Blockchain programming part 4: Java programming language

The purpose of this tutorial is to help you build a panoramic view of how people can develop blockchain technology.

1. Programming blockchain part 1: C ++ programming language
2. Programming blockchain part 2: Javascript programming language
3. Programming blockchain part 3: Python programming language

The previous 3 parts you learned about programming blockchain with C ++, JavaScript, Python. The purpose of this tutorial is to help you build a panoramic view of how people can develop blockchain technology in the Java language.

Java programming language

In this tutorial, the main task will be:

1. Create basic (very) basic 'blockchain'.
2. Make a simple proof of work (system exploitation).
3. Learn the capabilities of blockchain.

(Provided you have a basic understanding of object-oriented programming).

Keep in mind that the blockchain created in this article merely helps clarify concepts and helps you read more about the blockchain.

Setting

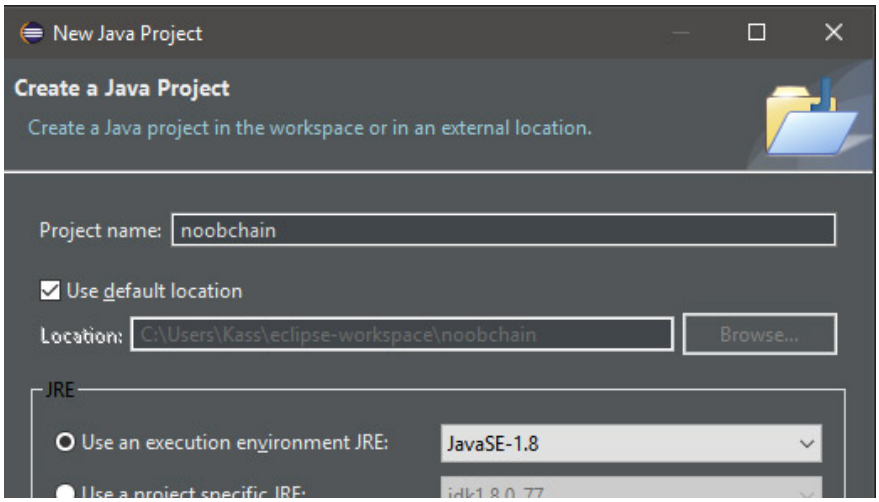
The article will use Java but you will be able to do the same in any OOP language. The article will use Eclipse, but you can also use any new favorite text editor.

You will need:

1. Java and JDK. (Duh) have been installed
2. Eclipse (or an IDE / other text editor).

However, you can use the GSON library of google. This will allow you to turn an object into Json o /. It is an extremely useful library that will continue to be used as other similar tools. If you do not want to, please use an alternative method.

In Eclipse, create a Java project (**file> new>**). Suppose the project name is ' **noobchain** ' and create a new Class of the same name (**NoobChain**).

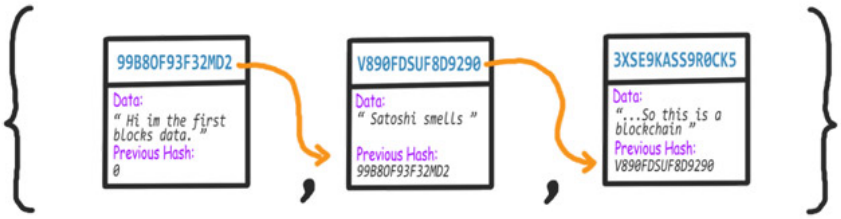


Now you are ready to create blockchain already!

1. Download and install Java on the computer
2. How to fix the error does not install Java

Create Blockchain

Blockchain is just a string or list of blocks. Each block in the blockchain will have its own Digital Signature, containing the previous block's digital signature and some data (for example, this data may be transactions).



Hash = Digital Signature

Each block contains not only the hash function of the previous block, but also its own hash (calculated from the previous hash value). If the previous block's data is changed, the previous hash value of the block will change (because it is computed in part by the data) and will affect all the hash of the blocks later. Compute and compare hashes to tell whether a blockchain is valid or not.

