Aytcoin

"Blockchain and Cryptocurrency Project"

By
Ahmad Shalabi
2018

Table of Contents

Introduction

Atycoin is a **Cryptocurrency** implemented using **blockchain** technology that include these features

- Core system
  - Block
  - Transaction
  - Base58 Addresses
  - Redis database to store blocks
- Proof-Of-Work
- Transaction system
  - Wallet
  - MerkleTree
  - Redis database to store unspent transaction output
- Peer-to-Peer network
  - Messages
  - Commands handler
  - Mempool to store unsaved transaction
- Command-line interface
  - Commands

Part 1 : Clean Code Principles

1 Meaningful Names

This Chapter discuss naming of variables, functions, arguments, classes and packages.

- Most of Names are revealing its intention and clean from disinformation, like:
  1. Package names, like ***messages*** and ***commands*** in network package and ***utility.***
  2. Classes name, like ***Block*** that hold data about each block in blockchain and ***TransactionOutput*** that hold the new ownership of coin and the coins value.
  3. Variables, like ***senderAddress*** in ***NetworkMessage*** class, ***timestamp*** in ***Block*** and ***Transaction***, and ***requiredHeight*** in ***GetBlocksMessage.***
  4. Functions, like ***makeRequest*** in ***NetworkMessage***, ***newCoinbaseTransaction*** in ***Transaction*** that used to make the first block in blockchain – Genesis Block – and as reward in each new block.
  5. Arguments, like arguments in ***newTransactinon***

```
public static Transaction newTransaction(Wallet sender, String recipient, int amount)
{ … }
```

- Use meaningful distinctions, pronounceable names and searchable Names, like ***privatekeyEncoded*** and ***publicKeyEncoded*** in ***Wallet*** class.
- Avoid Encoding, like in ***Command*** interface in ***cli.commands*** package, and ***NetworkMessage*** and ***NetworkCommand*** in ***networks*** package.
- Classes names have noun or noun phrase names like ***Node***, ***Block***, ***Transaction*** and ***Wallet***.
- Method names have verb or verb phrase names like ***mineBlock*** and ***addBlock*** in ***Blockchain,*** **getBalance**, **buildInputs** and ***buildOutputs*** in ***Transaction***, and ***execute***, ***send*** and ***getOutputStream*** in ***NetworkCommand*** class.
- Use Solution Doman Names, like ***BlocksDAO*** that responsible for access data object to handle storing and retrieving blocks in **redis database**, and ***NetworkCommandFactory*** that used to execute the correct response to each message between different nodes in network communications.
- Meaningful context as in encapsulate some data of ***Transaction*** in ***TrainsactionInput*** and ***TransactionOutput***.

2 Functions

- Most of the functions are small and do one thing and have not side effects, like:

  1. All the functions of ***Wallet*** class, as

     - ***getPrivateKey***, that convert ***priavteKeyEnocded*** to ***ECPrivateKey*** object

```
public ECPrivateKey getPrivateKey() {
    return Keys.getPrivateKey(privateKeyEncoded);
}
```

     - ***getAddress,*** that get **Base58** format of the specific key

```
public String getAddress() {
    byte[] publicKeyHashed = getPublicKeyHashed();
    return Address.getAddress(publicKeyHashed);
}
```

  2. ***send*** in ***NetworkCommand*** that responsible about send message to specific recipient using socket

```
protected void send(int recipient, NetworkMessage message) {
    try (Socket sendingConnection = new Socket(InetAddress.getLocalHost(),  recipient))
{
        getOutputStream(sendingConnection);
        String request = message.makeRequest();
        output.write(request);
        output.flush();
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}
```

  3. ***handleConnection***

```
private void handleConnection(Socket connection, int nodeAddress) {
    pool.execute(new ConnectionHandler(connection, nodeAddress));
}
```

4. ***getBalance*** in ***ChainState*** that use ***ChainStateDAO*** to calculate balance of specific publicKey

```
public int getBalance(byte[] publicKeyHashed) {
    int balance = 0;
    Set<String> transactionIDs = chainStateDAO.getAllReferenceTransaction();
    for (String transactionID : transactionIDs) {
        List<TransactionOutput> outputs =
chainStateDAO.getReferenceOutputs(transactionID);
        for (TransactionOutput output : outputs) {
            if (output.isLockedWithKey(publicKeyHashed)) {
                balance += output.getValue();
            }
        }
    }
    return balance;
}
```

- One level of abstraction and command query separation, like ***hasNext*** in ***BlocksIterator***

```
@Override
public boolean hasNext() {
    if (currentHashSerialized == null) {
        return false;
    }
    return !currentHashSerialized.equals(GENESIS_PREVIOUS_HASH);
}
```

- Minimize switch statement through buried it in low-level class or considering design pattern or using some way, like in ***NetworkCommandFactory*** that used **map** to store command object to get it from the map instead of using switch statement to compare the cases to make command object.

```
private static void initializeCommands() {
    commands.put("addresses", new AddressesCommandHandler());
    commands.put("block", new BlockCommandHandler());

                                        ...

}

public static NetworkCommand getCommand(String command) {
    NetworkCommand networkCommand = commands.get(command);

    if (networkCommand == null) {
        networkCommand = NULL_COMMAND;
    }
    return networkCommand;
}
```

- Function arguments
  - There are not function that have flag argument – make more than one thing-
  - Minimize number of arguments of function, as in these cases:
    - ***updateBlockHeader*** in ***ProofOfWork*** that make block object variable instance instead of pass it as parameter to function

```
private final Block block;

      …

private void updateBlockHeader(int nonce) {
    byte[] nonceBytes = Bytes.toBytes(nonce);
    //update the last 4 bytes
    System.arraycopy(nonceBytes, 0, blockHeader, blockHeader.length - 4, 4);
}
```

    - Make ***BlocksDAO*** and ***ChainStateDAO*** instance variable instead of pass it as parameter in ***Blockchain*** and **ChainState** respectively.
- Prefer exception to returning error codes, like ***getBlock*** in ***BlocksDAO***

```
public Block getBlock(String blockHash) {
    String blockSerialized = dbConnection.get(blockHash);

    if (blockSerialized == null) {
        throw new NoSuchElementException("Block not found in database.");
    }
    return deserialize(blockSerialized);
}
```

