple second block",genesisBlock.hash);

                System.out.println("Hash for block 2 : " + secondBlock.hash);

                Block thirdBlock = new Block("Another simple third block",secondBlock.hash);

                System.out.println("Hash for block 3 : " + thirdBlock.hash);

        }
}
```

- BLOCK

```
import
java.util.Date;

        public class Block {

                public String hash;

                public String previousHash;

                private String data; //our data will be a simple message.

                private long timeStamp; //as number of milliseconds since 1/1/1970.

                //Block Constructor.

                public Block(String data,String previousHash ) {

                        this.data = data;
```

```
                this.previousHash = previousHash;
                this.timeStamp = new Date().getTime();
            }
        }
```

- STRINGUTIL

```
import java.security.MessageDigest;

public class StringUtil {
    //Applies Sha256 to a string and returns the result.
    public static String applySha256(String input){

        try {
            MessageDigest digest = MessageDigest.getInstance("SHA-256");
            //Applies sha256 to our input,
            byte[] hash = digest.digest(input.getBytes("UTF-8"));
            StringBuffer hexString = new StringBuffer(); // This will contain hash as hexidecimal
            for (int i = 0; i < hash.length; i++) {
                String hex = Integer.toHexString(0xff & hash[i]);
                if(hex.length() == 1) hexString.append('0');
                hexString.append(hex);
            }
            return hexString.toString();
        }
        catch(Exception e) {
            throw new RuntimeException(e);
        }
    }
```

```
                        }
```

- CALCULATEHASH

```
public String
calculateHash()
{
                    String calculatedhash = StringUtil.applySha256(
                                previousHash +
                                Long.toString(timeStamp) +
                                data
                                );
                    return calculatedhash;
            }
```

- BLOCK (CONTRUCTOR)

```
public Block(String data,String previousHash ) {
                this.data = data;
                this.previousHash = previousHash;
                this.timeStamp = new Date().getTime();
                this.hash = calculateHash(); //Making sure
            we do this after we set the other values.
                    }
```

- STORING OUR BLOCKS IN AN ARRAYLIST AND IMPORTING WITH GSON

```
import
java.util.ArrayList;
                    import com.google.gson.GsonBuilder;


                    public class StmarytxChain {


                            public static ArrayList<Block> blockchain = new
                    ArrayList<Block>();


                            public static void main(String[] args) {
                                    //add our blocks to the blockchain ArrayList:
```

```
                blockchain.add(new Block("Hi im the first block", "0"));

                blockchain.add(new Block("Yo im the second
        block",blockchain.get(blockchain.size()-1).hash));
                blockchain.add(new Block("Hey im the third
        block",blockchain.get(blockchain.size()-1).hash));

                String blockchainJson = new
        GsonBuilder().setPrettyPrinting().create().toJson(blockchain);

                System.out.println(blockchainJson);
            }

        }
```

- ISCHAINVALID

```
public static Boolean isChainValid() {
        Block currentBlock;
        Block previousBlock;

        //loop through blockchain to check hashes:
        for(int i=1; i < blockchain.size(); i++) {
            currentBlock = blockchain.get(i);
            previousBlock = blockchain.get(i-1);
            //compare registered hash and calculated hash:
            if(!currentBlock.hash.equals(currentBlock.calculateHash()) ){
                System.out.println("Current Hashes not equal");
                return false;
            }
            //compare previous hash and registered previous hash
            if(!previousBlock.hash.equals(currentBlock.previousHash) ) {
```

```
                        System.out.println("Previous Hashes not equal");
                        return false;
                }
        }
        return true;
  }
```

- MINING BLOCKS

```
import java.util.Date;

                        public class Block {

                                public String hash;
                                public String previousHash;
                                private String data; //our data will be a simple message.
                                private long timeStamp; //as number of milliseconds since
                        1/1/1970.
                                private int nonce;

                                //Block Constructor.
                                public Block(String data,String previousHash ) {
                                        this.data = data;
                                        this.previousHash = previousHash;
                                        this.timeStamp = new Date().getTime();