That means, changing any data in this list, will change the digital signature and break the string.

Therefore, Firsts allows creating the Block class that makes up the blockchain:

```
import java.util.Date;
public class Block {
```

```
public String hash;
public String previousHash;
private String data;  // our data will be a simple message.
private long timeStamp;  // as number of milliseconds since 1/1/1970.
// Block Constructor.
public Block (String data, String previousHash) {
this.data = data;
this.previousHash = previousHash;
this.timeStamp = new Date (). getTime ();
}
}
```

As you can see, the Basic Block contains a **String hash** (keeping the digital signature). The **previousHash** variable contains the hash data and the string of the previous block to hold the block data.

Next we will need to find a way to create a digital signature.

There are many encryption algorithms that you can choose, however SHA256 is best suited for this example. We can enter

```
java.security.MessageDigest
```

to access SHA256 algorithm.

It is necessary to use SHA256 and then down the line to allow creation of a handy help method in a new 'utility' StringUtil class:

```
import java.security.MessageDigest;
public class StringUtil {
// Applies Sha256 to a string and returns the result.
public static String applySha256 (String input) {
try {
MessageDigest digest = MessageDigest.getInstance ("SHA-256");
// Applies sha256 to our input,
byte [] hash = digest.digest (input.getBytes ("UTF-8"));
StringBuffer hexString = new StringBuffer ();  // This will contain hash as hex
for (int i = 0; i
String hex = Integer.toHexString (0xff & hash [i]);
if (hex.length () == 1) hexString.append ('0');
hexString.append (hex);
}
return hexString.toString ();
}
catch (Exception e) {
throw new RuntimeException (e);
}
}
}
```

Don't worry too much if you don't understand the content of this help. All you need to know is that it takes a string and applies the **SHA256** algorithm, then returns the signature created as a string.

Now use **applySha256 helper** , in a new method in the Block class, to calculate the hash value. Need to calculate the hash from all parts of the block. So this block will include **previousHash, data** and **timeStamp:**

```
public String calculateHash () {
String calculatedhash = StringUtil.applySha256 (
previousHash +
Long.toString (timeStamp) +
data
);
return calculhash;
}
```

as well as allowing to add this method to the **Block constructor:**

```
public Block (String data, String previousHash) {
this.data = data;
this.previousHash = previousHash;
this.timeStamp = new Date (). getTime ();
this.hash = calculateHash ();  // Making sure we do this after we set the othe
}
```

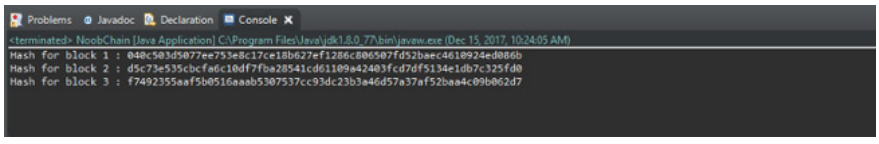
Some tests

The main NoobChain class allows creating some blocks and exporting the hash to the screen to see that everything is working properly.

The first block is called the genesis block, and because there is no previous block, we will only enter '0' as previous hash.

```
public class NoobChain {
public static void main (String [] args) {
Block genesisBlock = new Block ("Hi im the first block", "0");
System.out.println ("Hash for block 1:" + genesisBlock.hash);
Block secondBlock = new Block ("Yo im the second block", genesisBlock.hash);
System.out.println ("Hash for block 2:" + secondBlock.hash);
Block third Block = new Block ("Hey im the third block", secondBlock.hash);
System.out.println ("Hash for block 3:" + thirdBlock.hash);
}
}
```

The result will look like this:



Each block now has its own digital signature based on the information and signature of the previous block.