- Do not repeat yourself, like put ***serialize*** in abstract ***NetworkMessage*** instead of put it concrete implementation

```
protected final String serialize(String command, NetworkMessage message) {
    String msg = encoder.toJson(message);
    return command + msg;
}
```

3 Comments

- Inappropriate information like author name and last modified time managed by git.
- There are not commented-out code
- The code is clean from old and redundant comment
- comment written well, like

```
//Coinbase: One input, empty transactionID and -1 index
public boolean isCoinbaseTransaction() {
    TransactionInput input = inputs.get(0);
    return inputs.size() == 1 &&
            input.getReferenceTransaction().length == 0 &&
            input.getOutputIndex() == COINBASE_INDEX;
}
```

- TODO comments, that used for example to add improvement to enhance unfinished concept, like
  - ***Block*** class

```
private final int targetBits = 20; //TODO: Make it Adjusted to meet some
requirements
```

  - ***Trnsaction*** class

```
private static final int reward = 10; //TODO: Make reward adjustable
```

  - ***runProofOfWork***

```
//TODO: Add extraNonce when nonce overflow occurs
public int runProofOfWork() {
    //initialize blockHeader
    int nonce = 0;
    blockHeader = block.setBlockHeader(nonce);

    while (nonce < Integer.MAX_VALUE) {
        updateBlockHeader(nonce);
        byte[] hash = Hash.doubleSHA256(blockHeader);
        if (isHashMeetTarget(hash)) {
            break;
        }
        nonce++;
    }
    return nonce;
}
```

4 Formatting

- Vertical formatting

  1. Variables declared as close to their usage as possible.
  2. Instance variables at the beginning of class.
  3. Dependent functions close as possible to their calls.
  4. Conceptual affinity grouped together.

Like

```
public byte[] setBlockHeader(int nonce) {
    ByteArrayOutputStream buffer = new ByteArrayOutputStream();

    try {
        buffer.write(Bytes.toBytes(version));
        buffer.write(hashPrevBlock);
        buffer.write(merkleRoot);
        buffer.write(Bytes.toBytes(timestamp));
        buffer.write(Bytes.toBytes(targetBits));
        buffer.write(Bytes.toBytes(nonce));

        return buffer.toByteArray();
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}
```

5 Objects and Data Structures

- Data abstraction, like in ***Command*** in ***commands.cli*** package that separate common function for command line functionality

```
public interface Command {
    String getHelp();

    String[] getParams();

    void run(String[] args);
}
```

- All class follow the law of Demeter, like
  - ***ProofOfWork*** class especially in methods calling the methods of object passed as an argument to ***ProofOfWork***

```
public int runProofOfWork() {
      …
blockHeader = block.setBlockHeader(nonce);

      …
}
```

  - ***netBlock*** in ***Block*** class

```
 public static Block newBlock(List<Transaction> transactions, byte[]
hashPrevBlock, int height) {
     Block block = new Block(transactions, hashPrevBlock, height);
     block.setRemainingData();

     return block;
 }
```

- Hybrid Class – half object and half data structure – like ***Block***, ***Transaction***, ***TransactionInput***, ***TransactionOutput*** and ***Wallet***.
- Data Structure, like ***MerkleTree***, ***MerkleNode*** and messages of network as ***BlockMessage***, ***VersionMessage*** and ***NullMessage.***

6 Error Handling

- Use unchecked exception, like
  - ***getBlock*** in ***BlocksDAO***

```
public Block getBlock(String blockHash) {
    String blockSerialized = dbConnection.get(blockHash);

    if (blockSerialized == null) {
        throw new NoSuchElementException("Block not found in database.");
    }
    return deserialize(blockSerialized);
}
```

  - ***generateKeyPair*** in ***Keys*** Utility

```
public static KeyPair generateKeyPair() {
    try {
        ECParameterSpec ecSpec = ECNamedCurveTable.getParameterSpec("secp256k1");
       KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("ECDSA",  "BC");
        keyPairGenerator.initialize(ecSpec, new SecureRandom());
        return keyPairGenerator.generateKeyPair();
    } catch (NoSuchAlgorithmException | NoSuchProviderException |
InvalidAlgorithmParameterException e) {
        throw new RuntimeException(e);
    }
}
```

7 Boundaries

- Using third-party code:
  - Gson to serialize/deserialize for network communication, store object – Block and TransacionOutput- in database and store wallets in file
    - In database, BlocksDAO and ChainStateDAO, respectively

```
private Block deserialize(String serializedBlock) {
    return new Gson().fromJson(serializedBlock, Block.class);
}

private String serialize(Block block) {
    return new Gson().toJson(block);
}
```

```
private String serializeOutputs(List<TransactionOutput> outputs) {
    return new Gson().toJson(outputs);
}

private List<TransactionOutput> deserializeOutputs(String outputSerialized)
{
    Type listTypeToken = new TypeToken<List<TransactionOutput>>() {
    }.getType();

    return new Gson().fromJson(outputSerialized, listTypeToken);
}
```

    - In network, as example to deserialize ***VersionMessage*** in ***VersionCommandHandler***

```
 VersionMessage remoteMessage = new Gson().fromJson(message,
VersionMessage.class);
```

  - Bouncycastle as Security provider
  - Jedis, to communicate with redis database

```
public String getTipOfChain() {
    return dbConnection.get(TIP_OF_CHAIN_KEY);
}
```

8 Classes

- Class organization managed by IntelliJ IDEA as possible as can, begin with a list of variables then public static constants then private static variables, followed by private instance variable Public functions follow list of variables then private utilities. As in ***Wallets***
- Encapsulation, minimize access to variables and utility function as possible as can, as shown like ***loadFromFile*** and ***saveToFile***.

```
public class Wallets {
    private static final String WALLETS_FILE = String.format("wallets%d.txt",
AtycoinStart.getNodeID());
    private static final Wallets instance = new Wallets();
    private final String directory;
    private Map<String, Wallet> wallets;

    private Wallets() {...}

    public static Wallets getInstance() {...}

    // adds a Wallet to Wallets
    public String createWallet() {...}

    // returns a list of addresses stored in the wallet file
    public List<String> getAddresses() {...}

    // returns a Wallet by its address
    public Wallet getWallet(String address) {...}

    // loads wallets from the file
    private void loadFromFile() {...}

    // saves wallets to a file
    private void saveToFile() {...}
}
```

