TCS — Blockchain Tutorial 2

# Explore Blockchain with a Simple JavaScript Example

To reinforce your understanding of the role of blockchain we illustrate the main concepts by building a simple blockchain using JavaScript. This tutorial is adapted from a YouTube video that very nicely illustrates how the blockchain works.

1. [Create a Blockchain in JavaScript](https://www.youtube.com/watch?v=zVqczFZr124) — Blockchain in JavaScript (Part 1).
   * This part of the tutorial implements a very basic blockchain structure in JavaScript, which in no way could be regarded as usable, it merely illustrates concepts.
2. [Implementing Proof of Work in JavaScript](https://www.youtube.com/watch?v=HneatE69814) — Blockchain in JavaScript (Part 2).
   * This part of the tutorial adds a proof of work method to increase the difficulty of calculating the validity of the block, again merely to illustrates concepts

To complete the last three parts of this tutorial you will need to watch and follow the video tutorials:

1. [Miners Rewards and Transactions](https://www.youtube.com/watch?v=fRV6cGXVQ4I) — Blockchain in JavaScript (Part 3).
2. [Signing Transactions](https://www.youtube.com/watch?v=kWQ84S13-hw) — Blockchain in JavaScript (Part 4).
3. [Angular Frontend](https://www.youtube.com/watch?v=AQV0WNpE_3g) — Blockchain in JavaScript (Part 5).

You may simply click on the links above to follow the tutorials in YouTube or follow the brief written tutorials Part 1 and Part 2 below.

**Side note:**

You should note that at the time of writing there were many other elements of a blockchain’s block structure missing in Part 1 and Part 2 of tutorial, parts 3 to 5 had not been completed in the videos. However, completing Part 1 and Part 2 is useful to get a good idea of how a blockchain works. There are other resources that allow you t examine a blockchain and see how it works. For example, [BLOCKCYPHER](https://www.blockcypher.com/dev/bitcoin/#introduction) allows you to explore a number of blockchains including: Bitcoin, Ethereum, Litecoin, Dogecoin, and Bitcoin Testnet3

BlockCypher is a simple, mostly RESTful JSON API for interacting with blockchains, accessed over HTTP or HTTPS from the [api.blockcypher.com](https://api.blockcypher.com/v1/btc/main) domain. An API here, mifght return in JSON format the state of a particular block that includes attributes such as: Hash, height, depth, chain, total, … a full list of bitcoin block attributes and their description can be found on the BLOCKCYPHER [bitcoin block description](https://www.blockcypher.com/dev/bitcoin/#block).

As an example, if you want to examine any particular bitcoin transaction based on its blockheight, assuming you know the block height you can run this API endpoint:

<https://api.blockcypher.com/v1/btc/main/blocks/496500?txstart=1&limit=1>

This happens to be, at the time of writing, the latest blockheight. Figure 1 shows the returned JSON formatted block.

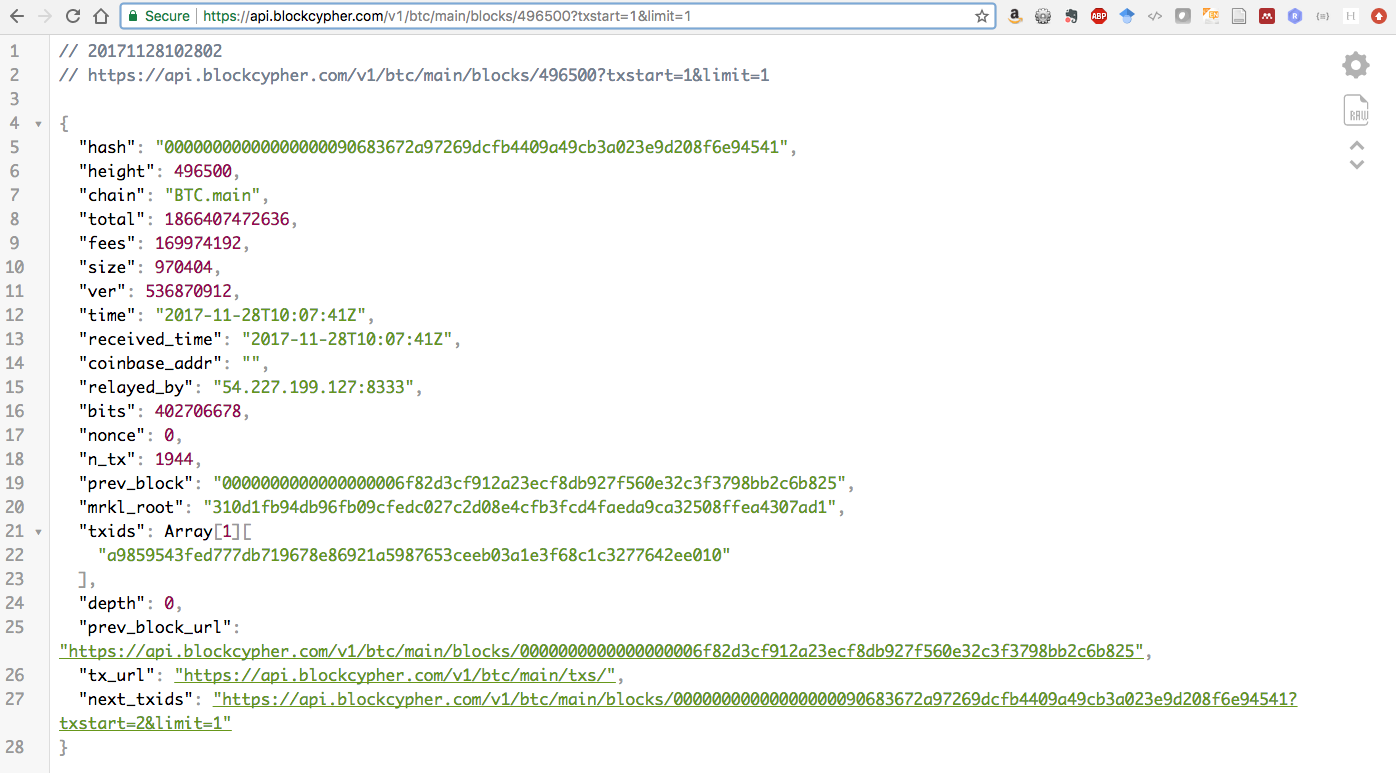


Figure 1: Bitcoin latest block at time of writing, see timestamp (time).

Further, the technology behind blockchain uses advanced cryptography protocols, peer to peer networks, and many other technological solutions. So, it is highly unlikely that you would develop a complete new blockchain technology each time we need a blockchain for a new project. Instead you would most likely build upon existing open source blockchain technologies such as [Ethereum](https://www.ethereum.org/) or [Hyperledger](https://hyperledger.org/). Or, in terms of the research on visualising blockchain currency transactions outlined in the presentation we would use published API endpoints to develop our application.

If you want some quick background reading on blockchain, here is a three-part series from marmalab on their blog page that looks interesting:

The Blockchain Explained to Web Developers:

1. [Part 1: The Theory](https://marmelab.com/blog/2016/04/28/blockchain-for-web-developers-the-theory.html)
2. [Part 2: In Practice](https://marmelab.com/blog/2016/05/20/blockchain-for-web-developers-in-practice.html)
3. [Part 3: The Truth](https://marmelab.com/blog/2016/06/14/blockchain-for-web-developers-the-truth.html)

Note, these articles were posted in 2016, things change, new opinions are formed, technology moves on!

Ok, let’s get started with Part 1.

## Part 1 — Create a Blockchain in JavaScript

Our initial presentation gave you a feel for what a blockchain is, how it is structured, how it works. Part 1 of this Blockchain Tutorial 2 is based on the [YouTube video](https://www.youtube.com/watch?v=zVqczFZr124) indicated above, and has been selected because it very neatly illustrates some important blockchain concepts, while also allowing you to reinforce these concepts through code experimentation.

However, we should be aware that several underlying concepts are missing. So, this blockchain would not work very well in practice.

In this Blockchain Tutorial 2, Part 1, we will call our blockchain 'myBitcoin' and we will use OOP (Object Oriented Programming) for JavaScript. If you completed our Blockchain Tutorial 1, you will recall we used OOP for PHP to create our Model.

### Create a block structure

So, step by step:

1. First, we should create a JavaScript file in our MVC frameworks scripts folder. Call this JavaScript file jsBlockchain.js. Add, a comment of some sort to the top to describe what the JavaScript file is going to do, good practice.
2. Next, let’s create a block class. We start by creating a constructor class called 'block' to represent a block in the blockchain. This block class will hold methods to represent the main components of the block, such as: the block hash, previous hash, data, timestamp, and so on, 
3. Figure 2. Check out what attributes a typical bitcoin block structure contains on the [BlockCypher block page](https://www.blockcypher.com/dev/bitcoin/#block), to realise this is indeed a very simple structure.



Figure 2: Create the Class Block and its constructor.

* Code explanation
  + Line 2-8: This is the Block class with its constructor
  + Line 5-7: This is the constructor of the class Block that holds the block properties: index, timestamp, data, and previoushash. From line 6 we will add some methods.
    - 'index' is where the block sits in the blockchain.
    - 'timestamp' tells us when the block was created.
    - 'data' holds any type of data you want to hold in the block, e.g. bitcoins transacted, address of sender, address of receiver, etc.
    - 'previoushash' is the hash value of the previous block in the blockchain, which ensures the integrity of the blockchain, and is initially set to empty, as we have no previoushash in the genesis block.

1. We now need to keep track of the block properties: index, timestamp, data and previoushash using the JavaScript keyword ‘this’ to own the Block class constructor object properties ([check out ‘this’ in w3schools](https://www.w3schools.com/js/js_this.asp)), Figure 3.

* Code explanation:
  + 7 – 10: The value of ‘this’ in JavaScript, will become the new object when the constructor is used to create an object. Check this out: [Gentle explanation of ‘this’ keyword in JavaScript](https://dmitripavlutin.com/gentle-explanation-of-this-in-javascript/). Also, read this [[quir]{mode]](https://www.quirksmode.org/js/this.html) for a good explanation.
    - The upshot of this is that when you invoke the construction (with its class Block properties) this.index = index; this.timestamp = timestamp, etc. these variables belong to that block.
  + Line 12: We also have to calculate the current block hash, so we will need a global variable for this, which at the moment is set to empty

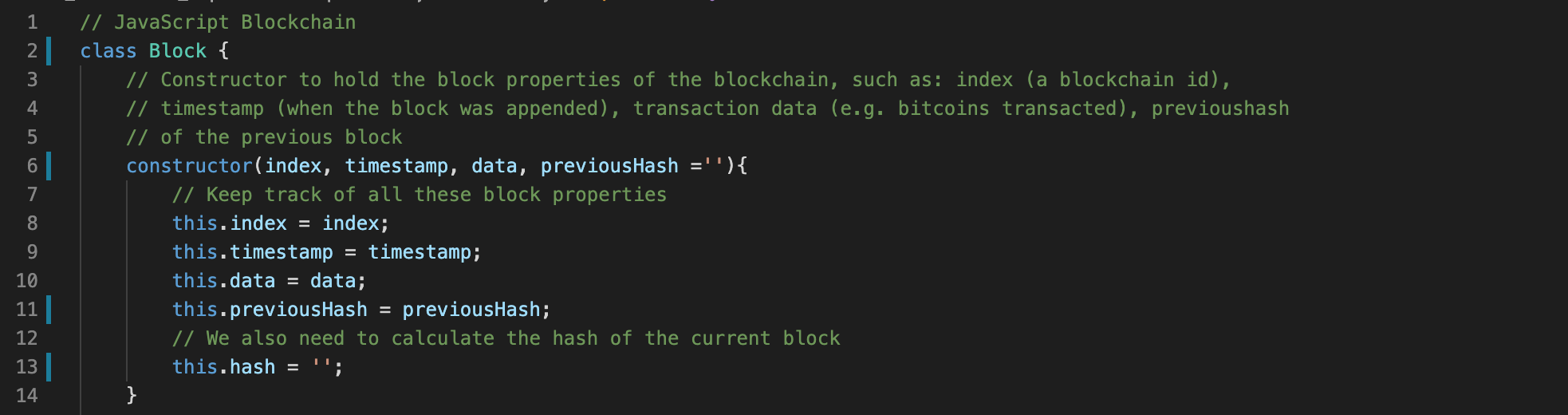


Figure 3: Keep track of the block properties

1. We need to create a method to calculate the current block hash using a JavaScript hash library such as SHA256. This method in the constructor will take the current block properties index, timestamp, data, previoushash and run them through a hash function to create the current block hash used to identfy this block on the blockchain.   
     
