CSC 212 Programming Assignment # 4
Implementing a Simplified Blockchain.
Due date: 11/4/2019
Bonus: +1. Plagiarism: -9.

Guidelines:	This is an **individual** assignment. The assignment must be submitted to **Web-CAT**

A blockchain is a distributed data structure used to keep track of all transactions made on a digital currency (like Bitcoin) without the need for a central authority, such as a bank or a government. To achieve this, a blockchain must prevent unauthorized transfers and illegal modifications to the data. This is all made possible through a clever use of hash functions.

In this assignment, your are going to implement a simplified (and centralized) blockchain structure. This structure has three levels:

1. Transaction.
2. Block.
3. Chain.

These concepts are explained in details in what follows.

1 Transaction

A transaction is the transfer of a specified amount of money from a sender to a receiver. To authorize the transfer, the sender must sign the transaction. The creation and verification of the signature is done using two very large numbers called *private key* and *public key*, which are used as follows:

- Private key: as the name indicates, this key is kept secret by the user. The private key is used to sign any input by computing a hash value of that input. This hash value is called the *signature*.
- Public key: the user makes this key available to all other users. This key can be used to verify that the signature is valid **without** needing to know the private key.

The class `Utils` contains a set of methods to help you sign and verify signatures. The first step is to generate a pair of private/public keys. For this use the method (the keys are randomly generated, which means different calls return different keys):

```
KPair kp =Utils.getKeyPair();
```

The class `KPair` is a simple class used to store the keys:

```
public class KPair {
  public byte[] publicKey;
  public byte[] privateKey;

  public KPair(byte[] publicKey, byte[] privateKey) {
    this.publicKey = publicKey;
    this.privateKey = privateKey;
  }
}
```

Notice that since the keys are very large numbers, they cannot be represented using Java primitive data types such as int or long. This is why they are declared as arrays of bytes (**byte**[]). You can use the class *Utils* to convert them to hexadecimal strings then print them:

```
System.out.println("Private␣key:␣" + Utils.toHex(kp.privateKey));
System.out.println("Public␣key:␣" + Utils.toHex(kp.publicKey));
```

Suppose now that we want to sign the following data:

```
int a = 10;
String s = "Hello";
```

First, we transform it to bytes array by transforming each variable to **byte**[], then concatenating the result:

```
byte[] aBytes = Utils.toBytes(a);
byte[] sBytes = s.getBytes();
byte[] input = Utils.concat(aBytes, sBytes);
```

We now can sign the input data (notice that we are using the private key here):

```
byte[] signature = Utils.sign(input, kp.privateKey);
```

The signature can be checked using the public key. If we change the data, the signature becomes invalid:

```
boolean valid = Utils.isSignatueValid(input, signature, kp.publicKey);
System.out.println(valid); // Prints true
input[0]++; // Change input
valid = Utils.isSignatueValid(input, signature, kp.publicKey);
System.out.println(valid); // Prints false
```

Having understood signature creation and verification, we can now introduce the interface `Transaction`. Notice that the sender and receiver are identified by their public keys (which must be unique):

```
public interface Transaction {

  // Set the sender.
  void setSender(byte[] sender);

  // Set the receiver.
  void setReceiver(byte[] receiver);

  // Set the amount of transfered money.
  void setAmount(int amount);

  // Return the signature of the transaction.
  void setSignature(byte[] signature);

  // Return the public key of the sender.
  byte[] getSender();

  // Return the public key of the receiver.
  byte[] getReceiver();

  // Return the amount of transfered money.
  int getAmount();

  // Return the signature of the transaction.
  byte[] getSignature();

  // Sign the transaction using pvk as private key. The signature must be
       made using Utils.sign.
  void sign(byte[] pvk);

  // Return true if the transaction is signed with a valid signature by
      the sender. The signature must be verified using Utils.
      isSignatueValid.
  boolean isSignatureValid();

  // Converts transaction data to bytes in the order: (sender public key,
       receiver public key, amount, signature)
  byte[] toBytes();
}
```

Your first step is to write the class `TransactionImp` that implements the interface `Transaction` and must have a default constructor.