- The class should be small as possible as can and have one reason to change – Single Responsibility Principle – like ***TransactionProcessor*** class that separated form ***Transaction*** and ***Blockchain*** classes that handle security issue of ***Transaction*** – Signing and Verification –.

```
public class TransactionProcessor {
    private Transaction transaction;
    private BlocksDAO blocksDAO;

     ...

    public boolean signTransaction(ECPrivateKey privateKey) {...}

    public boolean verifyTransaction() {...}
```

- Classes should have a small number of instance variables. Each of the methods of a class should manipulate one or more of those variables like in ***TransactionInput*** and ***TransactionOutput***

```
public class TransactionInput {
    private final byte[] referenceTransaction;
    private final int outputIndex;
    private byte[] rawPublicKey;
    private byte[] signature;

      ...
}
```

```
public class TransactionOutput {
    private final int value;
    private byte[] publicKeyHashed;

      ...
}
```

- Organize for change and Isolating form change, like in ***networkMessage*** and ***Commands*** handler that flexible to add new command in network and edit already exits one.

```
public abstract class NetworkMessage {
    private transient final static Gson encoder = new Gson();
    private final int senderAddress;

    public NetworkMessage(int senderAddress) {...}

    abstract public String makeRequest();

    protected final String serialize(String command, NetworkMessage message) {...}

    public int getSenderAddress() {...}
}
```

```
public abstract class NetworkCommand {
    private BufferedWriter output;

    abstract public void execute(String message, int nodeAddress);

    protected void send(int recipient, NetworkMessage message) {...}

    private void getOutputStream(Socket connection) throws IOException {...}
}
```

9 Systems

- Separate constructing a system from using it
  - Separation of ***main***, is separated in its class to setup and run application

```
public static void main(String[] args) {
    //Setup bouncy castle as security provider
    Security.addProvider(new BouncyCastleProvider());

    nodeID = Integer.parseInt(args[0]);
    System.out.printf("NodeID: %d%n", nodeID);

    Commander.getInstance().listen();
}
```

  - Factories – that responsible for creating the correct version of object – and Scaling up as in ***NetworkCommandFactory***

```
public class NetworkCommandFactory {
    private static final NetworkCommand NULL_COMMAND = new NullCommandHandler();
    private static final Map<String, NetworkCommand> commands;

    static {
        commands = new HashMap<>();
        initializeCommands();
    }

    private static void initializeCommands() {
        commands.put("addresses", new AddressesCommandHandler());
        commands.put("block", new BlockCommandHandler());
        commands.put("getaddresses", new GetAddressesCommandHandler());
        commands.put("getblocks", new GetBlocksCommandHandler());
        commands.put("getdata", new GetDataCommandHandler());
        commands.put("inventory", new InventoryCommandHandler());
        commands.put("transaction", new TransactionCommandHandler());
        commands.put("version", new VersionCommandHandler());
        commands.put("null", new NullCommandHandler());
    }

    public static NetworkCommand getCommand(String command) {
        NetworkCommand networkCommand = commands.get(command);

        if (networkCommand == null) {
            networkCommand = NULL_COMMAND;
        }
        return networkCommand;
    }
}
```

10 Concurrency

- Separate concurrency-related code from other code, like in
  - ***Node*** class to run network independently from the command line interface and using ***newSingleThreadExecutor*** to run it.
  - ***ConnectionHandler*** that responsible about received connection message after accept connection

```
public class ConnectionHandler implements Runnable {
    private final Socket receivingConnection;
    private final int nodeAddress;
    private BufferedReader input;

    public ConnectionHandler(Socket receivingConnection, int nodeAddress) {
        this.receivingConnection = receivingConnection;
        this.nodeAddress = nodeAddress;
    }

    @Override
    public void run() {
        try {
            getStreamInput();
            String request = input.readLine();
            String[] requestContent = request.split("\\s+");

            String command = requestContent[0];
            String message = requestContent[1];

            NetworkCommand networkCommand =
NetworkCommandFactory.getCommand(command);
            System.out.printf("Received %s command%n", command);

            networkCommand.execute(message, nodeAddress);
        } catch (IOException e) {
            stopConnection();
            throw new RuntimeException(e);
        } finally {
            stopConnection();
        }
    }

       ...
}
```

Part 2 Effective Java Code Principles

1 Creating and Destroying Objects

- Consider static factory methods instead of constructor, as in
  - ***Block***

```
public static Block newGenesisBlock(Transaction coinbase) {
   //add coinbase Transaction
   List<Transaction> transactions = new ArrayList<>();
   transactions.add(coinbase);

   return newBlock(transactions, Constant.EMPTY_BYTE_ARRAY, GENESIS_HEIGHT);
}

public static Block newBlock(List<Transaction> transactions, byte[] hashPrevBlock,
int height) {
   Block block = new Block(transactions, hashPrevBlock, height);
   block.setRemainingData();

   return block;
}
```

  - ***Transaction***

```
// creates a trimmed copy of Transaction to be used in signing and verifying
public static Transaction trimmedTransaction(Transaction transaction) {...}

//Reward to miner
public static Transaction newCoinbaseTransaction(String to) {...}

public static Transaction newTransaction(Wallet sender, String recipient, int amount)
{...}
```

  - ***TransactionOutput***

```
public static TransactionOutput newTransactionOutput(int value, String address) {
   TransactionOutput transactionOutput = new TransactionOutput(value,
Constant.EMPTY_BYTE_ARRAY);
   transactionOutput.lock(address);
   return transactionOutput;
}
```

- ***MerkleTree*** and ***MerkleNode***

```
public static MerkleTree newMerkleTree(List<List<Byte>> data) {...}
```

```
public static MerkleNode newMerkleNode(MerkleNode left, MerkleNode right,
byte[] data) {
```

- And all of the singleton class (list of singleton in class in Design pattern part).