   JavaScript does not have a native hash library so we need to import a hash library, e.g. crypto.js. I am using Visual Studio Code for my developer environment, because it has lots of nice features such as: an integrated terminal window, integrated GitHub, and so on. So, you will need to access a terminal window to install crypto.js. Before you install crypto.js you can have a play around with a SHA-256 Cryptographic Hash Algorithm on the [Moveable Type Scripts](https://www.movable-type.co.uk/scripts/sha256.html) site.  
     
   When you have finished exploring SHA-256 on this site, open up a terminal window to check if you already have node.js installed, which is an open source JavaScript runtime environment built on Chrome’s JavaScript engine, and if so you may also want to update it. If it is not installed on your laptop or home computer, then there are many online resources that show you how to install node.js. In my case, I just installed Brew first, then used Brew to install node.js — there are other ways to install [node.js](https://nodejs.org/en/), e.g. You will need admin rights.  
     
   Anyhow, assuming that you have now installed node, then [npm](https://www.npmjs.com/), the node package manager for JavaScript can be used to install crypto.js.  
     
   You will need to type ‘npm install --save crypto-js’ on the terminal command line. You will also need to decide where to install the SHA256 library and as these are JavaScript libraries, perhaps you should change directory in your terminal window to your scripts folder and install there  
     
   Go ahead and type ‘npm install --save crypto-js’ on the terminal command line. This will install the node\_modules folder containing the crypto-js folder contacting in turn a load of JavaScript files including the sha256.js file needed to calculate the current block hash. Figure 4 shows the installation on the terminal window in Visual Studio Code. You can see here that I just installed node and checked the version 9noe -v), and then I used npm to install crypto-js, and when running npm the first time I got a dialog box telling me a new version of npm was available, so I updated that.

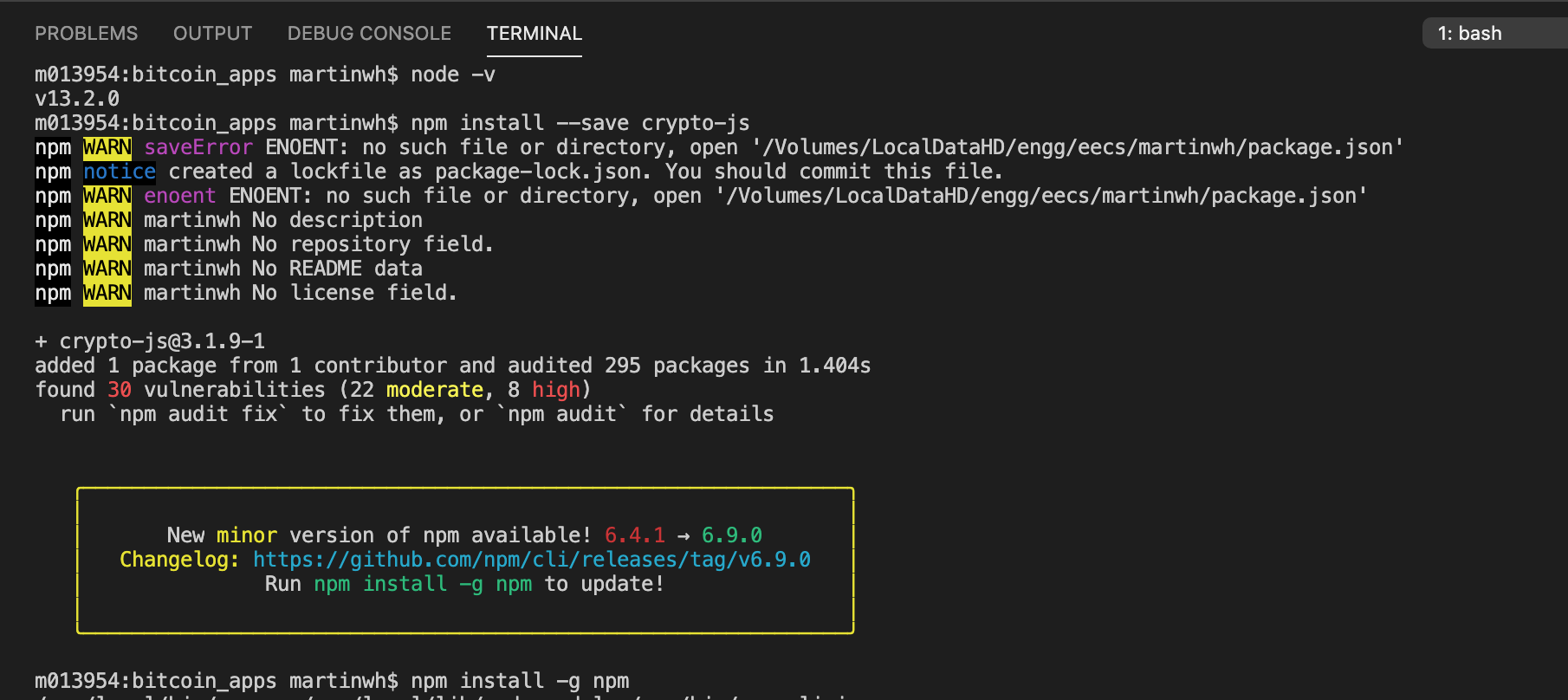


Figure 4: Install the node\_modules/crypto-js/sha256.js files used to calculate the current block hash value

1. Now we can start to create the calculate hash method and use the SHA256 library to do this, Figure 5.

* Code explanation
  + Line 2: This now includes (requires) the SHA256 JavaScript hash library, which is used to calculate the hash value of the current block.
  + Line 19 – 23: This is the calculateHash() method in the Block class, which uses the SHA256 JavaScript hash library to calculate the hash. We input to the SHA256 function the current index, previoushash, timestamp and data. In doing so we convert the JavaScript data into JSON string format, and also cast the hash result into a string format, otherwise SHA256 returns an object.
  + Line 15: We now modify this.hash to calculate the current block hash value.

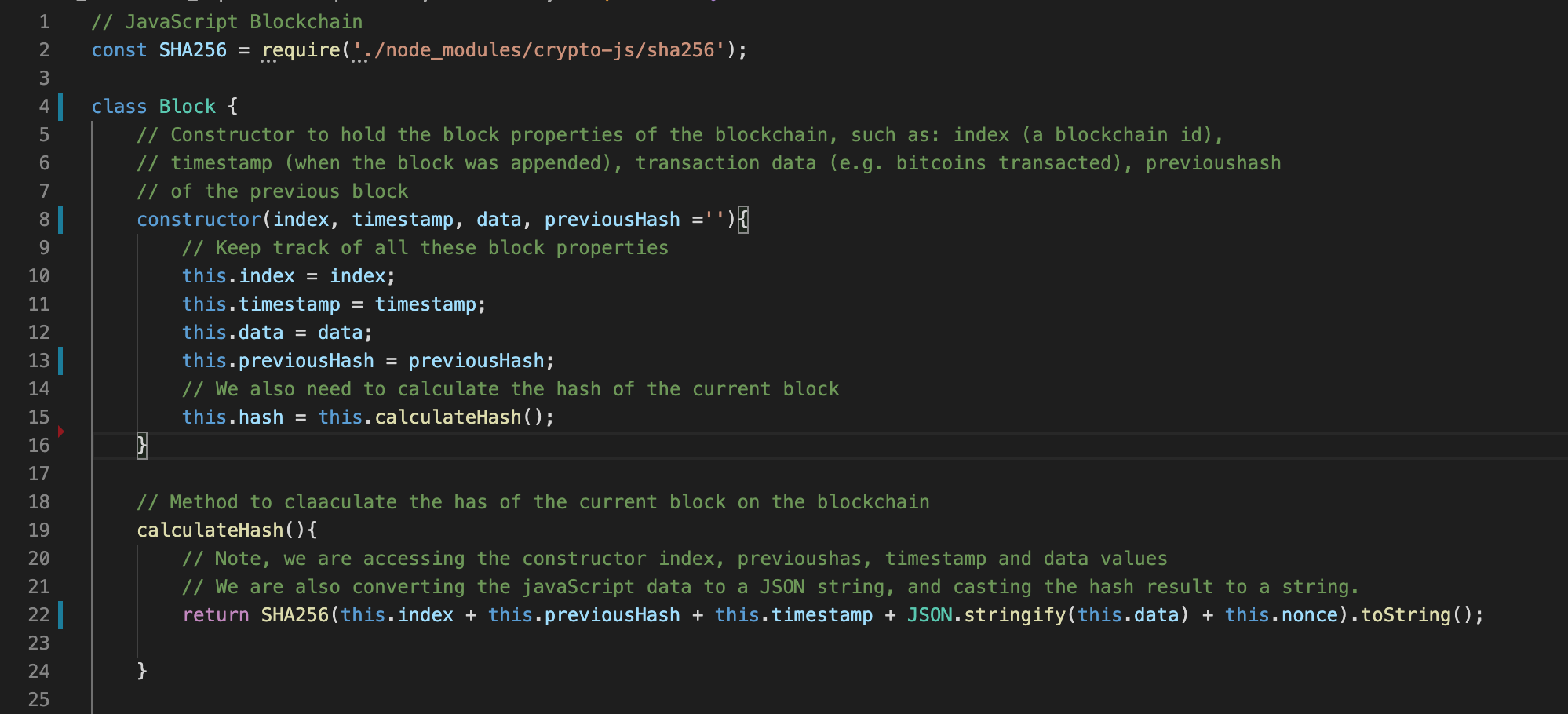


Figure 5: Add the calculatehash() method

Next, we should create a blockchain class with a constructor that initialises the blockchain.

### Create a blockchain structure

1. Create a new class for our blockchain, Figure 6.

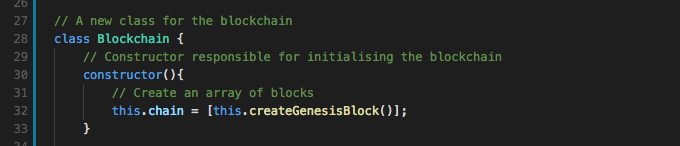


Figure 6: Create the new blockchain class to chain the blocks together

* Code explanation
  + Line 28 – 33: Clearly, we are creating a new blockchain class to chain the blocks together.
  + Line 30 – 32: Is a constructor method responsible for initialising the blockchain with an array of blocks.
  + Line 32: Creates a chain with an array of blocks.

1. Next, we need to initialise the blockchain with a genesis block, the first block in the blockchain, Figure 7.

* Code explanation
  + Line 36 -39: Create a method to create the genesis block, where
  + Line 38 returns the new (Genesis) block with appropriate values.

