

# Bitcoin and Cryptocurrency Technologies

## Assignment 1: ScroogeCoin

In ScroogeCoin (Lecture 1), the central authority Scrooge receives transactions from users. You will implement the logic used by Scrooge to **process transactions** and **produce the ledger**. Scrooge organizes transactions into time periods or blocks. In **each block, Scrooge will receive a list of transactions**, validate the transactions he receives, and publish a list of validated transactions.

Note that a transaction can reference another in the same block. Also, among the transactions received by Scrooge in a single block, more than one transaction may spend the same output. This would of course be a **double-spend**, and hence invalid. This means that transactions can't be validated in isolation; it is a tricky problem to choose a subset of transactions that are *together* valid.

You will be provided with a `Transaction` class that represents a ScroogeCoin transaction and has inner classes `Transaction.Output` and `Transaction.Input`.

A **transaction output** consists of a value and a public key to which it is being paid. For the public keys, we use the built-in Java [`PublicKey`](#) class.

A **transaction input** consists of the hash of the transaction that contains the corresponding output, the index of this output in that transaction (indices are simply integers starting from 0), and a digital signature. For the input to be valid, the signature it contains must be a valid signature over the current transaction with the public key in the spent output.

More specifically, the **raw data that is signed** is obtained from the `getRawDataToSign(int index)` method. To **verify a signature**, you will use the `verifySignature()` method included in the provided file `Crypto.java`:

```
public static boolean verifySignature(PublicKey pubKey, byte[] message,
byte[] signature)
```

This method takes a public key, a message and a signature, and returns true if and only signature correctly verifies over message with the public key `pubKey`.

Note that you are only given code to verify signatures, and this is all that you will need for this assignment. The computation of signatures is done outside the `Transaction` class by an entity that knows the appropriate private keys.

A transaction consists of a list of inputs, a list of outputs and a unique ID (see the `getRawTx()` method). The 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.

You will also be provided with a **UTXO** class that 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. We have included `equals`, `hashCode`, and `compareTo` functions in UTXO that allow the testing of equality and comparison between two UTXOs based on their indices and the contents of their `txHash` arrays.

Further, you will be provided with a **UTXOPool** class that 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.

You will be responsible for creating a file called `TxHandler.java` that implements the following API:

```
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);
}
```

Your implementation of `handleTxs()` should return a mutually valid transaction set of maximal size (one that can't be enlarged simply by adding more transactions). It need not compute a set of maximum size (one for which there is no larger mutually valid transaction set).

Based on the transactions it has chosen to accept, `handleTxs` should also update its internal `UTXOPool` to reflect the current set of unspent transaction outputs, so that future calls to `handleTxs()` and `isValidTx()` are able to correctly process/validate transactions that claim outputs from transactions that were accepted in a previous call to `handleTxs()`.

**Extra Credit:** Create a second file called `MaxFeeTxHandler.java` whose `handleTxs()` method finds a set of transactions with maximum total transaction fees -- i.e. maximize the sum over all transactions in the set of (sum of input values - sum of output values)).