- Enforce the singleton property with a private constructor, as in ***Wallets***

```
public class Wallets {
   private static final String WALLETS_FILE = String.format("wallets%d.txt",
AtycoinStart.getNodeID());
   private static final Wallets instance = new Wallets();
   private final String directory;
   private Map<String, Wallet> wallets;

   private Wallets() {
       directory = getClass().getResource("").getPath();
       loadFromFile();
   }

   public static Wallets getInstance() {
       return instance;
   }

      ...

}
```

- Enforce non-instantiability with a private constructor, as in

  - ***wallet***

```
private Wallet() {
}
```

  - ***Transaction***

```
private Transaction(List<TransactionInput> inputs, List<TransactionOutput>
outputs) {
    this.inputs = inputs;
    this.outputs = outputs;
    timestamp = System.currentTimeMillis() / 1000L; // Convert to Second
}
```

- ***Block***

```
private Block(List<Transaction> transactions, byte[] hashPrevBlock, int height)
{
    version = 1;
    timestamp = System.currentTimeMillis() / 1000L; // Convert to Second
    this.transactions = transactions;
    this.hashPrevBlock = hashPrevBlock;
    this.height = height;
}
```

- Avoid creating unnecessary objects, as in ***NetworkCommandFactory***, the null command loaded only once and refer to it when it needed.

```
private static final NetworkCommand NULL_COMMAND = new NullCommandHandler();
```

- The finalizers and cleaners are avoided because it unpredictable
- Prefer *Try-With-Resourses* to *Try-Finally* when manage **resources**, as in ***Node***

```
private void startServer() {
    try (ServerSocket server = new ServerSocket(nodeAddress)) {
        if (nodeAddress != KnownNodes.get(0)) {
            NetworkMessage version = new VersionMessage(NODE_VERSION,
nodeAddress);
            String request = version.makeRequest();

            connection = new Socket(InetAddress.getLocalHost(),
KnownNodes.get(0));

            getOutputStream();
            output.write(request);
            output.flush();

            output.close();
            connection.close();
        }

        while (true) {
            connection = server.accept();
            handleConnection(connection, nodeAddress);
        }
    } catch (IOException e) {
        pool.shutdown();
        throw new RuntimeException(e);
    }
}
```

2 Methods Common to All Objects

- Obey the general contract when *overriding* ***equals*** as in ***TransactionOutput***

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    TransactionOutput that = (TransactionOutput) o;
    return value == that.value &&
            Arrays.equals(publicKeyHashed, that.publicKeyHashed);
}
```

- Always *override* ***hashCode*** when you *override* ***equals*** as in ***TransactionOutput***

```
@Override
public int hashCode() {
    int result = Objects.hash(value);
    result = 31 * result + Arrays.hashCode(publicKeyHashed);
    return result;
}
```

- Always *override* ***toString***, as in ***TransactionOuput***

```
@Override
public String toString() {
    return new StringJoiner("\n")
            .add(String.format("\t\tValue: %d", value))
            .add(String.format("\t\tPublic Key Hashed: %s",
Bytes.toHex(publicKeyHashed)))
            .toString();
}
```

3 Classes and Interfaces

- Minimize the accessibility of classes and members
  - All of the instance variable are ***private***
  - All of the utility methods are ***private***, like

```
private void getOutputStream(Socket connection) throws IOException {
    output = new BufferedWriter(new
OutputStreamWriter(connection.getOutputStream()));
    output.flush();
}
```

  - ***serialize*** in ***NetworkMessage*** is ***protected***

```
protected final String serialize(String command, NetworkMessage message){
   String msg = encoder.toJson(message);
   return command + msg;
}
```

  - ***send*** in ***NetworkCommand*** is ***protected***

```
protected void send(int recipient, NetworkMessage message) {
    try (Socket sendingConnection = new
Socket(InetAddress.getLocalHost(), recipient)) {
        getOutputStream(sendingConnection);
        String request = message.makeRequest();
        output.write(request);
        output.flush();
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}
```

- Public classes have accessor methods not public fields, like in ***Block*** class

```
public byte[] getHash() {
   return hash;
}

public int getHeight() {
   return height;
}
```

- Minimize mutability
  - ***Wallet, TransactionOutput, MerkleTree, MerkleNode, Block*** and all of the network messages have not mutattor
  - ***Transaction*** just have mutattor for ***ID***.
  - ***Transaction*** in ***TransactionMessage*** is ***private final***

```
public class TransactionMessage extends NetworkMessage {
   private final Transaction transaction;

      ...

}
```

- Prefer interfaces to abstract classes when it is possible and , like ***Command*** in ***cli***

```
public interface Command {
   String getHelp();

   String[] getParams();

   void run(String[] args);
}
```

- Design interfaces for posterity as in ***NetworkCommand*** and ***NetworkMessage***
  - ***NetworkCommand***, sending over networks common to all the subclass but execute depend on what the subclass will do.

```
public abstract class NetworkCommand {
   private BufferedWriter output;

   abstract public void execute(String message, int nodeAddress);

   protected void send(int recipient, NetworkMessage message) {...}

   private void getOutputStream(Socket connection) throws IOException {...}
}
```

- ***NetworkMessage,*** have common serialization and each message has ***senderAddress*** to send response to it. But makeRequest message differ in command prefix and content of request message.

```
public abstract class NetworkMessage {
   private transient final static Gson encoder = new Gson();
   private final int senderAddress;

   public NetworkMessage(int senderAddress) {...}

   abstract public String makeRequest();

   protected final String serialize(String command, NetworkMessage message) {...}

   public int getSenderAddress() {...}
}
```

- Favor static member classes over non-static, as in Utilities
  - ***Address***
  - ***Base58***
  - ***Bytes***
  - ***Constant***
  - ***Hash***
  - ***Keys***
  - ***Signature***

- Limit source files to a single top-level class, there are no more than one class in each source file.

4 Enums and Annotations

- Consistently use the *Override* annotation, like
  - ***HelpCommand***

```
@Override
public void run(String[] args) {
   Commander.CommanderPrint("\
n--------------------------------------------------------------------------\n" +
           "-  ATYCOIN HELP\n" +

"--------------------------------------------------------------------------");

   Commander commander = Commander.getInstance();
   Collection<Command> commands = commander.getCommands();
   for (Command command : commands) {
       Commander.CommanderPrint(command.getHelp());
   }
}
```

  - ***AddressesCommandHandler***

```
@Override
public void execute(String message, int nodeAddress) {
   AddressesMessage remoteMessage = new Gson().fromJson(message,
AddressesMessage.class);
   List<Integer> newNodes = remoteMessage.getAddresses();

   for (int address : newNodes) {
       KnownNodes.add(address);
   }
}
```

  - ***NullMessage***

```
@Override
public String makeRequest() {
   String command = "null "; //space, Indicator to find command
   return serialize(command, this);
}
```

5 Methods

- Design method signatures carefully [clean code principles in part 1 cover this point very well].
- Use overloading judiciously, like ***toBytes*** in ***Bytes*** Utility (overloaded methods are static).