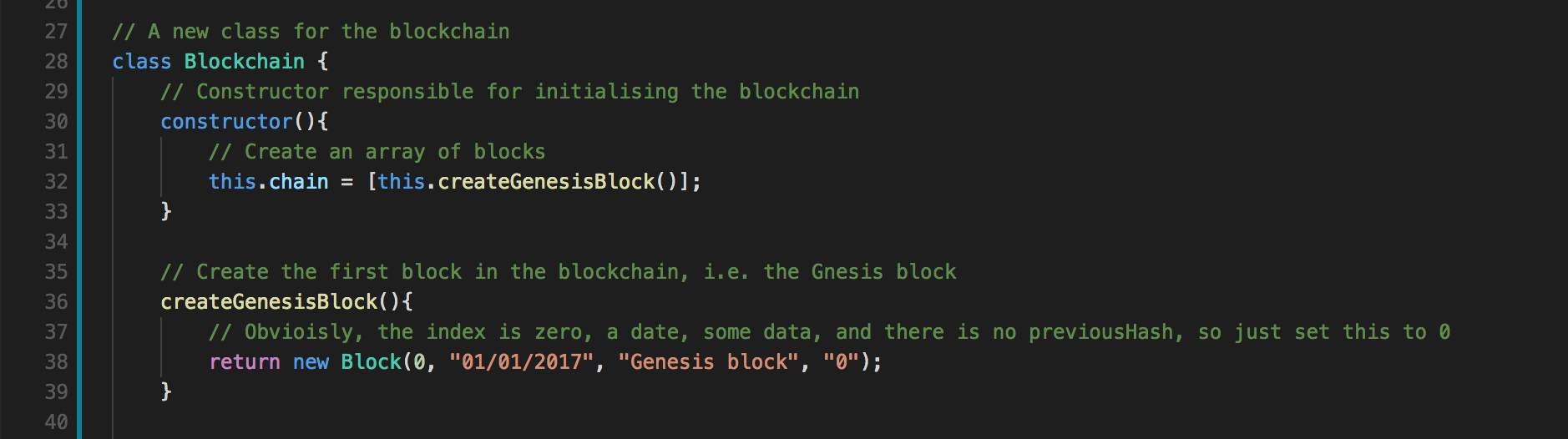


Figure 7: Create the genesis block

1. Next, we will need a couple of other methods, for example: the ability to get the latest block and add a new block to the blockchain, Figure 8

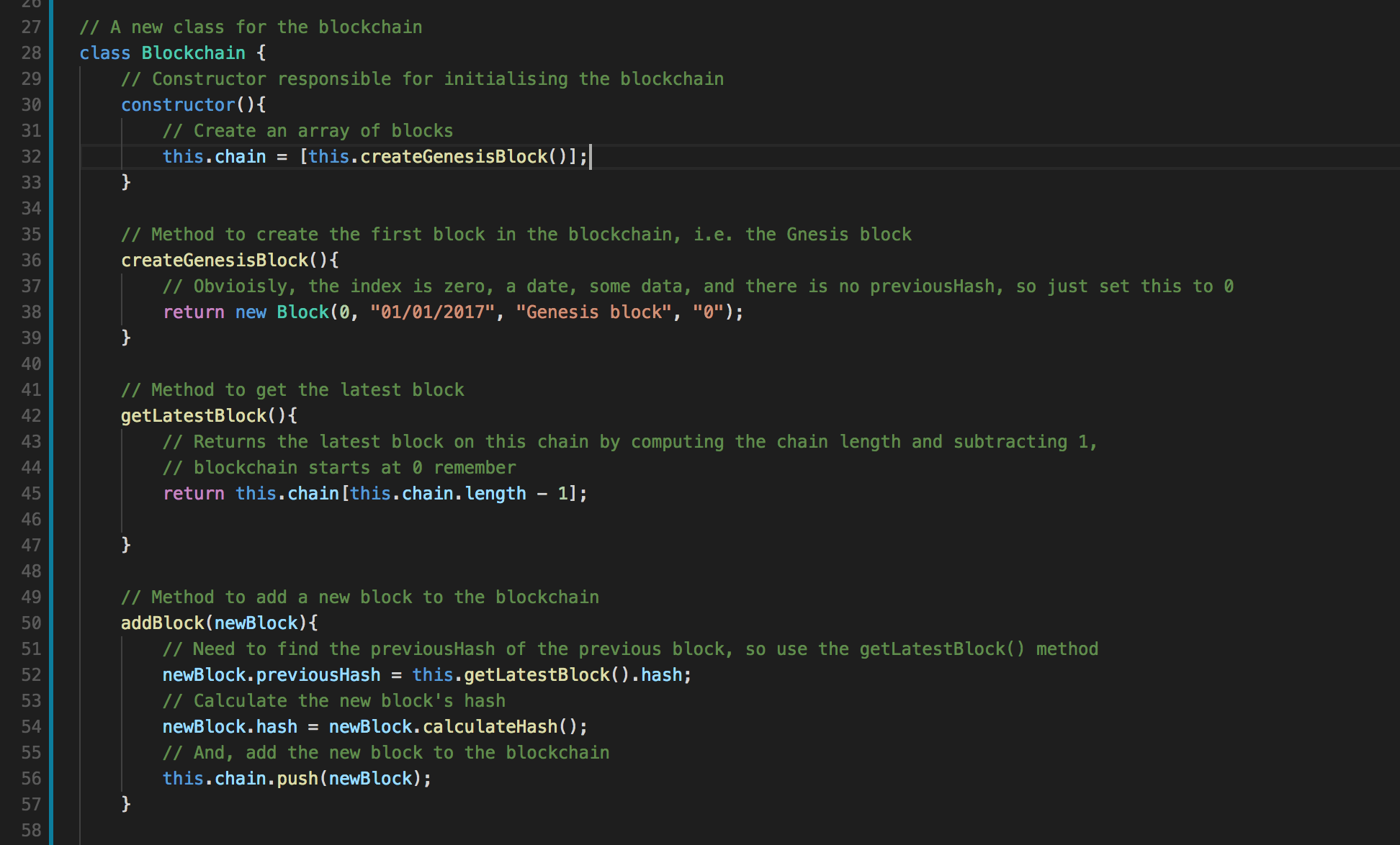


Figure 8: Create the getLatestBlock() and addBlock() methods.

* Line 42 – 47: The methods to get the latest block, where
* Line 45 returns the latest block index by finding the chain length and subtracting 1.
* Line 50 – 58: Create the addBlock method, where
* Line 52: Gets the latest block has to create the previousHash values to append to this new block
* Line 54: Creates the new block hash using the SHA256 based calculateHash() method, and
* Line 56 adds the new block to the blockchain.

1. Next, we need to test if all this works so far, so we will use a terminal window to see our results. But, first we need to create the new blockchain, add a couple of new blocks to the blockchain and use conole.log to view the results in the terminal window, Figure 9.

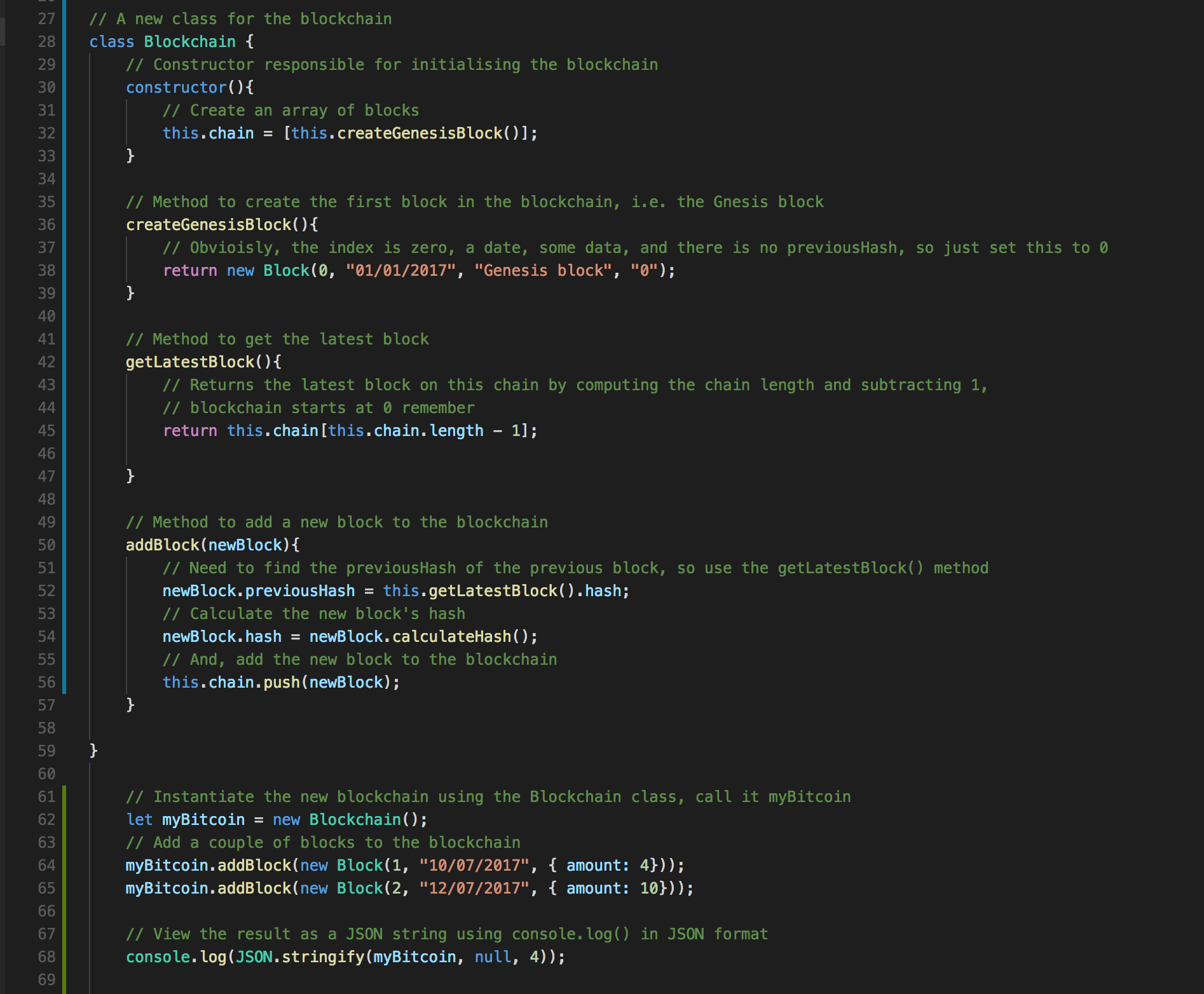


Figure 9: Test the blockchain

* Code explanation
  + Line 61: Instantiates the new blockchain (myBitcoin)
  + Lines 63 -64: Adds two new blocks to the blockchain
  + Line 67: Uses console.log to view the blockchain

1. You will need to open up a terminal window to test your blockchain by running your jsblockchain.js file, don’t forget to save it, debug it, etc. You should have node.js installed. Run node jsblockchain.js on the command line, Figure 10. We can see that we have output the blockchain, which has the property (or name) chain, which is also an array of blocks each containing an index, timestamp, data, previousHash, and hash (of the current block). If your’s does not work, then debug it before you move on. However, you can see that each block points back to the previous block via the previousHash