                                        this.hash = calculateHash(); //Making sure we do
                        this after we set the other values.
                                }

                                //Calculate new hash based on blocks contents
                                public String calculateHash() {
```

```
                String calculatedhash = StringUtil.applySha256(
                        previousHash +
                        Long.toString(timeStamp) +
                        Integer.toString(nonce) +
                        data
                        );
                return calculatedhash;
        }

        public void mineBlock(int difficulty) {
                String target = new String(new
char[difficulty]).replace('\0', '0'); //Create a string with
difficulty * "0"
                while(!hash.substring( 0,
difficulty).equals(target)) {
                        nonce ++;
                        hash = calculateHash();
                }
                System.out.println("Block Mined!!! : " + hash);
        }
}
```

```
import
java.util.ArrayList;
import com.google.gson.GsonBuilder;

public class StmarytxChain {

        public static ArrayList<Block> blockchain = new
ArrayList<Block>();
        public static int difficulty = 5;
```

```
        public static void main(String[] args) {
                //add our blocks to the blockchain ArrayList:

                blockchain.add(new Block("Hi im the first block",
"0"));
                System.out.println("Trying to Mine block 1... ");
                blockchain.get(0).mineBlock(difficulty);

                blockchain.add(new Block("Yo im the second
block",blockchain.get(blockchain.size()-1).hash));
                System.out.println("Trying to Mine block 2... ");
                blockchain.get(1).mineBlock(difficulty);

                blockchain.add(new Block("Hey im the third
block",blockchain.get(blockchain.size()-1).hash));
                System.out.println("Trying to Mine block 3... ");
                blockchain.get(2).mineBlock(difficulty);

                System.out.println("\nBlockchain is Valid: " +
isChainValid());

                String blockchainJson = new
GsonBuilder().setPrettyPrinting().create().toJson(blockchain);
                System.out.println("\nThe block chain: ");
                System.out.println(blockchainJson);
        }

        public static Boolean isChainValid() {
                Block currentBlock;
                Block previousBlock;
                String hashTarget = new String(new
char[difficulty]).replace('\0', '0');
```

```
		//loop through blockchain to check hashes:
		for(int i=1; i < blockchain.size(); i++) {
			currentBlock = blockchain.get(i);
			previousBlock = blockchain.get(i-1);
			//compare registered hash and calculated
hash:

	if(!currentBlock.hash.equals(currentBlock.calculateHash()
) ){
				System.out.println("Current Hashes
not equal");
				return false;
			}
			//compare previous hash and registered
previous hash

	if(!previousBlock.hash.equals(currentBlock.previousHash)
) {
				System.out.println("Previous Hashes
not equal");
				return false;
			}
			//check if hash is solved
			if(!currentBlock.hash.substring( 0,
difficulty).equals(hashTarget)) {
				System.out.println("This block
hasn't been mined");
				return false;
			}
		}
		return true;
	}
```

```
                        }
```

- WALLET CLASS

```
package Stmarytxchain;

import java.security.*;


public class Wallet {

        public PrivateKey privateKey;

        public PublicKey publicKey;

}
```

- GENERATE PRIVATE AND PUBLIC KEYS

```
Package

Stmarytxchain;

                import java.security.*;


                public class Wallet {


                        public PrivateKey privateKey;

                        public PublicKey publicKey;


                        public Wallet(){

                                generateKeyPair();

                        }


                        public void generateKeyPair() {

                                try {

                                        KeyPairGenerator keyGen =
                KeyPairGenerator.getInstance("ECDSA","BC");

                                        SecureRandom random =
                SecureRandom.getInstance("SHA1PRNG");
```

```
                        ECGenParameterSpec ecSpec = new
ECGenParameterSpec("prime192v1");
                        // Initialize the key generator and generate a KeyPair
                        keyGen.initialize(ecSpec, random);   //256 bytes
provides an acceptable security level
                        KeyPair keyPair = keyGen.generateKeyPair();
                        // Set the public and private keys from the keyPair
                        privateKey = keyPair.getPrivate();
                        publicKey = keyPair.getPublic();
                    }catch(Exception e) {
                        throw new RuntimeException(e);
                    }
            }