```
public static byte[] toBytes(long input) {
   ByteBuffer byteBuffer = ByteBuffer.allocate(Long.BYTES);
   byteBuffer.putLong(input);
   return byteBuffer.array();
}

public static byte[] toBytes(int input) {
   ByteBuffer byteBuffer = ByteBuffer.allocate(Integer.BYTES);
   byteBuffer.putInt(input);
   return byteBuffer.array();
}

public static byte[] toBytes(List<Byte> list) {
   int listLength = list.size();
   byte[] data = new byte[listLength];
   for (int i = 0; i < listLength - 1; i++) {
       data[i] = list.get(i);
   }
   return data;
}
```

- Return empty collections or arrays, not null. Like

```
private Map<String, Transaction> getReferenceTransactions() {
   Map<String, Transaction> referenceTransactions = new HashMap<>();
   for (TransactionInput input : transaction.getInputs()) {
       byte[] referenceTransactionID = input.getReferenceTransaction();
       Transaction referenceTransaction =
findTransaction(referenceTransactionID);
       if (referenceTransaction == null) {
           System.out.println("Transaction is not Found");
           return null;
       }
       String key = Hash.serialize(referenceTransactionID);
       referenceTransactions.put(key, referenceTransaction);
   }
   return referenceTransactions;
}
```

6 General Programming

- Minimize the scope of local variables
  - by declare them where it is first used and should contain an initialization, like in ***InventoryCommandHandler***

```
@Override
public void execute(String message, int nodeAddress) {

      ...

   NetworkMessage responseMessage = new NullMessage(nodeAddress);
   if (type.equals("block")) {
       //track downloaded blocks
       BlocksInTransit.addItems(items);

       String blockHash = items.get(items.size() - 1);
       responseMessage = new GetDataMessage(nodeAddress, "block", blockHash);

       BlocksInTransit.removeItem(items.size() - 1);
   } else if (type.equals("tx")) {
       String txId = items.get(0);

       if (Mempool.getItem(txId) == null) {
           responseMessage = new GetDataMessage(nodeAddress, "tx", txId);
       }
   }

   send(remoteMessage.getSenderAddress(), responseMessage);
}
```

  - Prefer ***for*** over ***while*** loop, as in ***newMerkleTree*** in ***MerkleTree***

```
for (int left = 0, right = left + 1; left < numberOfNodes; left += 2) {
   if (right == numberOfNodes) {
       right = left;
   }

   MerkleNode leftNode = nodes.get(left);
   MerkleNode rightNode = nodes.get(right);

   MerkleNode node = MerkleNode.newMerkleNode(leftNode, rightNode,
Constant.EMPTY_BYTE_ARRAY);

   newLevel.add(node);
}
```

- Prefer ***For-Each*** loops over traditional ***for*** loops because it have more clarity, flexibility and bug prevention. Like ***getBalance*** in ***ChainState***

```
public int getBalance(byte[] publicKeyHashed) {
    int balance = 0;
    Set<String> transactionIDs = chainStateDAO.getAllReferenceTransaction();
    for (String transactionID : transactionIDs) {
        List<TransactionOutput> outputs =
chainStateDAO.getReferenceOutputs(transactionID);
        for (TransactionOutput output : outputs) {
            if (output.isLockedWithKey(publicKeyHashed)) {
                balance += output.getValue();
            }
        }
    }
    return balance;
}
```

- Known and use the libraries, like
  - ***applaySHA256*** in ***Hash*** utility

```
public static byte[] applySHA256(byte[] data) {
    MessageDigest digest = new SHA256.Digest();
    return digest.digest(data);
}
```

  - ***sign*** in ***Signature***

```
public static byte[] sign(ECPrivateKey privateKey, byte[] data) {
    try {
        java.security.Signature signature =
java.security.Signature.getInstance("ECDSA", "BC");
        signature.initSign(privateKey);
        signature.update(data);
        return signature.sign();
    } catch (NoSuchAlgorithmException | SignatureException |
InvalidKeyException | NoSuchProviderException e) {
        throw new RuntimeException(e);
    }
}
```

- ***toPublicKey*** in Keys utility (using the ***ByteArrayOutputStream*** to convert public key to encoded form to extract ECPublikcKey object)

```
public static ECPublicKey toPublicKey(byte[] rawPublicKey) {
   // Get RawPublicKey (0x04 + xAffineCoord + yAffineCoord)
   ByteArrayOutputStream buffer = new ByteArrayOutputStream();
   buffer.write(0x04);
   rawPublicKey = Hex.decode(rawPublicKey);
   buffer.write(rawPublicKey, 0, rawPublicKey.length);
}
```

- The use of ***BigInteger*** to encode/decode addresses in ***Base58*** form

```
public static byte[] decode(String address) {
   if (address.isEmpty()) {
       return Constant.EMPTY_BYTE_ARRAY;
   }

   BigInteger sumOfDigits = BigInteger.ZERO;

   // sumOfDigits * 58 + indexOf(digit)
   char[] addressDigits = address.toCharArray();
   for (char digit : addressDigits) {
       sumOfDigits = sumOfDigits.multiply(BASE58);

       int index = ALPHABET.indexOf(digit);
       BigInteger valueOfIndex = BigInteger.valueOf(index);
       sumOfDigits = sumOfDigits.add(valueOfIndex);
   }

      ...
```

- Avoid ***float*** and ***double*** if exact answers are required, like in value to store coin amount and use sub-units to solve fraction value

