Assignment 2
(Due: Wednesday, March 27, 2024)

This is a programming assignment to help make you familiar with some aspects of the Bitcoin blockchain. Note that many things here are different from the Bitcoin protocol. This is just a simplified version.
The starting code is here.

Acknowledgement: This assignment is an adapted version of assignments provided by the Princeton Bitcoin book authors.

Part 1 - Transaction Handling

In this part, you will implement a simple transaction handler.

Specifications:

Transaction validation rules

Your transaction handler will receive a set of input transactions and is required to return a set of mutually valid transactions. The output set of transactions can be used to construct a block. (In part 2, you will be able to reuse your code for other purposes).
In addition to the transaction validation rules written in the javadoc, please note the following:

1. A transaction can spend the output of another transaction in the same block. Additionally, note that the received set of transactions could be **unordered**.
2. Among the transactions that the node receives, some transactions may try to spend the same output, i.e. double spend an output. This should not be allowed; only one transaction can spend an output. This means that transactions cannot be validated in isolation.
3. The constructed block does not need to be the largest block possible, however the main requirement is that the block should be a mutually valid transaction set of *maximal* size, i.e. a set that cannot be enlarged by any other transaction from the received set.

Transaction class

A transaction has a list of inputs, and a list of outputs. The **Transaction** class that has the following two inner classes:

- A **Transaction.Output** consists of a value and a public key to which it is being paid. For the public keys, the built-in Java PublicKey interface is used.

- A **Transaction.Input** consists of the hash of the transaction that contains the corresponding output, the index of that output in that transaction (indices are simply integers starting from 0), and a digital signature.

    - For a transaction input to be valid, the signature it contains must be a valid signature over the current transaction with respect to the public key in the spent output.
    - To verify a signature:
        - You can use the following method
          `public static boolean verifySignature(PublicKey pubKey, byte[] message,byte[] signature)`
          , which is provided in the Crypto.java class.
        - This method receives three inputs: the public key from the spent output, the signed raw data, and the signature. This method returns true only if the signature is valid.
        - To obtain the raw data for a specific input, you can call **`getRawDataToSign(int index)`** provided in the Transaction class. The index here is the index of the input.
        - Note that your code only needs to verify signatures. When we test your code, the computation of signatures will be done outside the Transaction class by an entity that has the private keys.

The Transaction class also contains methods to add and remove an input, add an output, compute digests to sign/hash, add a signature to an input, and compute and store the hash of the transaction once all inputs/outputs/signatures have been added.

UTXO

This class represents an unspent transaction output. A UTXO contains the hash of the transaction from which it originates as well as its index within that transaction. The class has customized implementations for equals, hashCode, and compareTo methods, which allow the testing of equality and comparison between two UTXOs based on their indices and the contents of their `txHash` arrays.

UTXOPool

This class represents the current set of outstanding UTXOs and contains a map from each UTXO to its corresponding transaction output. This class contains constructors to create a new empty UTXOPool or a copy of a given UTXOPool, and methods to add and remove UTXOs from the pool, get the output corresponding to a given UTXO, check if a UTXO is in the pool, and get a list of all UTXOs in the pool.

Requirements

Your task in this part will be to implement TxHandler.java according to the specifications in its java doc. Make sure to take the block constraints (Page 1) into account when implementing handleTxs()

```
public class TxHandler {

        /** Creates a public ledger whose current UTXOPool (collection of unspent
        * transaction outputs) is utxoPool. This should make a defensive copy of
        * utxoPool by using the UTXOPool(UTXOPool uPool) constructor. */
        public TxHandler(UTXOPool utxoPool) { }

        /** Returns true if
        * (1) all outputs claimed by tx are in the current UTXO pool,
        * (2) the signatures on each input of tx are valid,
        * (3) no UTXO is claimed multiple times by tx,
        * (4) all of tx's output values are non-negative, and
        * (5) the sum of tx's input values is greater than or equal to the sum of
        its output values; and false otherwise. */
        public boolean isValidTx(Transaction tx) { }

        /** Handles each epoch by receiving an unordered array of proposed
        * transactions, checking each transaction for correctness,
        * returning a mutually valid array of accepted transactions,
        * and updating the current UTXO pool as appropriate.
        */
        public Transaction[] handleTxs(Transaction[] possibleTxs) { }

}
```

Part 2 - Blockchain

In this part, you will implement a node that maintains a blockchain. The node can be thought of as part of a blockchain-based consensus protocol, where a node could receive transactions and blocks from any other node, and update its view accordingly.