}
```

- CREATING TRANSACTION CLASS

```
import
java.security.*;
    import java.util.ArrayList;

    public class Transaction {

        public String transactionId; // this is also the hash of the
    transaction.
        public PublicKey sender; // senders address/public key.
        public PublicKey reciepient; // Recipients address/public key.
        public float value;
        public byte[] signature; // this is to prevent anybody else from
    spending funds in our wallet.
```

```
        public ArrayList<TransactionInput> inputs = new
ArrayList<TransactionInput>();

        public ArrayList<TransactionOutput> outputs = new
ArrayList<TransactionOutput>();


        private static int sequence = 0; // a rough count of how many
transactions have been generated.


        // Constructor:

        public Transaction(PublicKey from, PublicKey to, float value,
ArrayList<TransactionInput> inputs) {

                this.sender = from;

                this.reciepient = to;

                this.value = value;

                this.inputs = inputs;

        }


        // This Calculates the transaction hash (which will be used as its
Id)

        private String calulateHash() {

                sequence++; //increase the sequence to avoid 2 identical
transactions having the same hash

                return StringUtil.applySha256(

                                StringUtil.getStringFromKey(sender) +

                                StringUtil.getStringFromKey(reciepient) +

                                Float.toString(value) + sequence

                                );

        }

}
```

- TRANSACTION INPUT AND OUTPUT CLASS

```
public class TransactionInput {
```

```
        public String transactionOutputId; //Reference to TransactionOutputs -> transactionId
        public TransactionOutput UTXO; //Contains the Unspent transaction output

        public TransactionInput(String transactionOutputId) {
                this.transactionOutputId = transactionOutputId;
        }
}


import java.security.PublicKey;

public class TransactionOutput {
        public String id;
        public PublicKey reciepient; //also known as the new owner of these coins.
        public float value; //the amount of coins they own
        public String parentTransactionId; //the id of the transaction this output was created in

        //Constructor
        public TransactionOutput(PublicKey reciepient, float value, String parentTransactionId) {
                this.reciepient = reciepient;
                this.value = value;
                this.parentTransactionId = parentTransactionId;
                this.id = StringUtil.applySha256(StringUtil.getStringFromKey(reciepient)+Float.toString(value)+parentTransactionId);
        }

        //Check if coin belongs to you
        public boolean isMine(PublicKey publicKey) {
```

```
            return (publicKey == reciepient);
        }

}
```

- COLLECTION OF ALL UTXOS

```
public class StmarytxChain {

        public static ArrayList<Block> blockchain = new ArrayList<Block>();
        public static HashMap<String,TransactionOutputs> UTXOs = new
HashMap<String,TransactionOutputs>(); //list of all unspent transactions.
        public static int difficulty = 5;
        public static Wallet walletA;
        public static Wallet walletB;

        public static void main(String[] args) {
```

- BOOLEAN METHOD

```
//Returns
true if new
transaction
could be
created.

          public boolean processTransaction() {

                    if(verifiySignature() == false) {
                              System.out.println("#Transaction Signature failed to
          verify");
                              return false;
                    }
```

```
		//gather transaction inputs (Make sure they are unspent):
		for(TransactionInput i : inputs) {
			i.UTXO = StmarytxChain.UTXOs.get(i.transactionOutputId);
		}

		//check if transaction is valid:
		if(getInputsValue() < StmarytxChain.minimumTransaction) {
			System.out.println("#Transaction Inputs to small: " +
getInputsValue());
			return false;
		}

		//generate transaction outputs:
		float leftOver = getInputsValue() - value; //get value of inputs
then the left over change:
		transactionId = calulateHash();
		outputs.add(new TransactionOutput( this.reciepient,
value,transactionId)); //send value to recipient
		outputs.add(new TransactionOutput( this.sender,
leftOver,transactionId)); //send the left over 'change' back to sender