```
public class TransactionOutput {
   private final int value;

      ...
```

- Prefer primitive types to boxed primitive as possible as can, all the variable are primitive type with exception in calculate ***MerkleTree*** to store list of ***Byte array***

```
public static MerkleTree newMerkleTree(List<List<Byte>> data) {
   List<MerkleNode> nodes = new ArrayList<>();

   for (List<Byte> datum : data) {
       byte[] datumBytes = Bytes.toBytes(datum);
       MerkleNode node = MerkleNode.newMerkleNode(null, null, datumBytes);
       nodes.add(node);
   }
}
```

- Avoid ***strings*** where other types are more appropriate, like
  - using byte array to store hash value

```
public class Block {

      ...

   private final byte[] hashPrevBlock;

      ...

   private byte[] merkleRoot;
   private int nonce;
   private byte[] hash;

      ...
```

  - use ***StringJoiner*** *t*o get ***String*** representation of a ***Transaction*** object

```
public String toString() {
   StringJoiner stringJoiner = new StringJoiner("\n", "", "\n");
   stringJoiner.add(String.format("--- Transaction %s:", Bytes.toHex(id)));

   TransactionInput input;
   for (int i = 0, size = inputs.size(); i < size; i++) {
       input = inputs.get(i);
       stringJoiner.add(String.format("\tInput %d:", i));
       stringJoiner.add(input.toString());
   }

   TransactionOutput output;
   for (int i = 0, size = outputs.size(); i < size; i++) {
       output = outputs.get(i);
       stringJoiner.add(String.format("\tOutput %d:", i));
       stringJoiner.add(output.toString());
   }
   return stringJoiner.toString();
}
```

- Beware the performance of string concatenation by using another method as getting ***Transaction*** in ***String*** format as seen in the previous point
- Refer to objects by their interfaces
  - ***networkCommand*** in ***ConnectionHandler***

```
public void run() {
   try {
       getStreamInput();
       String request = input.readLine();
       String[] requestContent = request.split("\\s+");

       String command = requestContent[0];
       String message = requestContent[1];

       NetworkCommand networkCommand = NetworkCommandFactory.getCommand(command);
       System.out.printf("Received %s command%n", command);

       networkCommand.execute(message, nodeAddress);
   } catch (IOException e) {
       stopConnection();
       throw new RuntimeException(e);
   } finally {
       stopConnection();
   }
}
```

  - ***responseMessage*** in ***GetDataCommandHandler***

```
public void execute(String message, int nodeAddress) {

      ...

   NetworkMessage responseMessage = new NullMessage(nodeAddress);
   String itemID = remoteMessage.getId();
   if (type.equals("block")) {
       Block block = BlocksDAO.getInstance().getBlock(itemID);
       responseMessage = new BlockMessage(nodeAddress, block);
   } else if (type.equals("tx")) {
       Transaction transaction = Mempool.getItem(itemID);
       responseMessage = new TransactionMessage(nodeAddress, transaction);
   }

   send(remoteMessage.getSenderAddress(), responseMessage);
}
```

- ***wallets*** in ***Wallets*** class

```
private Map<String, Wallet> wallets;
```

- ***command*** in ***Commander***

```
private void call(String[] rawArgs) {
    String function = rawArgs[0];
    String[] args = Arrays.copyOfRange(rawArgs, 1, rawArgs.length);

    Command command = commands.get(function);
    if (command == null) {
        CommanderPrint("command function: '" + function + "' not found. Type
help for a list of functions");
    } else {
        command.run(args);
    }
}
```

7 Exceptions

- Use Exceptions only for exceptional conditions and favor standard exception, like
  - ***BlocksDAO***

```
public Block getBlock(String blockHash) {
   String blockSerialized = dbConnection.get(blockHash);

   if (blockSerialized == null) {
       throw new NoSuchElementException("Block not found in database.");
   }
   return deserialize(blockSerialized);
}
```

  - ***HelpCommand***

```
@Override
public String[] getParams() {
   throw new UnsupportedOperationException("Not Supported");
}
```

- Use Runtime exception for programming error, like in ***Commander*** to exit from the application if the database not exist or have problem in connection.

```
public void listen() {

      ...

           if (!debugMode) {
               try {
                   call(rawArgs);
               } catch (ArrayIndexOutOfBoundsException e) {
                   CommanderPrint("command couldn't execute, perhaps not enough
arguments? try: " + rawArgs[0] + " -help");
               } catch (JedisConnectionException e) {
                   CommanderPrint("Failed connecting to database");
                   System.exit(0);
               } catch (Exception e) {
                   CommanderPrint("command failed to execute.");
               }
           } else { //Otherwise run the command with no safety net.
               call(rawArgs);
           }
}
```

- Use ***RuntimeException*** for programming error, like problem in ***Socket*** connection

```
protected void send(int recipient, NetworkMessage message) {
   try (Socket sendingConnection = new Socket(InetAddress.getLocalHost(), recipient))
{
       getOutputStream(sendingConnection);
       String request = message.makeRequest();
       output.write(request);
       output.flush();
   } catch (IOException e) {
       throw new RuntimeException(e);
   }
}
```

- Avoid unnecessary use of checked exceptions, the application does not throw ***checked*** exception.

- Don't ignore exceptions by provide empty catch statement, instead of using empty catch I throw ***RuntimeError***, as in ***TransactionInput*** and ***TransactionOutput***, respectively.

```
public byte[] concatenateData() {
   ByteArrayOutputStream buffer = new ByteArrayOutputStream();
   try {
       buffer.write(referenceTransaction);
       buffer.write(Bytes.toBytes(outputIndex));
       buffer.write(rawPublicKey);
       return buffer.toByteArray();
   } catch (IOException e) {
       throw new RuntimeException(e);
   }
}
```

```
public byte[] concatenateData() {
   ByteArrayOutputStream buffer = new ByteArrayOutputStream();
   try {
       buffer.write(Bytes.toBytes(value));
       buffer.write(publicKeyHashed);
       return buffer.toByteArray();
   } catch (IOException e) {
       throw new RuntimeException(e);
   }
}
```

8 Concurrency

- Prefer executor, tasks, and streams to thread
  - In ***StartNodecommand*** class, to start network layer for the current node separately form the command line interface

```
public class StartNodeCommand implements Command {
      private static final ExecutorService pool =
Executors.newSingleThreadExecutor();

      ...