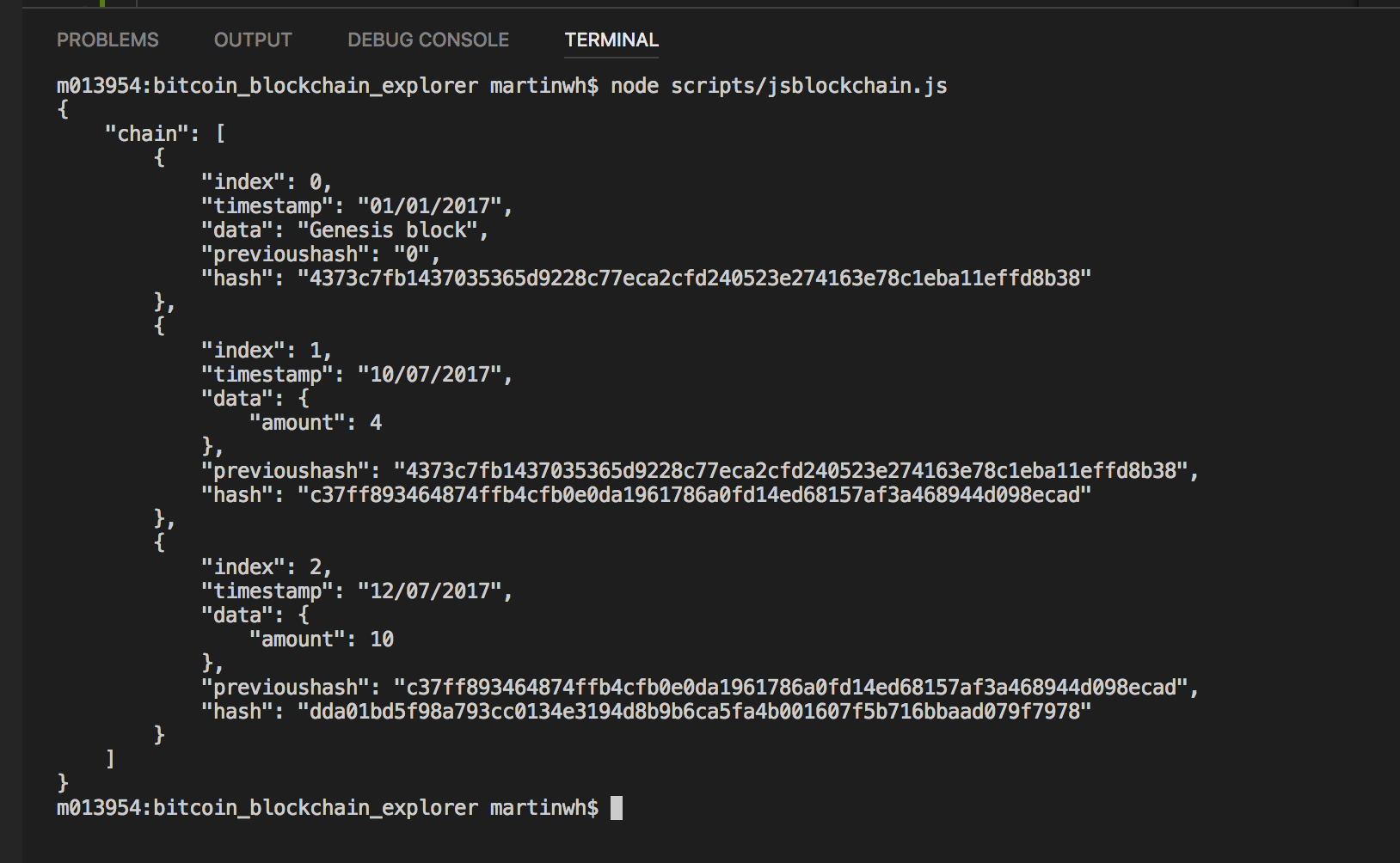


Figure 10: Running your myBitcoin blockchain

Already, at this stage, if you have completed the Blockchain Tutorial 1, you should be able to reflect on this piece of JavaScript code and consider how to re-implement it in PHP. For example, how can you invoke it from a browser application using JQuery? How can you integrate (or rebuild) it into your MVC framework, e.g. using the SQLite database and PHP to implement classes and associated methods for the blocks and the blockchain. You would also need the PHP controller methods to invoke the PHP methods associated with PHP block and blockchain classes.

Moving on, the whole tenet about a blockchain is that it should be immutable, i.e. any block data cannot be changed without invalidating the rest of the chain. Clearly though, at this stage, we could simply just change the data in a block and recalculate the rest of the chain. This is in effect exposes two issues:

1. We need to implement a method to verify the integrity of our blockchain, so we will do that now;
2. And, later in Part 2 of this Blockchain Tutorial 2, we will implement a ‘proof of work’ that makes it computationally too expensive change a block’s data value and then re-calculate all the following blockchain hash values — make it so that you need computational power greater than 51% of all the miners in the network.
   * This turns out to be critical in order to prevent what is often called the ‘double spending problem’ in digital currencies, i.e. that of preventing Alice from sending Bob 2 bitcoins, and then trying to send her sister the same 2 bitcoins — which at today’s Bitstamp Exchange Rate of $7,351 would be nice little earner if you could get away with it!

### Create a blockchain validation method

1. So, let’s create a new method in our blockchain class to verify if the blockchain is valid, i.e. isChainValid(), Figure 11.

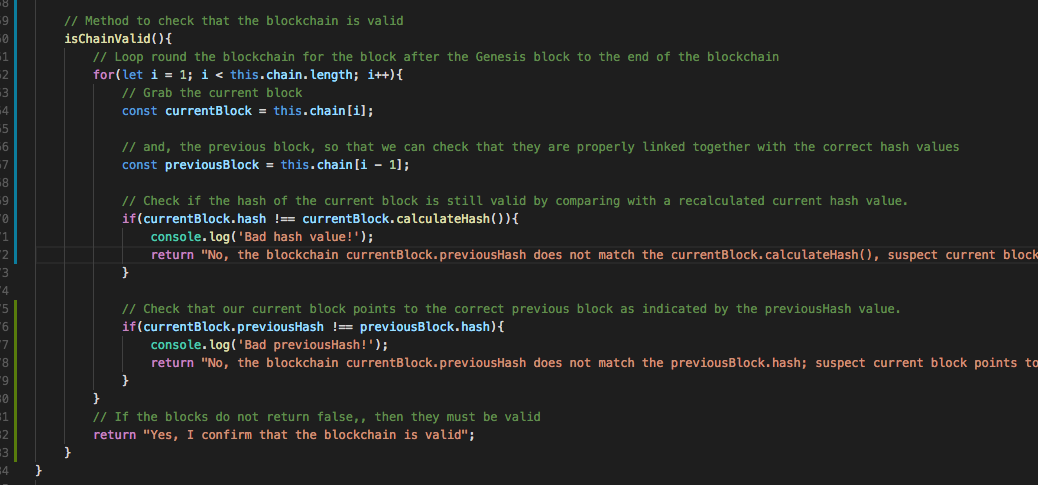


Figure 11: JavaScript methods to check the blockchain is valid

* Code explanation
  + Lines 60 -83: Implement the isChainValid() method, where first
  + Lines 62 – 80: We set up a for loop to loop around the blockchain from block 1 (we don’t need block 0 because this has no previousHash, it being the Genesis block) to the end indicated by ‘this.chain.length - 1’. And, of course, we increment the for-loop counter to traverse the chain. At each loop, we check that the current block points back to the previous block by comparing the currentBlock.previoushash value in the current block matchs the previousBlock.hash value in the previous block. If true, clearly the blocks are linked together correctly, but if false we return a message indicating so. You could just return false. I have clipped off the message in Figure 11, so don’t forget to enclose it in double quotes and end with a semi-colon. Similarly, we check that the currentBlock.hash value matches a recalculated hash value for the current block. If different, it indicates that the current block has been tampered with.
  + Line 64 - 67: we set up constants to store the current block and the previous block. We will access the current block hash and previous hash values to compare them to ensure we are linking to the current blocks in the chain and check that any data has not been tampered with.
  + Lines 70 – 73: As mentioned above, check if the hash of the current block is still valid by asking it to recalculate the current hash value. If any data value has been changed in any of the blocks the hash value (calculated with calculateHash()) will differ for the block's stored has value indicating that the data has been tampered with. This effectively makes the blockchain invalid. Of course, if the data has been changed, why not simply recalculate the hash too and store the new hash value. Of course, if the blockchain is very long, and data in an early block was changed, we might have to recalculate thousand upon thousands of blocks in the chain. But hey, we have fast computers right? This is considered in Part 2 of this Blockchain Tutorial 2. We simply make computing a hash value computationally too expensive to perform, as you'll see later.
  + Lines 76 – 79: As explained above, check that our current block points to the correct previous block as indicated by the previousHash value. So, we need to check if the previous Hash property is the correct hash set in the current block. If this is not the case, we know something is wrong because our current block does not point to the previous block; it points to some other block, perhaps a falsely created blockchain created by a dastardly cryptocurrency thief!
  + Line 82: If both checks return true, we can return true, or a simple message indicating so — the blockchain is valid

1. You will have noticed some console.log() statements. I put those in because, everyone implements bugs, so these were just to trace when we find ourselves having to debug!

### Test the integrity of our blockchain

1. Let’s test the code so far, comment out the old console.log() on line 96, and add the new console.log() statement on line 93, Figure 12.



Figure 12: Test the blockchain validation

* Code explanation
  + Lines 87 – 90: As before, this code instantiate a new blockchain and adds two blocks, so we should have three blocks as before, i.e. the genesis block and the two transaction blocks
  + Line 93: A new console.log() function concatenates a message ‘Is blockchain valid?’ with the result to be returns from the myBitcoin (the new blockchain) isChainValid() method.

1. Let’s verify the integrity of our blockchain. On the command line, you can now test, as you did before in Figure 10, by entering node jsblockchain.js, but make sure you are inside your scripts folder (or provide the path to your scripts folder), Figure 13.



Figure 13: Test the blockchain is valid

1. Ok, well, assuming no bugs, it is bound to show that the blockchain is valid because we haven’t tampered with it. Let’s do a bit of tampering, Figure 14.

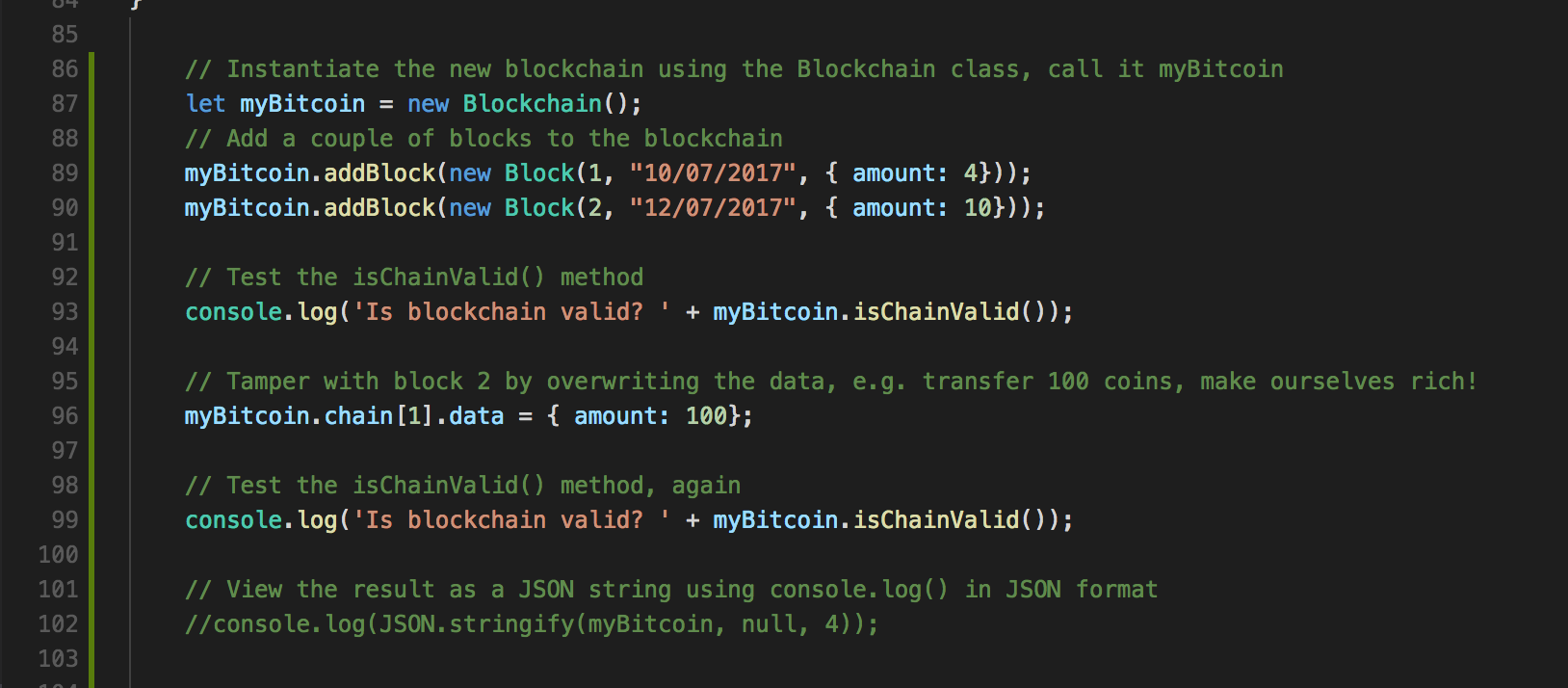


Figure 14: Tamper with the blockchain by over writing the data in block 2

* Code explanation
  + Lines 87 – 93: As before
  + Line 96: Here we are overwriting block 2 index by 1, block 1 being the zero Genesis block, by assigning (in JSON format) the amount: 100 coins.
  + Line 99: Copy the previous console.log() to validate the blockchain. So, we should see that when the blockchain is created, and the two blocks are added, it will validate ok, but when we tamper with the block it should show invalidated. Run the jsblockchain.js file again to test it, Figure 15.