		//add outputs to Unspent list
		for(TransactionOutput o : outputs) {
			StmarytxChain.UTXOs.put(o.id , o);
		}

		//remove transaction inputs from UTXO lists as spent:
		for(TransactionInput i : inputs) {
			if(i.UTXO == null) continue; //if Transaction can't be
found skip it
			StmarytxChain.UTXOs.remove(i.UTXO.id);
		}
```

```
                return true;
        }

    //returns sum of inputs(UTXOs) values
        public float getInputsValue() {
                float total = 0;
                for(TransactionInput i : inputs) {
                        if(i.UTXO == null) continue; //if Transaction can't be
    found skip it
                        total += i.UTXO.value;
                }
                return total;
        }

    //returns sum of outputs:
        public float getOutputsValue() {
                float total = 0;
                for(TransactionOutput o : outputs) {
                        total += o.value;
                }
                return total;
        }
```

- WALLET UPDATE

```
import
java.security.*;
        import java.security.spec.ECGenParameterSpec;
        import java.util.ArrayList;
        import java.util.HashMap;
        import java.util.Map;
```

```
public class Wallet {

	public PrivateKey privateKey;
	public PublicKey publicKey;

	public HashMap<String,TransactionOutput> UTXOs = new
HashMap<String,TransactionOutput>(); //only UTXOs owned by this wallet.

	public Wallet() {...

	public void generateKeyPair() {...

  //returns balance and stores the UTXO's owned by this wallet in
this.UTXOs
	public float getBalance() {
		float total = 0;
	 for (Map.Entry<String, TransactionOutput> item:
NoobChain.UTXOs.entrySet()){
		TransactionOutput UTXO = item.getValue();
	    if(UTXO.isMine(publicKey)) { //if output belongs to me ( if
coins belong to me )
		UTXOs.put(UTXO.id,UTXO); //add it to our list of unspent
transactions.
		total += UTXO.value ;
	    }
	 }
		return total;
	}
	//Generates and returns a new transaction from this wallet.
	public Transaction sendFunds(PublicKey _recipient,float value ) {
		if(getBalance() < value) { //gather balance and check funds.
```

```
                        System.out.println("#Not Enough funds to send
transaction. Transaction Discarded.");
                        return null;
                }
    //create array list of inputs
                ArrayList<TransactionInput> inputs = new
ArrayList<TransactionInput>();

                float total = 0;
                for (Map.Entry<String, TransactionOutput> item:
UTXOs.entrySet()){
                        TransactionOutput UTXO = item.getValue();
                        total += UTXO.value;
                        inputs.add(new TransactionInput(UTXO.id));
                        if(total > value) break;
                }

                Transaction newTransaction = new Transaction(publicKey,
_recipient , value, inputs);
                newTransaction.generateSignature(privateKey);

                for(TransactionInput input: inputs){
                        UTXOs.remove(input.transactionOutputId);
                }
                return newTransaction;
        }

}
```

- GENERATE MERKLE ROOTS

```
//Tacks in array of
transactions and
returns a merkle root.
                        public static String getMerkleRoot(ArrayList<Transaction>
transactions) {
                                int count = transactions.size();
                                ArrayList<String> previousTreeLayer = new
                        ArrayList<String>();
                                for(Transaction transaction : transactions) {

                               previousTreeLayer.add(transaction.transactionId);
                                }
                                ArrayList<String> treeLayer = previousTreeLayer;
                                while(count > 1) {
                                        treeLayer = new ArrayList<String>();
                                        for(int i=1; i < previousTreeLayer.size();
                        i++) {

                               treeLayer.add(applySha256(previousTreeLayer.get(i-1) +
                        previousTreeLayer.get(i)));
                                        }
                                        count = treeLayer.size();
                                        previousTreeLayer = treeLayer;
                                }
                                String merkleRoot = (treeLayer.size() == 1) ?
                        treeLayer.get(0) : "";
                                return merkleRoot;
                               }
import java.util.ArrayList;
                        import java.util.Date;