      @Override
      public void run(String[] args) {

            ...

      pool.execute(Node.getInstance(minerAddress));
```

  - In ***Node*** class, to handle incoming connection

```
public class Node implements Runnable {

      ...

   private final ExecutorService pool;

   private Node(int nodeAddress, String minerAddress) {
      this.nodeAddress = nodeAddress;
      miningAddress = minerAddress;
      pool = Executors.newFixedThreadPool(8);
   }


   private void handleConnection(Socket connection, int nodeAddress) {
      pool.execute(new ConnectionHandler(connection, nodeAddress));
   }
}
```

9 Serialization

- I used JSON as an alternatives to solve security issues of java serialization through using GSON library

```
private Block deserialize(String serializedBlock) {
   return new Gson().fromJson(serializedBlock, Block.class);
}

private String serialize(Block block) {
   return new Gson().toJson(block);
}
```

- Deserialize complex data structure like ***map*** to store wallet in ***Wallets***

```
private void loadFromFile() {

      ...

      else {
            Gson decoder = new Gson();
            Type mapTypeToken = new TypeToken<Map<String, Wallet>>() {
            }.getType();

            wallets = decoder.fromJson(deserializeWallets, mapTypeToken);
      }
}
```

Part 3: Design Pattern

1 Data Access Object Pattern

- To separate access of data from the application of two database:

1. Blocks

Key	Value
ID of block (Hash code)	Block object serialized using Json
l	Hash of last block that added to blockchain (***tipOfchain***)
h	Best height of blockchain

Important point:

- Connection to database

```
private BlocksDAO(int port) {
    dbConnection = new Jedis("localhost", port);
}
```

Note: Currently Connection pool have dependency problem from the provider of ***slf4j*** logger package because another package use a different version than ***jedis.***

- ***addBlock*** that add new block and update ***tipOfchain*** and ***bestHeight*** using redis transaction – Manual commit of these data –.
- ***getTipOfChain***

```
public String getTipOfChain() {
    return dbConnection.get(TIP_OF_CHAIN_KEY);
}
```

- ***getBlock***
- ***getBestHeight***
- ***getBlockHashes*** to get all blocks hashes using iterator

```
    ...
for (Block block : this) {
    if (requiredHeight != 0 && block.getHeight() <= requiredHeight) {
        break;
    }
    blockHashes.add(Hash.serialize(block.getHash()));
}
```

2. Chain state

Key	Value
ID of Transaction (Hash code)	List of unspent transactions outputs

Important point:

- Connection to database

```
private ChainStateDAO(int port) {
    dbConnection = new Jedis("localhost", port);
    dbConnection.select(CHAIN_STATE_DB);
}
```

- ***setUnspentOutputs***

```
public void setUnspentOutputs(byte[] id, List<TransactionOutput> outputs) {
    String key = getKey(id);
    String value = serializeOutputs(outputs);
    dbConnection.set(key, value);
}
```

- ***deleteUnspentOutputs***

```
public void deleteUnspentOutputs(byte[] id) {
    String key = getKey(id);
    dbConnection.del(key);
}
```

- ***getAllRefernceTransaction***

```
public Set<String> getAllReferenceTransaction() {
    return dbConnection.keys("*");
}
```

- ***getReferanceOutputs***

```
public List<TransactionOutput> getReferenceOutputs(byte[] id) {
    String key = getKey(id);
    return getReferenceOutputs(key);
}

public List<TransactionOutput> getReferenceOutputs(String key) {
    String outputsSerialized = dbConnection.get(key);
    return deserializeOutputs(outputsSerialized);
}
```

2 Factory Pattern

- Used to

1. Get correct response (commands) for a specific message in network using ***HashMap*** and abstract class (NetworkCommand)

```
public class NetworkCommandFactory {
    private static final NetworkCommand NULL_COMMAND = new
NullCommandHandler();
    private static final Map<String, NetworkCommand> commands;

    static {
        commands = new HashMap<>();
        initializeCommands();
    }

    private static void initializeCommands() {
        commands.put("addresses", new AddressesCommandHandler());
        commands.put("block", new BlockCommandHandler());
        commands.put("getaddresses", new GetAddressesCommandHandler());
        commands.put("getblocks", new GetBlocksCommandHandler());
        commands.put("getdata", new GetDataCommandHandler());
        commands.put("inventory", new InventoryCommandHandler());
        commands.put("transaction", new TransactionCommandHandler());
        commands.put("version", new VersionCommandHandler());
        commands.put("null", new NullCommandHandler());
    }

    public static NetworkCommand getCommand(String command) {
        NetworkCommand networkCommand = commands.get(command);

        if (networkCommand == null) {
            networkCommand = NULL_COMMAND;
        }
        return networkCommand;
    }
}
```

2. Execute command in command-line interface using ***HashMap*** and ***Command*** Interface

```
private void setup() {
   commands = new HashMap<>();
   commands.put("createblockchain", new CreateBlockchainCommand());
   commands.put("createwallet", new CreateWalletCommand());
   commands.put("debug", new DebugModeCommand());
   commands.put("getbalance", new GetBalanceCommand());
   commands.put("help", new HelpCommand());
   commands.put("listaddresses", new ListAddressesCommand());
   commands.put("printchain", new PrintChainCommand());
   commands.put("send", new SendCommand());
   commands.put("startnode", new StartNodeCommand());
   scanner = new Scanner(System.in);
}
```

3 Strategy Pattern

1. ***NetworkCommand*** and its concrete implementation, execution changed at run time

```
public abstract class NetworkCommand {
    private BufferedWriter output;

    abstract public void execute(String message, int nodeAddress);

    protected void send(int recipient, NetworkMessage message) {
        try (Socket sendingConnection = new Socket(InetAddress.getLocalHost(),
recipient)) {
            getOutputStream(sendingConnection);
            String request = message.makeRequest();
            output.write(request);
            output.flush();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }

    private void getOutputStream(Socket connection) throws IOException {
        output = new BufferedWriter(new
OutputStreamWriter(connection.getOutputStream()));
        output.flush();
    }
}
```

As example of concrete implementation of ***NetworkCommand***