Currently, blockchain doesn't have many blocks, so store blocks in **ArrayList** and also enter **gson** to treat it as **Json** .

```
import java.util.ArrayList;
import com.google.gson.GsonBuilder;
public class NoobChain {
public static ArrayList blockchain = new ArrayList ();
public static void main (String [] args) {
// add our blocks to the ArrayList blockchain:
blockchain.add (new Block ("Hi im the first block", "0"));
blockchain.add (new Block ("Yo im the second block", blockchain.get (blockchai
blockchain.add (new Block ("Hey im the third block", blockchain.get (blockchai
String blockchainJson = new GsonBuilder (). SetPrettyPrinting (). Create (). To
System.out.println (blockchainJson);
}
}
```

Now, the display results will be the same as what we expect in a blockchain.

Next, find a way to test the integrity of the blockchain.

Create Boolean **isChainValid () method** in NoobChain class. It will loop through all the blocks in the string and compare the hash. This method will need to check the hash variable actually with the calculated hash and the previous hash of the hash with the previousHash variable.

```
public static Boolean isChainValid () {
CurrentBlock block;
Block previousBlock;
// loop through blockchain to check hashes:
for (int i = 1; i
currentBlock = blockchain.get (i);
previousBlock = blockchain.get (i-1);
// compare registered hash and calculate hash:
if (! currentBlock.hash.equals (currentBlock.calculateHash ())) {
System.out.println ("Current Hashes not equal");
return false;
}
// compare hash before and registered previous hash
if (! previousBlock.hash.equals (currentBlock.previousHash)) {
System.out.println ("Previous Hashes not equal");
return false;
}
}
return true;
}
```

Any changes to blockchain blocks will cause this method to return **false** .

Bitcoin network nodes share their blockchain and valid longest chain will be accepted by the network. So what prevents someone from forging data in an old block then creating a completely new blockchain?**Proof of work** (proof of activity). Proof of work **Hashcash** proof of work system means that it takes a lot of time and effort to calculate new blocks. Therefore the attacker will spend a lot of computational effort when combining the blocks.

Exploiting blocks

You will have to ask the miner to perform **proof-of-work** by trying different variable values ??in the block until the hash value begins with a specific number, starting at **0** .

Add an **int,** called **nonce,** to the **calculateHash ()** method and **mineBlock () method** :

```
import java.util.Date;
public class Block {
public String hash;
public String previousHash;
private String data;  // our data will be a simple message.
private long timeStamp;  // as number of milliseconds since 1/1/1970.
private int nonce;
// Block Constructor.
public Block (String data, String previousHash) {
this.data = data;
this.previousHash = previousHash;
this.timeStamp = new Date (). getTime ();
this.hash = calculateHash ();  // Making sure we do this after we set the othe
}
// Calculate new hash based on blocks contents
public String calculateHash () {
String calculatedhash = StringUtil.applySha256 (
previousHash +
Long.toString (timeStamp) +
Integer.toString (nonce) +
data
);
return calculhash;
}
public void mineBlock (int difficulty) {
String target = new String (new char [difficulty]). Replace ('', '0');  // Crea
while (! hash.substring (0, difficulty) .equals (target)) {
nonce ++;
hash = calculateHash ();
}
System.out.println ("Block Mined !!!:" + hash);
}
}
```

In fact each miner will start repeating from a random point, some miners can try random numbers for nonce. The harder solutions will require more integer.MAX_VALUE, the miner can now try to change the timestamp.

MineBlock () method contains int named named **difficulty** , this is the number of numbers starting from 0 that they must solve. Low difficulty levels like 1 or 2 can be solved almost immediately on most computers, so raise it to about 4-6 for testing. At the time of writing, Litecoin's difficulty level is about 442,592.