                        public class Block {
```

```
	public String hash;
	public String previousHash;
	public String merkleRoot;
	public ArrayList<Transaction> transactions = new
ArrayList<Transaction>(); //our data will be a simple message.
	public long timeStamp; //as number of milliseconds since
1/1/1970.
	public int nonce;

	//Block Constructor.
	public Block(String previousHash ) {
		this.previousHash = previousHash;
		this.timeStamp = new Date().getTime();

		this.hash = calculateHash(); //Making sure we do this after
we set the other values.
	}

	//Calculate new hash based on blocks contents
	public String calculateHash() {
		String calculatedhash = StringUtil.applySha256(
				previousHash +
				Long.toString(timeStamp) +
				Integer.toString(nonce) +
				merkleRoot
				);
		return calculatedhash;
	}

	//Increases nonce value until hash target is reached.
	public void mineBlock(int difficulty) {
		merkleRoot = StringUtil.getMerkleRoot(transactions);
```

```
                String target = StringUtil.getDificultyString(difficulty);
//Create a string with difficulty * "0"
                while(!hash.substring( 0, difficulty).equals(target)) {
                        nonce ++;
                        hash = calculateHash();
                }
                System.out.println("Block Mined!!! : " + hash);
        }

        //Add transactions to this block
        public boolean addTransaction(Transaction transaction) {
                //process transaction and check if valid, unless block is
genesis block then ignore.
                if(transaction == null) return false;
                if((previousHash != "0")) {
                        if((transaction.processTransaction() != true)) {
                                System.out.println("Transaction failed to
process. Discarded.");
                                return false;
                        }
                }
                transactions.add(transaction);
                System.out.println("Transaction Successfully added to
Block");
                return true;
        }

}
```

- FINAL TESTING OF CODES

```
public class StmarytxChain {

        public static ArrayList<Block> blockchain = new ArrayList<Block>();
```

```
	public static HashMap<String,TransactionOutput> UTXOs = new
HashMap<String,TransactionOutput>();


	public static int difficulty = 3;

	public static float minimumTransaction = 0.1f;

	public static Wallet walletA;

	public static Wallet walletB;

	public static Transaction genesisTransaction;


	public static void main(String[] args) {

		//add our blocks to the blockchain ArrayList:

		Security.addProvider(new
org.bouncycastle.jce.provider.BouncyCastleProvider()); //Setup Bouncey castle as a Security
Provider


		//Create wallets:

		walletA = new Wallet();

		walletB = new Wallet();

		Wallet coinbase = new Wallet();


		//create genesis transaction, which sends 100 StmarytxCoin to walletA:

		genesisTransaction = new Transaction(coinbase.publicKey, walletA.publicKey,
100f, null);

		genesisTransaction.generateSignature(coinbase.privateKey); //manually sign
the genesis transaction

		genesisTransaction.transactionId = "0"; //manually set the transaction id

		genesisTransaction.outputs.add(new
TransactionOutput(genesisTransaction.reciepient, genesisTransaction.value,
genesisTransaction.transactionId)); //manually add the Transactions Output

		UTXOs.put(genesisTransaction.outputs.get(0).id,
genesisTransaction.outputs.get(0)); //its important to store our first transaction in the
UTXOs list.
```

```
        System.out.println("Creating and Mining Genesis block... ");
        Block genesis = new Block("0");
        genesis.addTransaction(genesisTransaction);
        addBlock(genesis);

        //testing
        Block block1 = new Block(genesis.hash);
        System.out.println("\nWalletA's balance is: " + walletA.getBalance());
        System.out.println("\nWalletA is Attempting to send funds (40) to
WalletB...");
        block1.addTransaction(walletA.sendFunds(walletB.publicKey, 40f));
        addBlock(block1);
        System.out.println("\nWalletA's balance is: " + walletA.getBalance());
        System.out.println("WalletB's balance is: " + walletB.getBalance());