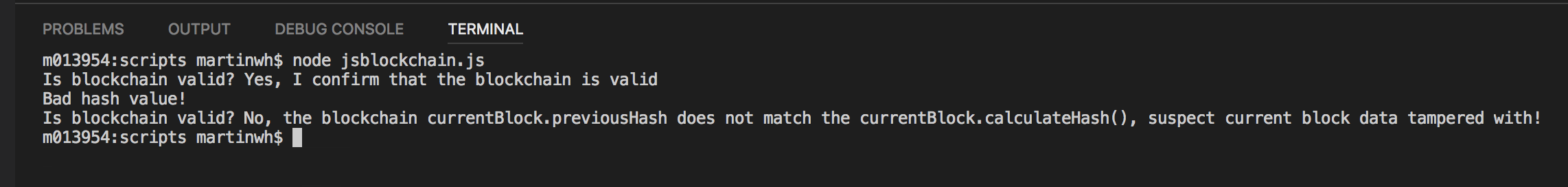


Figure 15: Test for tampering with a block’s data

1. Ok, so far, we only tampered with the data, we tried to get rich quick! It didn’t work, we got found out. But, what if we recalculate the hash value for block 2 and replace the hash value for block 2 with the new hash value. That will work, won’t it? We will then have the data, i.e. 4 coins replaced with 100 coins, and the new hash value will be correct. Let’s give it a go, Figure 16.



Figure 16: Let’s recalculate the hash to try and defeat the block validation

* Code explanation
  + Line 99: We are now overwriting block 2 hash value by recalculating its hash. Did this work? Figure 17 indicates that it did not. We can see that we hit our test to validate that a currentBlock.previousHash value did not match the previousBlock.hash value. Here, we see that in block 2 (index 1) the hash value starting with b5dc5… (i.e. the new recalculated hash) does not match the following block’s previousHash ea161 … Ooops! And, of course we output a message to say so.

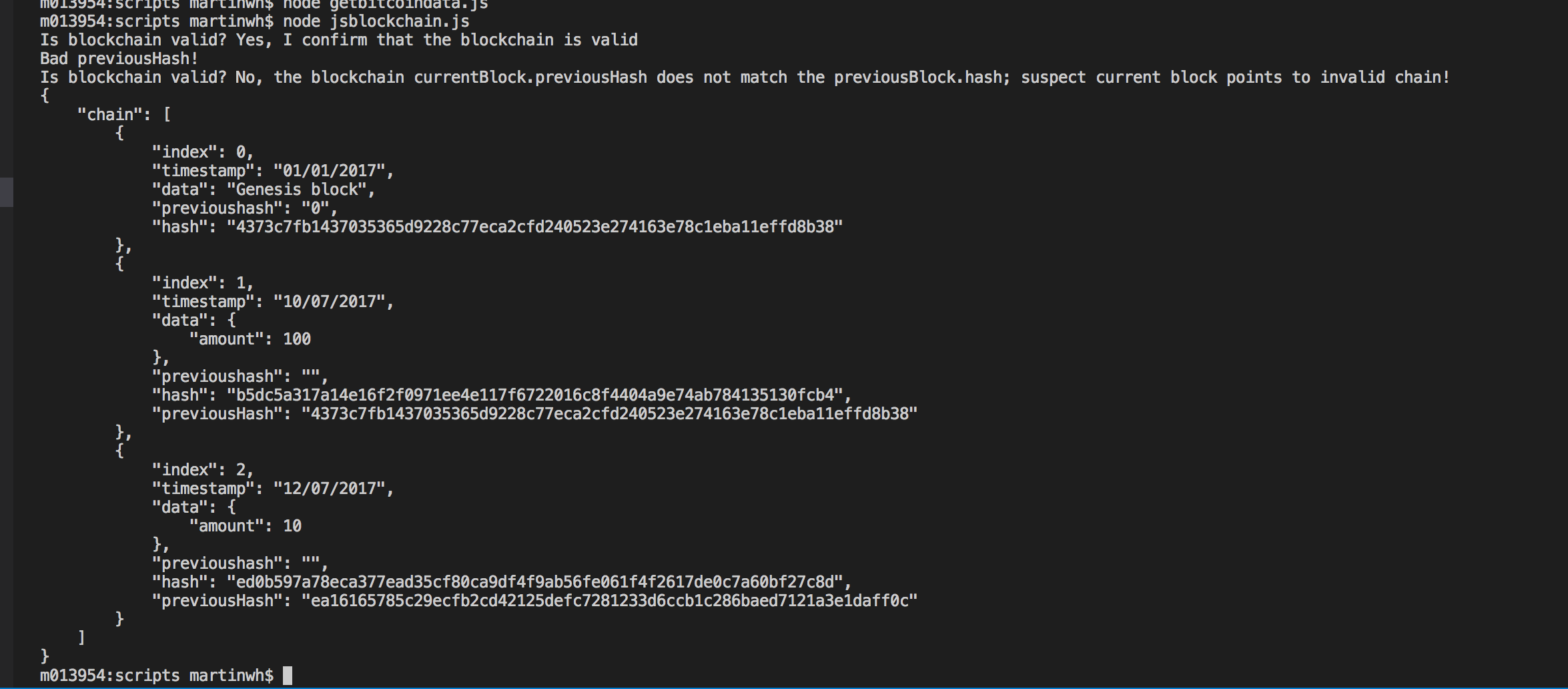


Figure 17: Test for tampering with a block’s hash value

* + We could, of course put, in more code to pin point the block that has been tampered with, etc. But, suffice to say, the chain is broken somehow. We could write code to roll back the blockchain to a previous valid version, etc. but this is beyond the scope of this tutorial.
  + This simple illustration of blockchain concepts also lacks many other features, such as a ‘proof of work’, which we will look at next in Part 2 of this Blockchain Tutorial 2, a distributed, e.g. peer 2 peer, network, lots of other ‘miners’ validating the block, it also doesn’t check to see if you have enough funds (a bitcoin wallet does this), and many other limitations. However, it does demonstrate how a blockchain works.

Next, in Part 2 of this Blockchain Tutorial 2, we will look at the concept of ‘proof of work’ as a ‘consensus’ method to prevent an attacker simply recalculating hash values for the whole blockchain, thus preventing, in effect, the creation of a new valid blockchain.

## Part 2 — Implementing a Proof of Work in JavaScript

As we are going to move forward investigating the Bitcoin blockchain, particularly when we consider exploiting APIs and WebSockets to do so, it might be instructive to consider some basic Bitcoin terminology. You have seen, in Part 1 of this Blockchain Tutorial 2, that hash functions are critical to validating a blockchain’s integrity, i.e. that the blockchain itself has not been tampered with. Hash functions are also used to **link blocks together** in the blockchain, they are also **used for mining**, **generating addresses** (e.g. for Bob and Alice to send and receive bitcoins to each other), **public and private key generation**, and **individual transactions**. Before we continue with implementing a Proof of Work, let’s review some of the key concepts behind blockchains.

* **Blockchain** — simply put, a bitcoin blockchain is a distributed shared public ledger that tracks all bitcoin transactions, and not much else — it has no smart contract technology. In slightly more detail, a blockchain is a distributed ledger technology that is an append only record system shared across a peer to peer network which uses cryptographic means to secure authenticated and verifiable transactions, and the blockchain uses a consensus (validation) protocol between all network nodes (miners) to verify transactions. Clearly, the structure of the Bitcoin blockchain is rather more sophisticated than the simple example coded above. Every user of the Bitcoin software will have a copy of the Bitcoin blockchain, so as we indicated above when we wrote our rather simple methods to verify the integrity of our blockchain, if an attacker tried to tamper with their copy of the blockchain it would generate a subsequent chain that would stick out like a sore thumb against the many thousands of valid blockchain copies, hence one would not use it. Here are a couple of very nice YouTube videos that describe the Blockchain very concisely: [How does a blockchain work — simply explained](https://www.youtube.com/watch?v=SSo_EIwHSd4), and also how they can be used: [Blockchains: how can they be used](https://www.youtube.com/watch?v=aQWflNQuP_o) applications such as governance (digital voting), verifying mileage of a car, tracking food products in supply chains, and so on.
* **Mining** — Mining is simply the process of solving a cryptographic hash function (or puzzle). Miners compete with other miners to solve a complicated mathematical problem, i.e. a cryptographic hash function. For bitcoin mining, about every 10 minutes, a miner tries to match a pre-formatted cryptographic hash value to solve a block of transactions. If a miner solves the block first then that miner is rewarded, currently with 12.5 BTC (Bitstamp Last Price was: $7204.51 on the 04/12/2019 at 09:49:56) — a potential profitable endeavour. However, it is not easy to solve the cryptographic hash function, which is basically a one-way encryption without a key. The hash function takes an input and returns a seemly random fixed length hash value — you saw this in the Part 1 Blockchain Tutorial 2 above where we used the SHA256 cryptography library to create the JavaScript calculateHash() method. We saw that any change to the input will create a completely different has value, and it is this randomness that makes it impossible to predict the output of the hash function. We can, therefore, use this characteristic as a ‘proof of work’ for validating the block. In reality, the hash value that a miner has to match will have lots of leading zeros, the length of which is an indication of a measureable difficulty. In Part 2 of this tutorial we will add a mining function that sets this difficultly level (the number of leading zeros of the hash function). That is, we can more or less determine how long it might take to find a match, but there is no way to find a match any better other than by guessing. This is why we have the evolution of hardware used to solve the cryptographic hash value for the block. In the beginning miners used a simple PC, then they discovered that the Graphics GPU could be programmed to do it faster, subsequently this led to the development of ASIC miners specifically developed to solve the making of guesses at the hash value billions of times a second. Such arrays of hardware are not cheap to run and consume lots of electricity. That is why 80% of mining operations, for example, are located in China where electricity is cheap[[1]](#footnote-1). So, in summary mining is the process of solving a cryptographic hash function to validate block transactions, and is subsequently rewarded with bitcoins. This video will help to set mining in the context of the Byzantine Generals Problem, and explain the concept of mining in the context of Bitcoin and cryptocurrency mining: [Bitcoin and cryptocurrency mining explained](https://www.youtube.com/watch?v=kZXXDp0_R-w).
* **Block** — Blocks in the Bitcoin blockchain are basically batches of transactions that are queued in a memory pool waiting to be validated within a specific timeframe (every 10 minutes). We saw in our simple example above where we manually did two transactions (input to the data property), in reality hundreds, maybe more (have a look at a typical Bitcoin block) would be hashed together with the index, previoushash, timestamp, etc. and linked to the hash from the previous block, thus creating a chain, hence the term blockchain. As already discussed, and demonstrated in the Part 1 Blockchain Tutorial 2, if Alice tried to ‘double spend’ she would have to modify a transaction in that block, recalculate the hash, and recalculate hashes for all subsequent blocks, i.e. mine the altered block and all subsequent blocks. But, and this is critical, Alice would have to do this within one time frame (roughly 10 minutes in the case of the Bitcoin blockchain) before everyone else on the network (all other miners) could mine one block. This becomes computationally infeasible when the rest of the chain is significantly longer. Bitsonblocks infographic very neatly summarizes the Bitcoin block structure, see Figure 18. Incidentally, the [Bits on Blocks — Thoughts on blockchain technology](https://bitsonblocks.net/) looks like a reasonable resource.