```
public class VersionCommandHandler extends NetworkCommand {
    @Override
    public void execute(String message, int nodeAddress) {
        int localBestHeight = BlocksDAO.getInstance().getBestHeight();

        VersionMessage remoteMessage = new Gson().fromJson(message,
VersionMessage.class);
        int remoteBestHeight = remoteMessage.getBestHeight();

        NetworkMessage responseMessage = new NullMessage(nodeAddress);

        if (localBestHeight < remoteBestHeight) {
            responseMessage = new GetBlocksMessage(nodeAddress, localBestHeight);
        } else if (localBestHeight > remoteBestHeight) {
            int version = remoteMessage.getVersion();
            responseMessage = new VersionMessage(version, nodeAddress);
        }

        KnownNodes.add(remoteMessage.getSenderAddress());

        send(remoteMessage.getSenderAddress(), responseMessage);

        NetworkMessage getKnownNodesMessage = new GetAddressesMessage(nodeAddress);
        send(remoteMessage.getSenderAddress(), getKnownNodesMessage);
    }
}
```

2. ***Command*** interface and its concrete implementation in ***cli***

```
public interface Command {
    String getHelp();

    String[] getParams();

    void run(String[] args);
}
```

```
public class ListAddressesCommand implements Command {
    @Override
    public String getHelp() {
        return "cmd: listaddresses \n" +
                "- description: Lists all addresses from the wallet file \n" +
                "- usage: listaddresses param [situational...] \n" +
                "- param: '-help', '-params' \n" +

"------------------------------------------------------------------------";
    }

    @Override
    public String[] getParams() {
        return new String[]{"-help", "-params"};
    }

    @Override
    public void run(String[] args) {
        if (args.length < 1) {
            Wallets wallets = Wallets.getInstance();
            List<String> addresses = wallets.getAddresses();

            for (String address : addresses) {
                Commander.CommanderPrint(address);
            }
            return;
        }

        String[] params = getParams();

        //check if command exists in params list
        if (!Arrays.asList(params).contains(args[0])) {
            Commander.CommanderPrint("ERROR ! unknown parameters...");
            Commander.CommanderPrint(Arrays.toString(params));
            return;
        }

        if (args[0].equals(params[0])) { //-help
            Commander.CommanderPrint(getHelp());
        } else if (args[0].equals(params[1])) { //-params
            Commander.CommanderPrint(Arrays.toString(params));
        } else {
            Commander.CommanderPrint("Invalid parameter. Enter 'listaddresses -
help'");
        }
    }
}
```

4 Null Object Pattern

- As in ***NullCommandHandler*** as response on ***NullMessage*** reguest

```
public class NullCommandHandler extends NetworkCommand {
    @Override
    public void execute(String message, int nodeAddress) {
        //nothing to execute
    }
}
```

Example of usage:

- ***excute*** method in ***VersionCommandHandler***

```
NetworkMessage responseMessage = new NullMessage(nodeAddress);

if (localBestHeight < remoteBestHeight) {
    responseMessage = new GetBlocksMessage(nodeAddress, localBestHeight);
} else if (localBestHeight > remoteBestHeight) {
    int version = remoteMessage.getVersion();
    responseMessage = new VersionMessage(version, nodeAddress);
}
```

5 Singleton Pattern

Used in

- Commander
- Blockchain
- BlocksDAO
- ChainState
- ChainStateDAO
- Node
- Wallets

to handle resources in the above Object (most of them discussed previously)

6 Iterator Pattern

To get an access to the block stored in blocks database and to implement ***Iterable*** interface to use for-each loop

```
public class BlocksIterator implements Iterator<Block> {
    private static final String GENESIS_PREVIOUS_HASH =
Hash.serialize(Constant.EMPTY_BYTE_ARRAY);
    private final BlocksDAO blocksDAO;
    private String currentHashSerialized;

    public BlocksIterator() {
        blocksDAO = BlocksDAO.getInstance();
        currentHashSerialized = blocksDAO.getTipOfChain();
    }

    @Override
    public boolean hasNext() {
        if (currentHashSerialized == null) {
            return false;
        }
        return !currentHashSerialized.equals(GENESIS_PREVIOUS_HASH);
    }

    @Override
    public Block next() {
        Block block = blocksDAO.getBlock(currentHashSerialized);

        // get next blockHash
        currentHashSerialized = Hash.serialize(block.getHashPrevBlock());
        return block;
    }
}
```

```
public class BlocksDAO implements Iterable<Block> {

      ...

      @Override
      public Iterator<Block> iterator() {
          return new BlocksIterator();
      }

      ...

}
```

Part 4: Data Structures

1. ***commands*** in ***Commander*** and ***NetworkCommandFactory*** are ***Map*** to retrieve and create object by O(1).
2. ***blocksInTransit*** in ***BlockInTransit*** is list of String to track downloading of block.
3. ***knownNodes*** in ***KnownNodes*** is list of Integer just for simulate multiple node concept using port. (The correct choose is map that store IP as key and last available time to get known nodes by O(1) and remove nodes depend on last connection time.
4. ***mempool*** in ***Mempool*** to store Transaction by its hash and retrieve Transaction by O(1)
5. ***transactions*** in ***Block*** is list, I choose list because when I make new block I iterate over each transaction to check it. Also the 0 index always coinbase Transaction. But to improve update searching for Transaction in blockchain I must migrate to ***LinkedHashMap***.
6. ***MerkleTree***,

   Why used:

   - Significantly reduces the amount of data that a trusted authority has to maintain to proof the integrity of the transaction.
   - Significantly reduces the network I/O packet size to perform consistency and data verification as well as data synchronization.
   - Separate the validation of the data from the data itself.

7. ***inputs*** and ***outputs*** in ***Transaction*** are list
   - There are no need to store it as key – value pair
   - outputs is list because TransactionInput designed to store reference to TransactionOutput using transaction hash where transactionOutput reside and its index in the output list.
   - At most, there are two output element one for recipient of coin and another as change for the sender of the coin.
8. ***wallets*** in ***Wallets*** is ***map*** to store and retrieve wallet by its public key.