        Block block2 = new Block(block1.hash);
        System.out.println("\nWalletA Attempting to send more funds (1000) than it
has...");
        block2.addTransaction(walletA.sendFunds(walletB.publicKey, 1000f));
        addBlock(block2);
        System.out.println("\nWalletA's balance is: " + walletA.getBalance());
        System.out.println("WalletB's balance is: " + walletB.getBalance());

        Block block3 = new Block(block2.hash);
        System.out.println("\nWalletB is Attempting to send funds (20) to
WalletA...");
        block3.addTransaction(walletB.sendFunds( walletA.publicKey, 20));
        System.out.println("\nWalletA's balance is: " + walletA.getBalance());
        System.out.println("WalletB's balance is: " + walletB.getBalance());

        isChainValid();
```

```
	}

	public static Boolean isChainValid() {
		Block currentBlock;
		Block previousBlock;
		String hashTarget = new String(new char[difficulty]).replace('\0', '0');
		HashMap<String,TransactionOutput> tempUTXOs = new
HashMap<String,TransactionOutput>(); //a temporary working list of unspent transactions at
a given block state.
		tempUTXOs.put(genesisTransaction.outputs.get(0).id,
genesisTransaction.outputs.get(0));

		//loop through blockchain to check hashes:
		for(int i=1; i < blockchain.size(); i++) {

			currentBlock = blockchain.get(i);
			previousBlock = blockchain.get(i-1);
			//compare registered hash and calculated hash:
			if(!currentBlock.hash.equals(currentBlock.calculateHash()) ){
				System.out.println("#Current Hashes not equal");
				return false;
			}
			//compare previous hash and registered previous hash
			if(!previousBlock.hash.equals(currentBlock.previousHash) ) {
				System.out.println("#Previous Hashes not equal");
				return false;
			}
			//check if hash is solved
			if(!currentBlock.hash.substring( 0, difficulty).equals(hashTarget)) {
				System.out.println("#This block hasn't been mined");
```

```
                return false;
            }

            //loop thru blockchains transactions:
            TransactionOutput tempOutput;
            for(int t=0; t <currentBlock.transactions.size(); t++) {
                Transaction currentTransaction =
currentBlock.transactions.get(t);

                if(!currentTransaction.verifiySignature()) {
                    System.out.println("#Signature on Transaction(" + t +
") is Invalid");
                    return false;
                }
                if(currentTransaction.getInputsValue() !=
currentTransaction.getOutputsValue()) {
                    System.out.println("#Inputs are note equal to outputs
on Transaction(" + t + ")");
                    return false;
                }

                for(TransactionInput input: currentTransaction.inputs) {
                    tempOutput = tempUTXOs.get(input.transactionOutputId);

                    if(tempOutput == null) {
                        System.out.println("#Referenced input on
Transaction(" + t + ") is Missing");
                        return false;
                    }

                    if(input.UTXO.value != tempOutput.value) {
```

```
                                        System.out.println("#Referenced input
Transaction(" + t + ") value is Invalid");
                                        return false;
                                }

                                tempUTXOs.remove(input.transactionOutputId);
                        }

                        for(TransactionOutput output: currentTransaction.outputs) {
                                tempUTXOs.put(output.id, output);
                        }

                        if( currentTransaction.outputs.get(0).reciepient !=
currentTransaction.reciepient) {
                                System.out.println("#Transaction(" + t + ") output
reciepient is not who it should be");
                                return false;
                        }
                        if( currentTransaction.outputs.get(1).reciepient !=
currentTransaction.sender) {
                                System.out.println("#Transaction(" + t + ") output
'change' is not sender.");
                                return false;
                        }

                }

        }
        System.out.println("Blockchain is valid");
        return true;
    }
```

```
        public static void addBlock(Block newBlock) {
                newBlock.mineBlock(difficulty);
                blockchain.add(newBlock);
        }
}
```