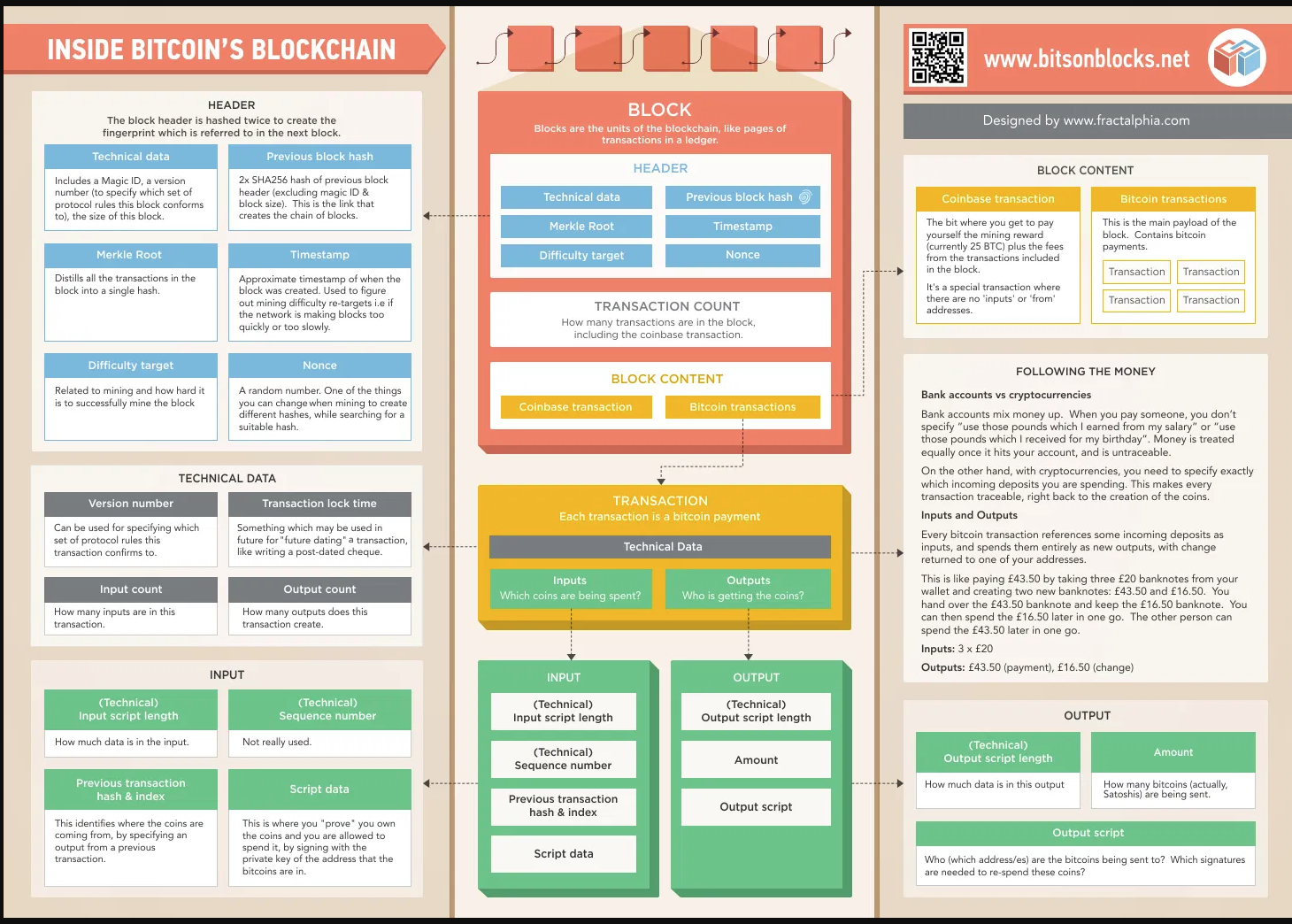


Figure 18: Inside Bitcoin's Blockchain[[2]](#footnote-2)

* **Proof of Work** — The concept of ‘proof of work’ is already hinted at above, it simply refers to the hashing rate or computer power required to mine (match the cryptographic hash value) one block. As discussed, we can only find the matching hash value by trial and error, i.e. guessing, then the act of finding a valid block, i.e. doing all the computational work required to find the hash value is proof that the work has been done — remember the Byzantine Generals Problem. You can think of it as a mechanism to slow down the validation of the block transactions such that if Alice tried to [double spend](https://www.investopedia.com/terms/d/doublespending.asp) her bitcoins, she would have to solve potentially thousands upon thousands of hash values inside the time it takes to solve one block (or the General would have to re-write all the previous Generals’ messages). Alice is unlikely to have the computational power to do that. However, there is the notion of a 51% attack, I’ll leave you to read on[[3]](#footnote-3). Part 2 of this tutorial below shows you how to code a Proof of Work using the SHA256 algorithm (library), but it is also interesting to consider other consensus mechanisms such as Proof of Stake: [Proof of Stake versus Proof of Work](https://www.youtube.com/watch?v=M3EFi_POhps) and an interesting video on 3rd generation Blockchains: [Cardano Simply Explained](https://www.youtube.com/watch?v=Do8rHvr65ZA&vl=en), which also use Proof of Stake blockchain protocols.
* **Address** — These are basically public hexadecimal strings used for receiving bitcoins. They are basically derived from a private key through the use of a hash function. The private key is used to control access to the bitcoin address, and anyone who has that private key can spend any unused outputs at that address. Similarly, if you lose your private key, then I am afraid, it is ‘tears before bedtime’, you just lost your bitcoins[[4]](#footnote-4). Many people of lost their bitcoins, check out this Telegraph article: [The £625m lost forever — the phenomenon of disappearing Bitcoins](http://www.telegraph.co.uk/technology/news/11362827/The-625m-lost-forever-the-phenomenon-of-disappearing-Bitcoins.html).
* **Private key** — A private key is basically a password, a 256-bit number. You can convert a 256-bit number composed of 0’s and 1’s into a Bitcoin private key, which through the use of hash functions, again, can be converted into an address. There are open source algorithms that allow you to create a private key, and a Bitcoin wallet will generate a unique address for each transaction for you[[5]](#footnote-5).

### Implement a Proof of Work for our simple blockchain

So far in this tutorial we can see that we can create new blocks really quickly, all we need to do a transaction is to calculate its hash and add it to an array. Modern computers can do this incredibly quickly, but we don’t want people to create hundreds of thousands of blocks per second and spam our blockchain. Blocks have to be added to the blockchain in a controlled (trusted) and incentivised way!

Also, as we have seen, we can easily tamper with a block’s data, e.g. Alice tries to overwrite a previous block entry and recalculates the hash of this block, she can then recalculate all the hash functions for the blockchain, ending up with a new valid blockchain — even though it has been tampered with!

Clearly, we need to prevent Alice from doing this, because if it were allowed to happen confidence in the blockchain and its data, e.g. bitcoin transactions would evaporate, no one would use it. We therefore have a security problem, we must prevent Alice from [double spending](https://www.investopedia.com/terms/d/doublespending.asp)!

“Double-spending is the risk that a digital currency can be spent twice. It is a potential problem unique to [digital currencies](https://www.investopedia.com/terms/d/digital-currency.asp) because digital information can be reproduced relatively easily by savvy individuals who understand the blockchain network and the computing power necessary to manipulate it.”[[6]](#footnote-6)

The solution to this lies in the implementation of a **Proof of work** consensus protocol, as briefly described above, for our blockchain. Again, a Proof of Work requires that we have put in some effort (computationally) into making a new block — this process is also called mining.

Bitcoin mining requires the hash value of a block to begin with a certain number of zeros — the more zeros, the more difficult it is (computationally) to ‘guess’ the hash – remember the hash value is a one way function, it cannot be recalculated, for example like solving for x in 5x = 15. So, because you cannot influence the output of a hash function you simply have to make many guesses to try and find a hash value that matches the required hash function with the leading zeros — you have to hope that your guess matches the hash function before competing miners’ guesses. To be clear, imagine the equation 5x = 15 wasn’t solvable (clearly it is), but if it wasn’t it would take 4 guesses to find x, 0. 1, 2 and 3 (the solution). Now imagine this equation is the hash value (it happens to be an hexadecimal value — 00000d430e16dd68b1de5fa59b4a9d519a30ebb61e4a2ae3bd6db07e7df8607b — difficulty level set to 5) usually with many leading zeros controlled by the difficulty level. It can take millions of guessed to solve this, i.e. guess a matching value with the same number of zeros.

Remember, we said above, that Bitcoin can also vary the difficulty of matching the hash by increasing the required number of leading zeros in the hash value. Bitcoin uses this difficulty mechanism to keep the verification and addition of one new block to the blockchain about every 10 minutes.

In practice, Bitcoin increases the difficulty as more miners mine and their computing power gets better, such that the miners are making correct guesses faster, thus bitcoin increases the difficulty to produce a steady stream of blocks added to the blockchain, one every 10 minutes. Now, let’s implement some code to mine a block!

1. We will add a new method to the Block class called mineBlock() that takes as input a difficulty attribute, which will simply be a number to determine the number of leading zeros required in a matching hash value — remember, the more leading zeros required the longer it takes to randomly find a matching hash value, Figure 19.

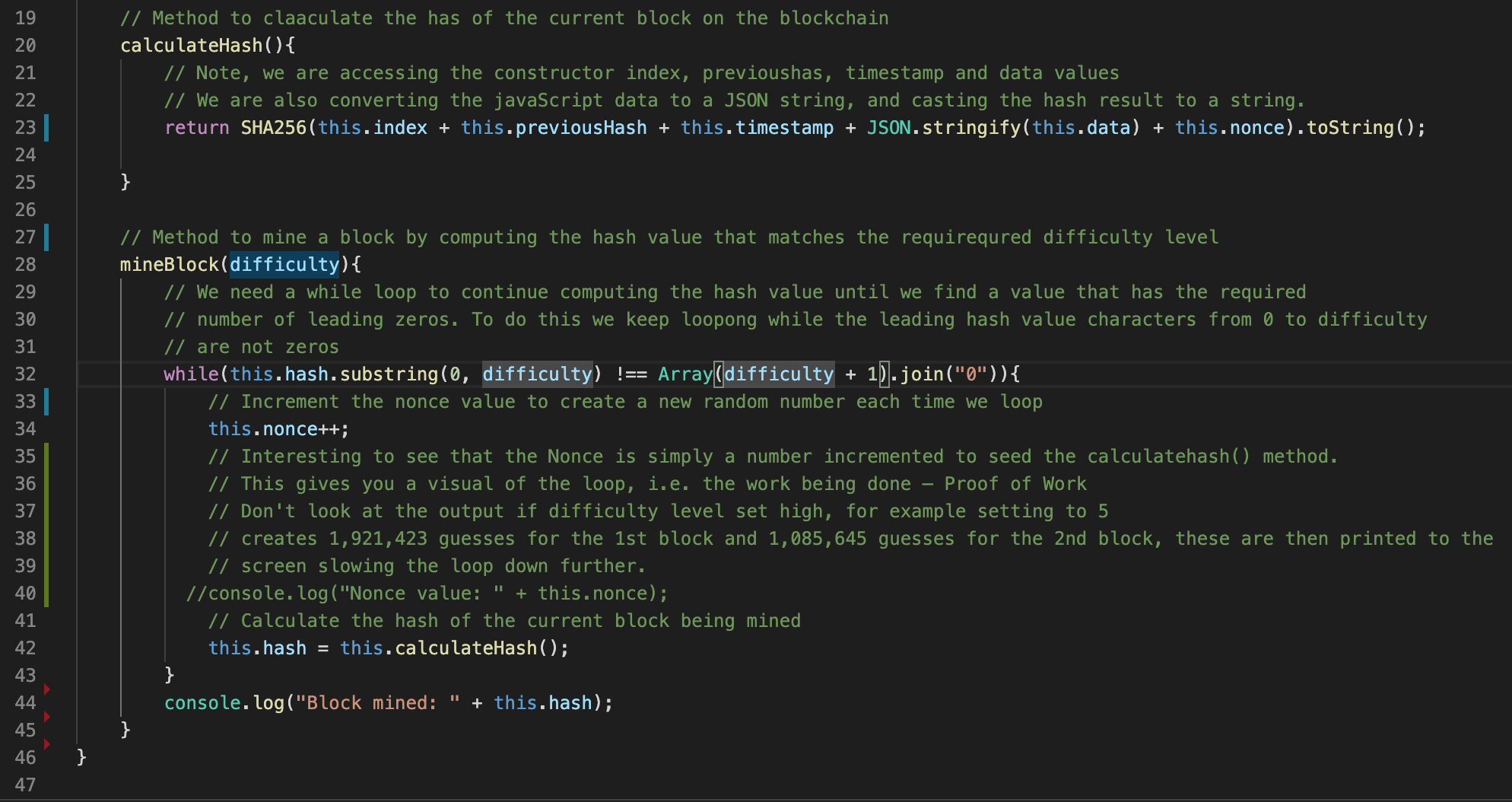


Figure 19: Create the mineBlock() method to mine (find a matching hash value) based on leading zeros

