Class BlockChain

java.lang.Object
blockchaintask0.BlockChain

```
public class BlockChain
extends java.lang.Object
```

The BlockChain Class

This class represents a simple BlockChain.

Constructor Summary

Constructors

Constructor	Description
`BlockChain()`	This BlockChain has exactly three instance members - an ArrayList to hold Blocks and a chain hash to hold a SHA256 hash of the most recently added Block.

Method Summary

All Methods Static Methods Instance Methods Concrete Methods

Modifier and Type	Method	Description
void	`addBlock(blockchaintask0.Block newBlock)`	A new Block is being added to the BlockChain.
void	`computeHashesPerSecond()`	This method computes exactly 2 million hashes and times how long that process takes.
blockchaintask0.Block	`getBlock(int i)`	return block at position i
java.lang.String	`getChainHash()`	
int	`getChainSize()`	
int	`getHashesPerSecond()`	get hashes per second

		chain.
double	**getTotalExpectedHashes**()	Compute and return the expected number of hashes required for the entire chain.
java.lang.String	**isChainValid**()	If the chain only contains one block, the genesis block at position 0, this routine computes the hash of the block and checks that the hash has the requisite number of leftmost 0's (proof of work) as specified in the difficulty field.
static void	**main**(java.lang.String[] args)	This routine acts as a test driver for your Blockchain.
void	**repairChain**()	This routine repairs the chain.
java.lang.String	**toString**()	This method uses the toString method defined on each individual block.

Methods inherited from class java.lang.Object

equals, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructor Detail

BlockChain

```
public BlockChain()
```

This BlockChain has exactly three instance members - an ArrayList to hold Blocks and a chain hash to hold a SHA256 hash of the most recently added Block. It also maintains an instance variable holding the approximate number of hashes per second on this computer. This constructor creates an empty ArrayList

Method Detail

getChainHash

```
public java.lang.String getChainHash()
```

Returns:

the chain hash.

getTime

```
public java.sql.Timestamp getTime()
```

Returns:

the current system time

getLatestBlock

```
public blockchaintask0.Block getLatestBlock()
```

Returns:

a reference to the most recently added Block.

getChainSize

```
public int getChainSize()
```

Returns:

the size of the chain in blocks.

computeHashesPerSecond

getHashesPerSecond

```
public int getHashesPerSecond()
```

get hashes per second

Returns:

the instance variable approximating the number of hashes per second.

addBlock

```
public void addBlock(blockchaintask0.Block newBlock)
```

A new Block is being added to the BlockChain. This new block's previous hash must hold the hash of the most recently added block. After this call on addBlock, the new block becomes the most recently added block on the BlockChain. The SHA256 hash of every block must exhibit proof of work, i.e., have the requisite number of leftmost 0's defined by its difficulty. Suppose our new block is x. And suppose the old blockchain was a <-- b <-- c <-- d then the chain after addBlock completes is a <-- b <-- c <-- d <-- x. Within the block x, there is a previous hash field. This previous hash field holds the hash of the block d. The block d is called the parent of x. The block x is the child of the block d. It is important to also maintain a hash of the most recently added block in a chain hash. Let's look at our two chains again. a <-- b <-- c <-- d. The chain hash will hold the hash of d. After adding x, we have a <-- b <-- c <-- d <-- x. The chain hash now holds the hash of x. The chain hash is not defined within a block but is defined within the block chain. The arrows are used to describe these hash pointers. If b contains the hash of a then we write a <-- b.

Parameters:

`newBlock` - is added to the BlockChain as the most recent block

toString

```
public java.lang.String toString()
```

This method uses the toString method defined on each individual block.

Overrides:

`toString` in class `java.lang.Object`

getBlock

```
public blockchaintask0.Block getBlock(int i)
```

return block at position i

Parameters:

`i` -

Returns:

block at postion i

getTotalDifficulty

```
public int getTotalDifficulty()
```

Compute and return the total difficulty of all blocks on the chain. Each block knows its own difficulty.

Returns:

totalDifficulty

getTotalExpectedHashes

```
public double getTotalExpectedHashes()
```

Compute and return the expected number of hashes required for the entire chain.

Returns:

totalExpectedHashes

isChainValid

```
public java.lang.String isChainValid()
```

If the chain only contains one block, the genesis block at position 0, this routine computes the hash of the block and checks that the hash has the requisite number of leftmost 0's (proof of work) as specified in the difficulty field. It also checks that the chain hash is equal to this computed hash. If either check fails, return an error message. Otherwise, return the string "TRUE". If the chain has more blocks than one,

TRUE if the chain is valid, otherwise return a string with an appropriate error message

repairChain

```
public void repairChain()
```

This routine repairs the chain. It checks the hashes of each block and ensures that any illegal hashes are recomputed. After this routine is run, the chain will be valid. The routine does not modify any difficulty values. It computes new proof of work based on the difficulty specified in the Block.

main

```
public static void main(java.lang.String[] args)
```

This routine acts as a test driver for your Blockchain. It will begin by creating a BlockChain object and then adding the Genesis block to the chain. The Genesis block will be created with an empty string as the pervious hash and a difficulty of 2.

On start up, this routine will also establish the hashes per second instance member. All blocks added to the Blockchain will have a difficulty passed in to the program by the user at run time. All hashes will have the proper number of zero hex digits representing the most significant nibbles in the hash. A nibble is 4 bits. If the difficulty is specified as three, then all hashes will begin with 3 or more zero hex digits (or 3 nibbles, or 12 zero bits).

It is menu driven and will continously provide the user with seven options:

Block Chain Menu

0. View basic blockchain status.

1. Add a transaction to the blockchain.

2. Verify the blockchain.

3. View the blockchain.

4. Corrupt the chain.

5. Hide the corruption by repairing the chain.

6. Exit.

will then prompt for and then read a line of data from the user (representing a transaction). The program will then add a block containing that transaction to the block chain. The program will display the time it took to add this block. Note: The first block added after Genesis has index 1. The second has 2 and so on. The Genesis block is at position 0.

If the user selects option 2, then call the isChainValid method and display the results. It is important to note that this method will execute fast. Blockchains are easy to validate but time consuming to modify. Your program needs to display the number of milliseconds it took for validate to run. If the user selects option 3, display the entire Blockchain contents as a correctly formed JSON document. See www.json.org. If the user selects option 4, she wants to corrupt the chain. Ask her for the block index (0..size-1) and ask her for the new data that will be placed in the block. Her new data will be placed in the block. At this point, option 2 (verify chain) should show false. In other words, she will be making a data change to a particular block and the chain itself will become invalid.

If the user selects 5, she wants to repair the chain. That is, she wants to recompute the proof of work for each node that has become invalid - due perhaps, to an earlier selection of option 4. The program begins at the Genesis block and checks each block in turn. If any block is found to be invalid, it executes repair logic.

Important:

Within your comments in the main routine, you must describe how this system behaves as the difficulty increases. Run some experiments by adding new blocks with increasing difficulties. Describe what you find. Be specific and quote some times.

You need not employ a system clock. You should be able to make clear statements describing the approximate run times associated with addBlock(), isChainValid(), and chainRepair().

Parameters:

`args` - is unused