2 Block

A block contains transaction data. In real applications a block may contain several transactions, but in our case we assume that the block contains only one transaction. More specifically, our block contains:

- A transaction.
- The hash value of the block: This is a value computed using a specific hash function that takes as input the block represented as **`byte[]`** and computes a hash value. To reduce the number of blocks that can be added to the system, the hash value must satisfy the condition that the first k bytes must be 0. The hash functions used here have an almost zero probability of collision, which means that if the hash values are different, then the blocks must be different.
- A nonce: Making the hash value valid can only be done by modifying the block data. Since we cannot change the transaction data, we add a dummy integer called *nonce* to the block. The value of the nonce can be changed to make the hash value of the block valid.
- The miner's public key: This is the public key of the user who added the block (and who computed the nonce).
- The hash value of the previous block: By including the hash of the previous block, we ensure that any change to the data in the previous block will also make the hash value of the current block invalid.

The class `Utils` contain methods that can be used to implement a block. To compute a hash value, first transform to `byte[]` then call `Utils.getHash`:

```java
int a = 10;
String s = "Hello";
byte[] aBytes = Utils.toBytes(a);
byte[] sBytes = s.getBytes();
byte[] input = Utils.concat(aBytes, sBytes);
byte[] hash = Utils.getHash(input);
System.out.println("Hash␣value:␣" + Utils.toHex(hash));
```

To check if a block hash is valid call `Utils.validBlockHash`.

Your second step is to write the class `BlockImp` that implements the interface `Block` below and must have a default constructor.

```java

public interface Block {

  // Return the miner public key.
  byte[] getMiner();

  // Return the nonce of the block.
  int getNonce();

  // Return the transaction included in the block
  Transaction getTransaction();

  // Return the hash value of the block.
  byte[] getHash();

  // Return the hash of the previous block, 0 otherwise.
  byte[] getPrevHash();

  // Set the miner's public key.
  void setMiner(byte[] miner);

  // Set the nonce.
  void setNonce(int nonce);

  // Set the transaction
  void setTransaction(Transaction transaction);

  // Set previous hash
  void setPrevHash(byte[] prevHash);

  // Set the hash value of the block.
  void setHash(byte[] hash);

  // Check if the hash is valid.
  boolean isHashValid();

  // Compute hash of the block using Utils.getHash. The order of the data
       is as specified in toBytes.
  void compHash();

  // Find the smallest non-negative nonce that makes the hash of the
      block valid according to Utils.validHash.
  void mine();

  // Converts block data (except hash) to bytes in the order: (nonce,
      transactions (converted to bytes using Transaction.toBytes()), hash
      of previous block)
  byte[] toBytes();

}
```

3 Chain

The last level in a blockchain is the chain itself, which can be thought of as a list of blocks as shown in Figure 1. Notice that each block contains a hash value computed as a function of blocks' data and the hash value of the previous block. Hence, if a block is modified all blocks after it will have invalid hash values. This makes it very hard for someone to change the content of a block.

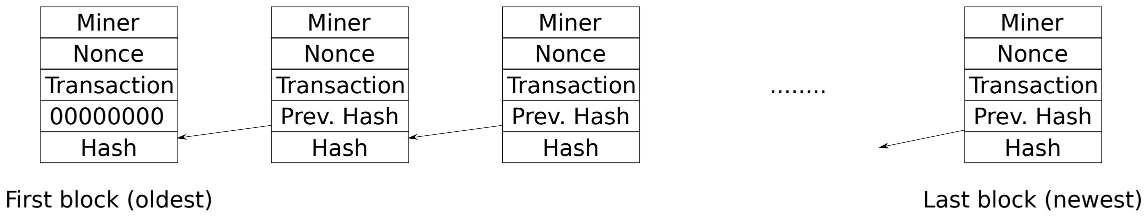

Figure 1: Example of a blockchain.

Every block except the first one must contain a transaction that shows how money is transferred between users. The first block may or may not contain a transaction. Miners obtain a specific amount of money as compensation for computing the nonce. This is the only way money is made. All users start with a 0 balance and can only transfer money if they have enough in their balance to do it.