* Code explanation
  + Line 28 - 45: Create the mineBlock() method that takes an input parameter ‘difficulty’ used to modify the time taken to mine a block by making it continue mining until it finds a hash value with the required number of leading zeros indicated by the difficulty level. You can see the analogy: A miner is picking away at the rock finding all sorts of rubbish minerals (bad hash values), until suddenly a diamond or gold is found (matched hash value), in this case the leading zeros are the diamonds or the gold.
  + Line 32 – 43: Set up a while loop to loop round the calculateHash() method until the require leading zeros are matched in a returned hash value.
  + Line 32: To find the hash value with enough leading zeros we take a substring of the value of the current hash ‘this.hash’ starting with 0 and ending with the character indicated by the difficulty level. So, if difficulty is set to 2, this will take the first two characters of the hash value, which we then need to check if they are both zeros. Similarly, if difficulty is set to 5, we look at the first 5 characters to see if they are all zeros. So, we keep running this loop as long as we are not finding the leading zeros. To do this we set up an array of length difficulty + 1, e.g. 6 if difficulty is 5 and join this with the character zero. This is a quick trick to make a leading string of 0’s that is exactly the length of difficulty. To be fair, I am not a JavaScript Array expert, I can never get my head around where an array starts [0], its length, and indexing the array to find values. So, I found myself wondering why difficulty + 1. But, if you test this in jsfiddle it becomes clear as Figure 20 demonstrates. We can see that the .join() method is placing a 0 between each array entry, which in this case is filled with 5 food items. As, in this case, we require 4 leading zeros then we need 5 food items. Delete the food items and just enter 5, i.e. Difficulty + 1, and we get only 4 zeros returned, with which to match our hash value, Figure 21.

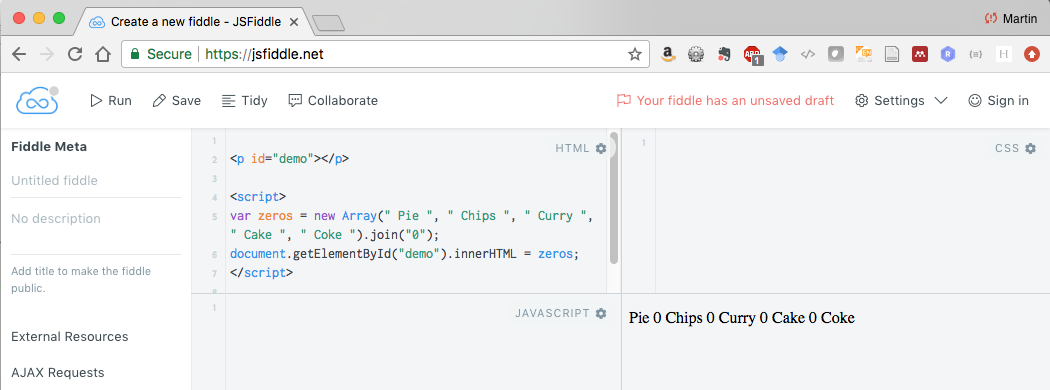


Figure 20: Load an array with food and join each food item with a zero

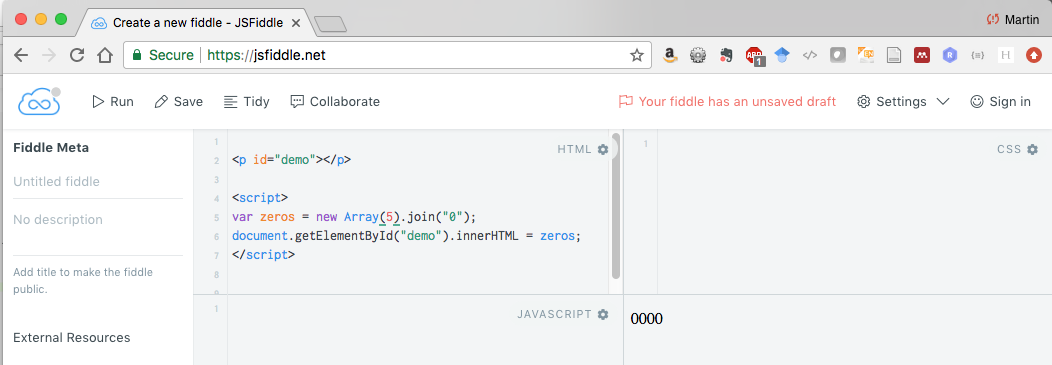


Figure 21: Join leading zeros to an empty array

* + Line 42: Here we now calculate the hash of the current block being mined. But, we have an issue, we need to be able to change the hash of the block being mined because as it stands we will be stuck in an endless loop. We cannot change the index, previousHash (else we will lose connection to the previous block), the timestamp, or the data because we will invalidate the block, etc. However, we solve this problem by introducing a random number, which happens to be called a nonce, that we append to the hash function calculation. We will increment this nonce value to produce a new random nonce value at each loop. We also have to take account of this nonce value in the calculateHash() method.
  + Line 34: Increment the nonce value.
  + Line 44: we output the hash of the block mined using the console.log() function.

1. Add the nonce variable to the Block class and modify the caculateHash() method.

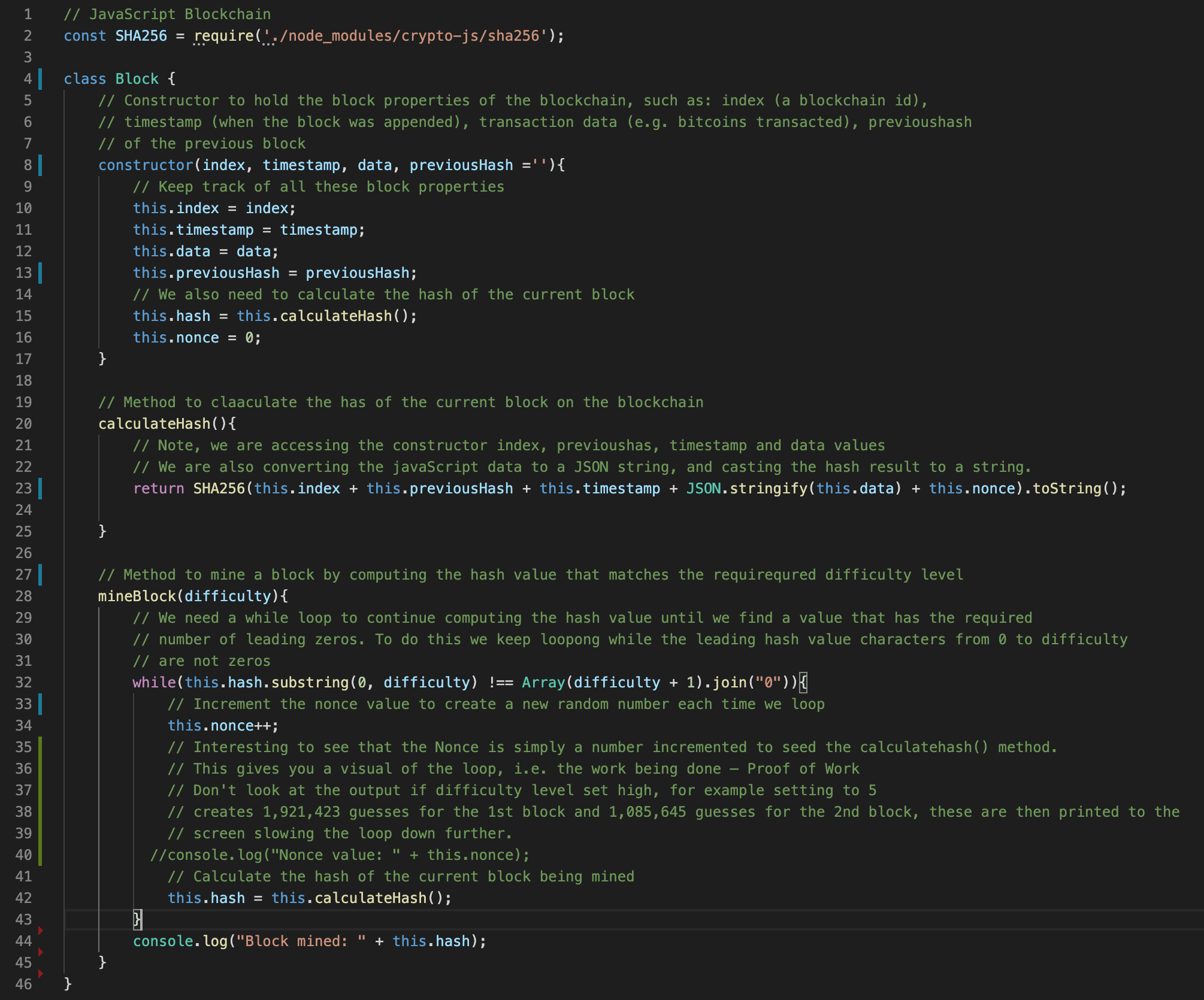


Figure 22: The new completed Block class with the nonce added

* Code explanation
  + Line 16: We have added the nonce and set it to zero.
  + Line 23: We append the nonce value to the input parameter for the SHA256 hash function. Because the nonce value is being changed every loop in the while loop, this in effect changes the message being input to the SHA256 function every time resulting in a new random hash being generated. The while loop in line 31, as we recall, is continually executing until it finds a hash with the required number of lead zeros. Don’t set difficulty to 10, like I did. It gets a bit boring waiting for it to find a hash value with 10 leading zeros!

1. Next we need to change the addBlock method in the Blockchain class to use the mineBlock() method rather than using the old newBlock.hash = newBlock.calculateHash() code, Figure 23.

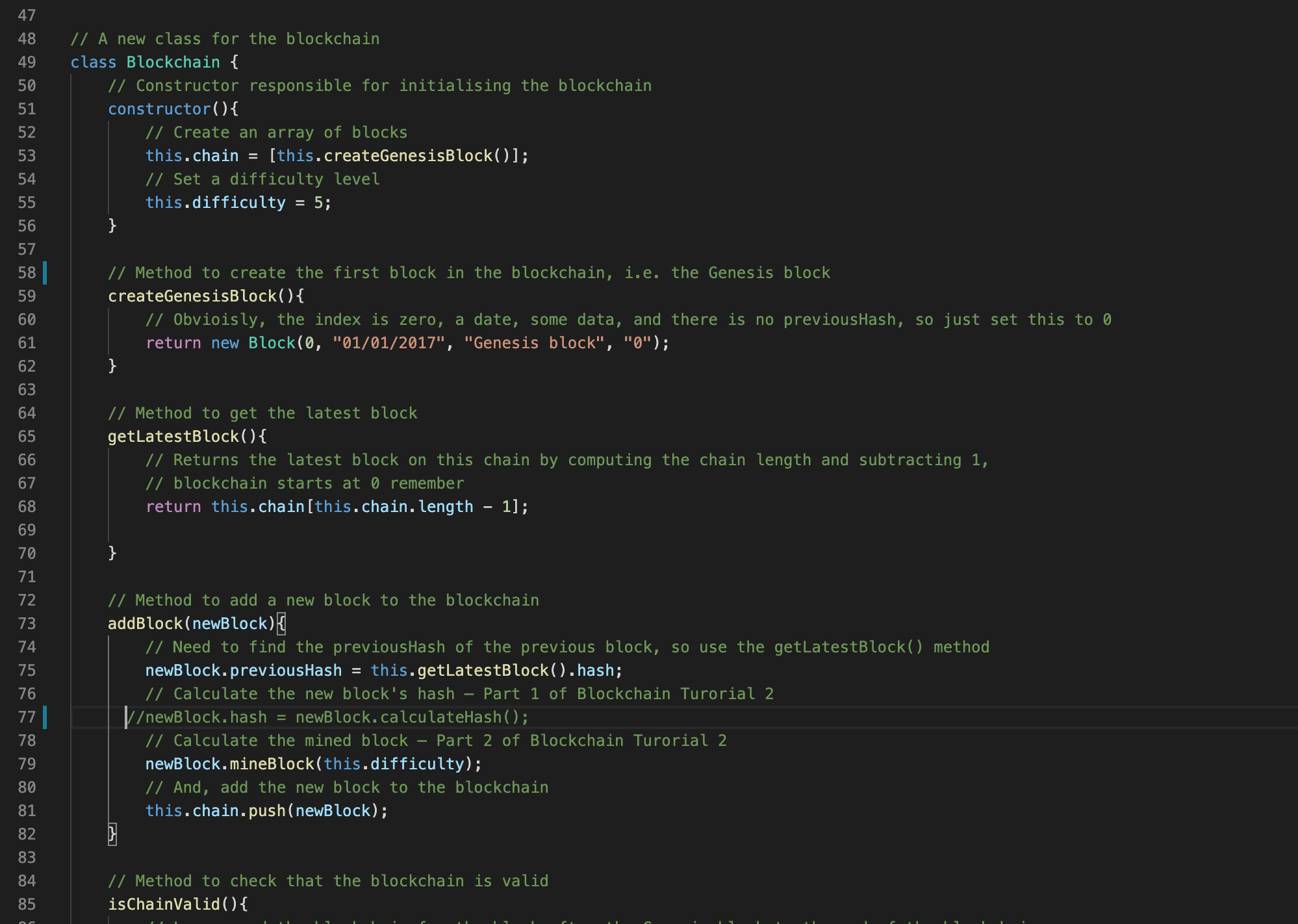


Figure 23: Modify the addBlock() method to use the minedBlock() method.