Specifications

Additional Classes and Notes

Block.java	Stores the block data structure.
BlockHandler.java	Uses BlockChain.java to process a newly received block, create a new block, or process a newly received transaction.
ByteArrayWrapper.java	A utility file which creates a wrapper for byte arrays such that it could be used as a key in hash functions. (See TransactionPool.java)
Transaction.java	As before. Note that there is a functionality to create a coinbase transaction. Take a look at the Block.java constructor to see how a coinbase transaction is created.
TransactionPool.java	Implements a pool of transactions, required when creating a new block.
TxHandler.java	The class you implemented in the previous part. Hint: It might help to add a public method getUTXOPool() in the TxHandler.java.

Requirements

Your task in this part is to implement the BlockChain class. This class is responsible for maintaining a blockchain. Since the entire blockchain could be huge in size, you should only keep around the most recent blocks.

Since there can be (multiple) forks, **blocks form a tree rather than a list**. Your design should take this into account. You have to maintain a UTXO pool corresponding to every block on top of which a new block might be created.

```
//BlockChain should maintain only limited block nodes to satisfy the functions
//You should not have all the blocks added to the block chain in memory
//as it would cause a memory overflow.

public class BlockChain {
	public static final int CUT_OFF_AGE = 10;

	/**
	 * create an empty block chain with just a genesis block. Assume
	 * {@code genesisBlock} is a valid block
	 */
	public BlockChain(Block genesisBlock) {
		// IMPLEMENT THIS
	}

	/** Get the maximum height block */
	public Block getMaxHeightBlock() {
		// IMPLEMENT THIS
	}

	/** Get the UTXOPool for mining a new block on top of max height block */
	public UTXOPool getMaxHeightUTXOPool() {
		// IMPLEMENT THIS
	}

	/** Get the transaction pool to mine a new block */
	public TransactionPool getTransactionPool() {
		// IMPLEMENT THIS
	}

	/**
	 * Add {@code block} to the blockchain if it is valid. For validity, all
	 * transactions should be valid and block should be at
	 * {@code height > (maxHeight - CUT_OFF_AGE)}, where maxHeight is the
	 * current height of the blockchain.
	 * <p>
	 * Assume the Genesis block is at height 1. For example, you can try creating a
```

```
     * new block over the genesis block (i.e. create a block at height 2) if the
     * current blockchain height is less than or equal to CUT_OFF_AGE + 1. As
     * soon as the current blockchain height exceeds CUT_OFF_AGE + 1, you cannot
     * create a new block at height 2.
     *
     * @return true if block is successfully added
     */
    public boolean addBlock(Block block) {
        // IMPLEMENT THIS
    }

    /** Add a transaction to the transaction pool */
    public void addTransaction(Transaction tx) {
        // IMPLEMENT THIS
    }
}
```

Assumptions/Hints

- A new genesis block won't be mined. If you receive a block which claims to be a genesis block (parent is a null hash) in the addBlock(Block b) function, you can return false.
- If there are multiple blocks at the same height, return the oldest block in getMaxHeightBlock() function.
- Assume for simplicity that a coinbase transaction of a block is available to be spent in the next block mined on top of it. (This is contrary to the actual Bitcoin protocol when there is a "maturity" period of 100 confirmations before it can be spent).
- Maintain only one global Transaction Pool for the blockchain and keep adding transactions to it when receiving transactions, and remove transactions from it if a new block is received or created. It's okay if some transactions get dropped during a blockchain reorganization, i.e., when a side branch becomes the new longest branch. Specifically, transactions present in the original main branch (and thus removed from the transaction pool) but absent in the side branch might get lost.
- The coinbase value is kept constant at 25 bitcoins whereas in reality it halves roughly every 4 years.
- When checking for validity of a newly received block, just checking if the transactions form a valid set is enough. The set does not have to be a maximum possible set of transactions. Also, you do not need to do any proof-of-work checks.

Additional exercise

- The way the hash of the COINBASE transactions is computed in the provided code could lead to an issue. Identify the problematic scenario, and search for how the actual implementation of Bitcoin prevents it.

Deliverables and Submission Requirements

- **A zip file containing:** your implementation of TxHandler.java and BlockChain.java only. Please don't rename the files or the required methods. You can add private methods and inner classes in these two classes, but don't change or include the other files.

- The name of your file should be: **asg2_id_name.zip** (id should be your student id).
  The zip file should only contain TxHandler.java and BlockChain.java.
  The submission forms will be posted later.

- Provide the answer of the above exercise as a comment in the Blockchain.java class.
- Please follow all the academic integrity guidelines. You are expected to write the code and answer the exercise individually without any unauthorized help and without checking online implementations that solve the same problems.

  In order for your assignment to be graded, please include the following statement as comments in the two files that you are submitting.

  "I acknowledge that I am aware of the academic integrity guidelines of this course, and that I worked on this assignment independently without any unauthorized help with coding or testing." - <Name>