Example 1. *Consider the chain in Figure 2:*

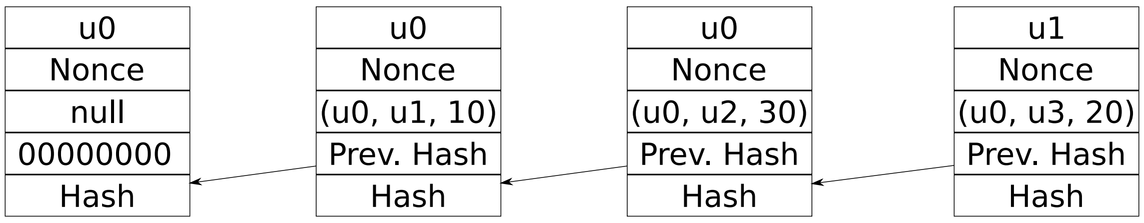

Figure 2: Example of a blockchain.

If the reward for the miner is 100 per block, then the balance of the users evolves as follows:

	After block 1	After block 2	After block 3	After block4
u0	100	190	260	240
u1	0	10	10	110
u2	0	0	30	30
u3	0	0	0	20

Example 2. *In the chain shown in Figure 3, the third block is invalid, because user u2 does not have enough money to make this transfer. Hence block 3 and all blocks after it (in this case only block 4) are considered invalid.*

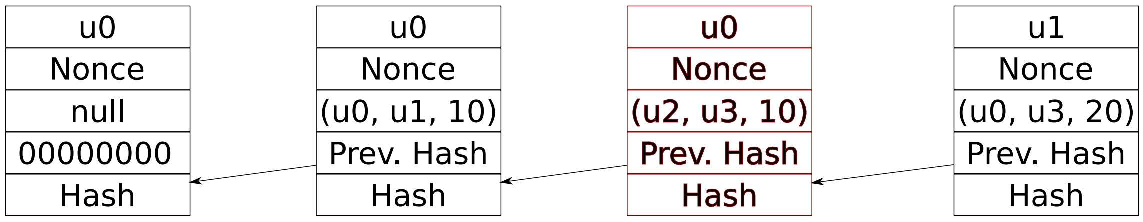

Figure 3: Example of a blockchain with an invalid block.

Your last step is to write the class `BlockchainImp` that implements the interface `Blockchain` below and must have a default constructor.

```
public interface Blockchain {

  // This is the initial hash used in the first block.
  public final static byte[] initHash = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
     0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0, 0 };

  // Reward for miner.
  public static final int minerReward = 100;

  // Return the length of the chain
  int length();

  // Get the hash of the last block
  byte[] getLastBlockHash();

  // Add a block to the chain. If the block is not the first block and
     does not contain a transaction, the block is not inserted and false
     is returned, otherwise the block is inserted at the end and true is
     returned.
  boolean addBlock(Block b);

  // Return the list of blocks in their order (first block must be the
     first in the list).
  List<Block> getBlocks();

  // Return the balance of user pbk. If pbk does not exist in the chain,
     0 is returned. If the balance of pbk becomes negative at any point,
     the method returns -1.
  int getBalance(byte[] pbk);

  // Remove all invalid blocks. A block is valid if all blocks before it
     are valid, its hash is correct, and its transactions are valid. A
     transaction is valid if it's signature is valid, and the sender has
     enough money to make the transfer.
  void removeInvalid();

}
```

4 Deliverable and rules

You must deliver:

1. Source code submission to Web-CAT. You have to upload the following classes (and any other classes you need) in a zipped file:

- `TransactionImp.java`
- `BlockImp.java`
- `BlockchainImp.java`

Notice that you should **not upload** the interfaces `Transaction`, `Block`, `Blockchain`, `List` and the classes `Utils`, `LinkedList`.

The submission **deadline** is: **11/04/2019**.
You have to read and follow the following rules:

1. The specification given in the assignment (**class and interface names, and method signatures**) must not be modified. Any change to the specification results in compilation errors and consequently the mark zero.
2. All data structures used in this assignment **must be implemented** by the student. The use of Java collections or any other data structures library is strictly forbidden.
3. This assignment is an individual assignment. Sharing code with other students will result in harsh penalties.
4. Posting the code of the assignment or a link to it on public servers, social platforms or any communication media including but not limited to Facebook, Twitter or WhatsApp will result in disciplinary measures against any involved parties.
5. The submitted software will be evaluated automatically using Web-Cat.
6. All submitted code will be automatically checked for similarity, and if plagiarism is confirmed penalties will apply.
7. You may be selected for discussing your code with an examiner at the discretion of the teaching team. If the examiner concludes plagiarism has taken place, penalties will apply.