* Code explanation
  + Line 77: Comment out the old newBlock.hash = newBlock.calculateHash() code
  + Line 79: And, call the mineBlock() method with input parameter ‘difficulty’.
  + Line 55: So, we need to set up the difficulty level in the constructor for the Blockchain class.

1. Now we can test our code. Go to the bottom of your JavaScript file and comment out the old test code, and add a couple of console.log() functions as indicated in Figure 24.

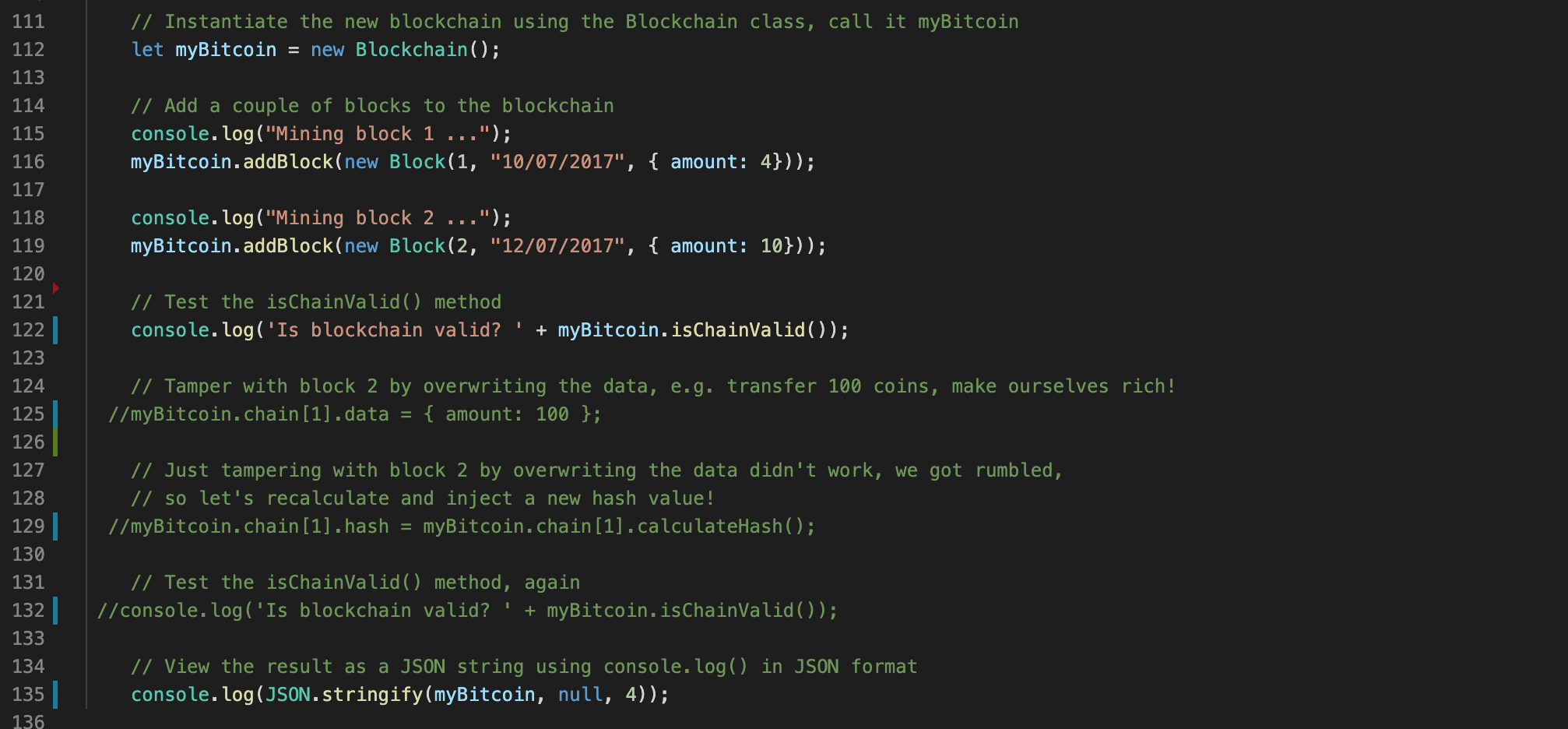


Figure 24: Comment out the old test code and add a couple more console.log() functions

* Code explanation
  + Line 115: Add a console.log() to indicate that mining is going on, when mining completed, the console.log("Block mined: " + this.hash) in the mineBlock() method of the Block clkass will also output the mined hash value.

1. Run your code as indicated in Figure 25 and check the results are correct. You will notice the code executes quite quickly with a low difficulty level, e.g. 2. It doesn’t take the JavaScript SHA256 function long to find matching hash values with 2 leading zero through random guesses — you could try putting some trace console.log statements in to output the guesses, probably in the hundreds for difficulty level 2 to over a million for difficulty level 5. Change the default level to 5 and observe. You will notice a significant increase in time to find matching hash values (i.e. mine our block) with 5 leading zeros. Don’t try 10, like I did, you’ll be up all night — JavaScript is relatively slow! Anyway, you can see the effect, the whole point of organising mining as a process of finding a given number of leading zeros in the hash value is to deliberately slow down the creation of the block to prevent attackers being able to recalculate all transactions and all hashs in a chain, thus creating a new valid chain, even though it would be false. Such actions would destroy the confidence of a cryptocurrecy like bitcoin. So, that essentially demonstrates the concept of Proof of Work as a mechanism to control how fast we add a block to the blockchain, and why it is essential for the safety of the blockchain.

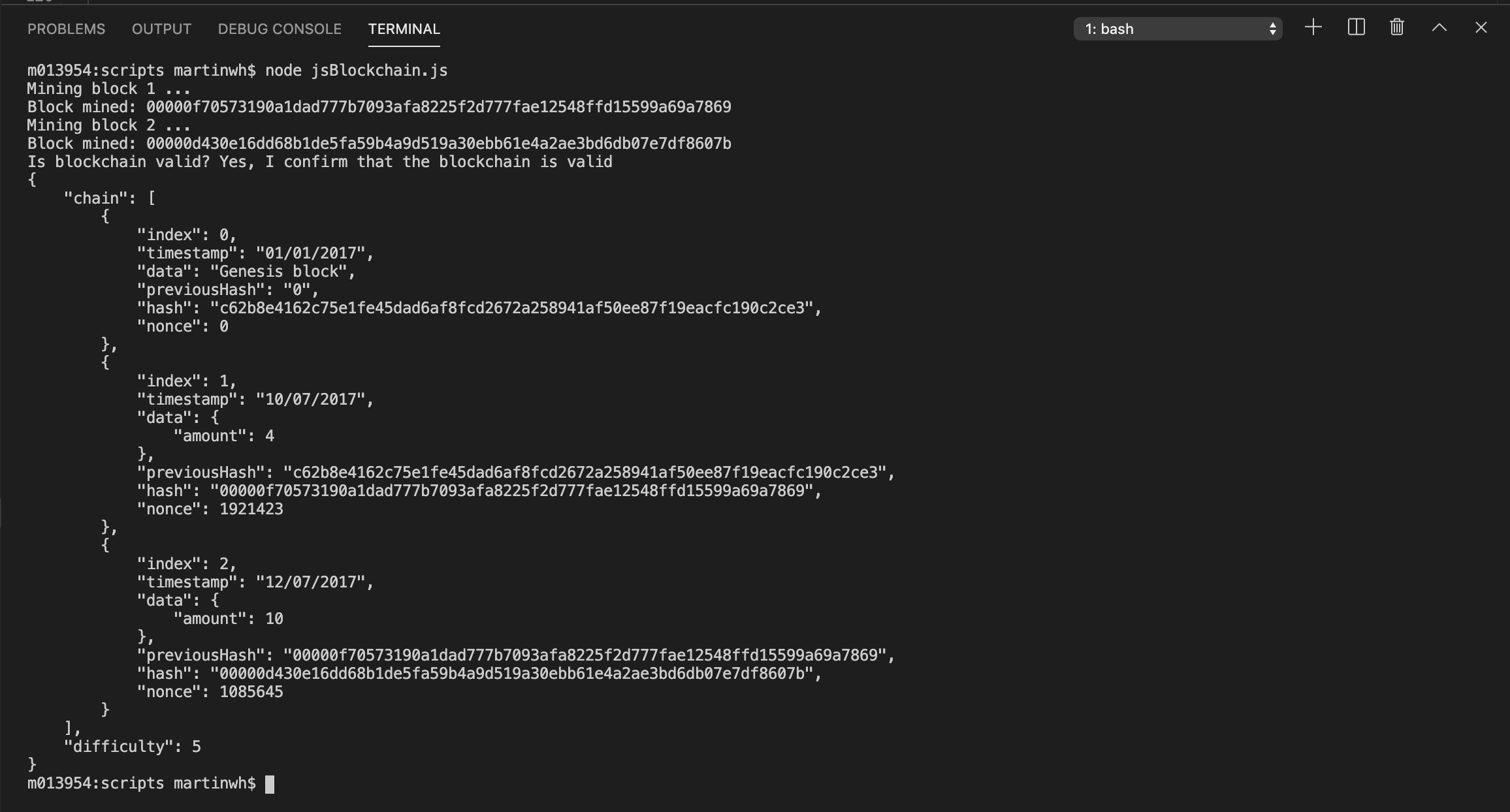


Figure 25: run your code in the terminal command line.

Ok, that is the end of Part 2 Blockchain Tutorial 2.

To complete the last three parts of this tutorial you will need to watch and follow the video tutorials:

1. [Miners Rewards and Transactions](https://www.youtube.com/watch?v=fRV6cGXVQ4I) — Blockchain in JavaScript (Part 3).
2. [Signing Transactions](https://www.youtube.com/watch?v=kWQ84S13-hw) — Blockchain in JavaScript (Part 4).
3. [Angular Frontend](https://www.youtube.com/watch?v=AQV0WNpE_3g) — Blockchain in JavaScript (Part 5).

1. https://www.huffingtonpost.com/ameer-rosic-/what-is-bitcoin-mining-a-\_b\_13764842.html [↑](#footnote-ref-1)
2. <https://bitsonblocks.net/2015/09/14/inside-bitcoins-blockchain/> [↑](#footnote-ref-2)
3. https://www.investopedia.com/terms/1/51-attack.asp [↑](#footnote-ref-3)
4. https://bitzuma.com/posts/six-things-bitcoin-users-should-know-about-private-keys/ [↑](#footnote-ref-4)
5. https://en.bitcoin.it/wiki/Address [↑](#footnote-ref-5)
6. <https://www.investopedia.com/terms/d/doublespending.asp> [↑](#footnote-ref-6)