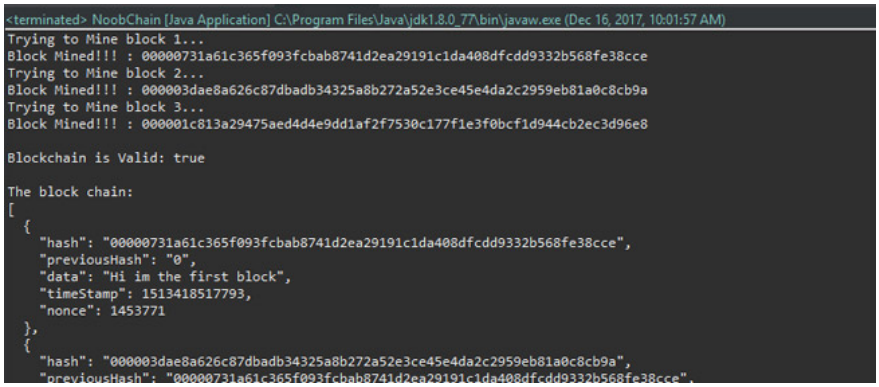
Now we add the difficulty of making static variables into the NoobChain class:

```
public static int difficulty = 5;
```

It is recommended to update the NoobChain class to activate **mineBlock () method** for each new block. Boolean **isChainValid ()** also checks whether each block has a hash that has been resolved (by digging).

```
import java.util.ArrayList;
import com.google.gson.GsonBuilder;
public class NoobChain {
public static ArrayList blockchain = new ArrayList ();
public static int difficulty = 5;
public static void main (String [] args) {
// add our blocks to the ArrayList blockchain:
blockchain.add (new Block ("Hi im the first block", "0"));
System.out.println ("Trying to Mine block 1 .");
blockchain.get (0) .mineBlock (difficulty);
blockchain.add (new Block ("Yo im the second block", blockchain.get (blockchai
System.out.println ("Trying to Mine block 2 .");
blockchain.get (1) .mineBlock (difficulty);
blockchain.add (new Block ("Hey im the third block", blockchain.get (blockchai
System.out.println ("Trying to Mine block 3 .");
blockchain.get (2) .mineBlock (difficulty);
System.out.println ("nBlockchain is Valid:" + isChainValid ());
String blockchainJson = new GsonBuilder (). SetPrettyPrinting (). Create (). To
System.out.println ("nThe block chain:");
System.out.println (blockchainJson);
}
public static Boolean isChainValid () {
CurrentBlock block;
Block previousBlock;
String hashTarget = new String (new char [difficulty]). Replace ('', '0');
// loop through blockchain to check hashes:
for (int i = 1; i
currentBlock = blockchain.get (i);
previousBlock = blockchain.get (i-1);
// compare registered hash and calculate hash:
if (! currentBlock.hash.equals (currentBlock.calculateHash ())) {
System.out.println ("Current Hashes not equal");
return false;
}
// compare hash before and registered previous hash
if (! previousBlock.hash.equals (currentBlock.previousHash)) {
System.out.println ("Previous Hashes not equal");
return false;
}
// check if hash is solved
if (! currentBlock.hash.substring (0, difficulty) .equals (hashTarget)) {
System.out.println ("This block has not been mined");
return false;
}
}
return true;
}
}
```

The results displayed will be as follows:



Digging each block will take a bit of time! (about 3 seconds). You should try changing different difficulty values ??to see how it affects the time needed to dig each block.

If someone spoofs data in the blockchain system:

1. Blockchain will not be valid
2. Will not be able to create a longer blockchain.
3. The honest blockchain in the network will have the advantage of time in the longest chain.

A fake blockchain will not be able to catch longer and valid strings unless they have a greater computing speed than all the other nodes in the network combined (maybe with a quantum computer in the future or whatever. there).

The basic blockchain is complete! Keep in mind the following about your Blockchain:

1. Made up of data storage blocks.
2. There is a digital signature that connects your blocks together.
3. Request proof of work to confirm new blocks.
4. It is possible to check if the data in it is valid and has been changed.

See more:

1. Programming blockchain part 5: Solidity programming language