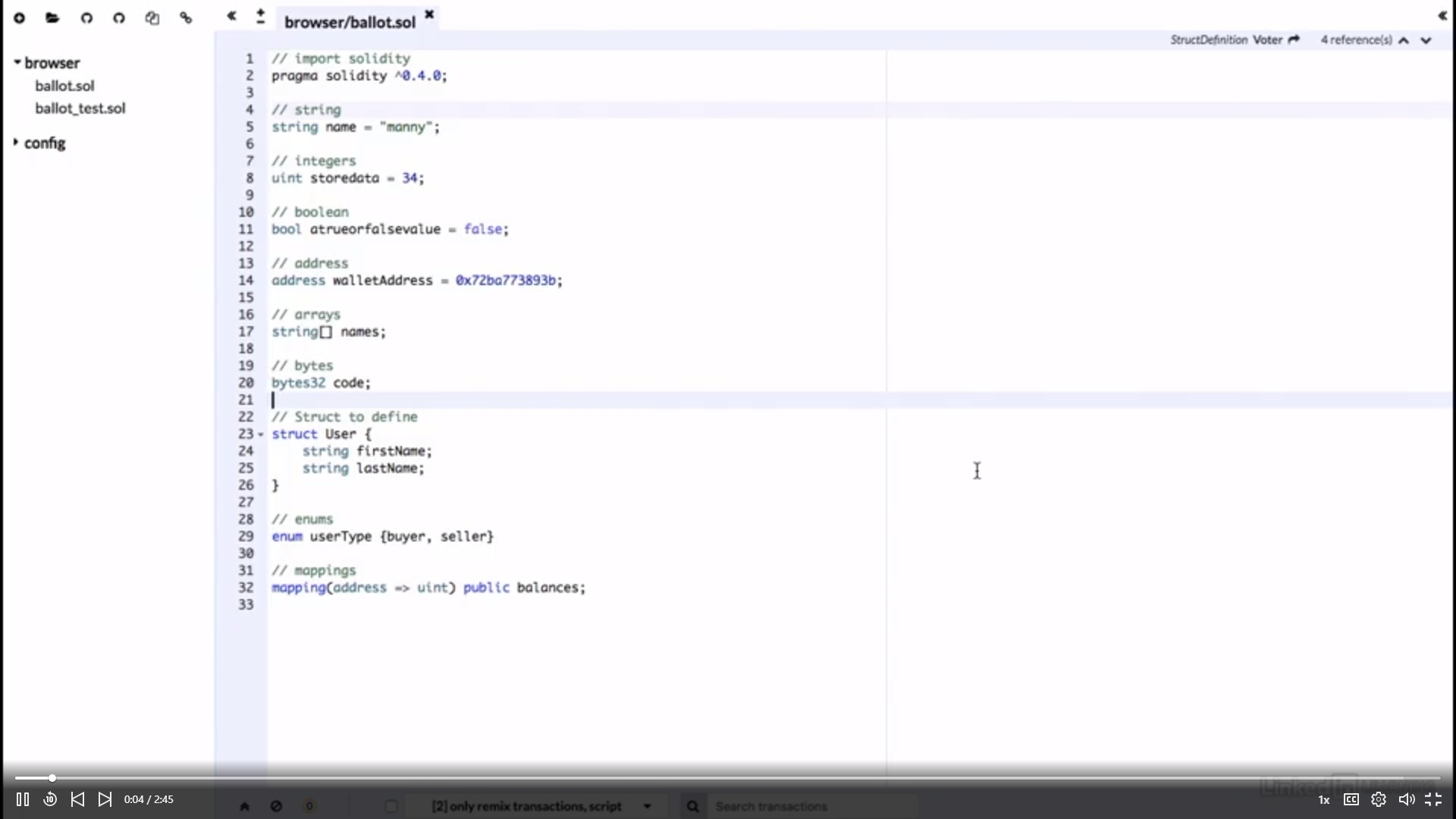
Refunding gas-fees in Qtum is enabled by creating new outputs as part of

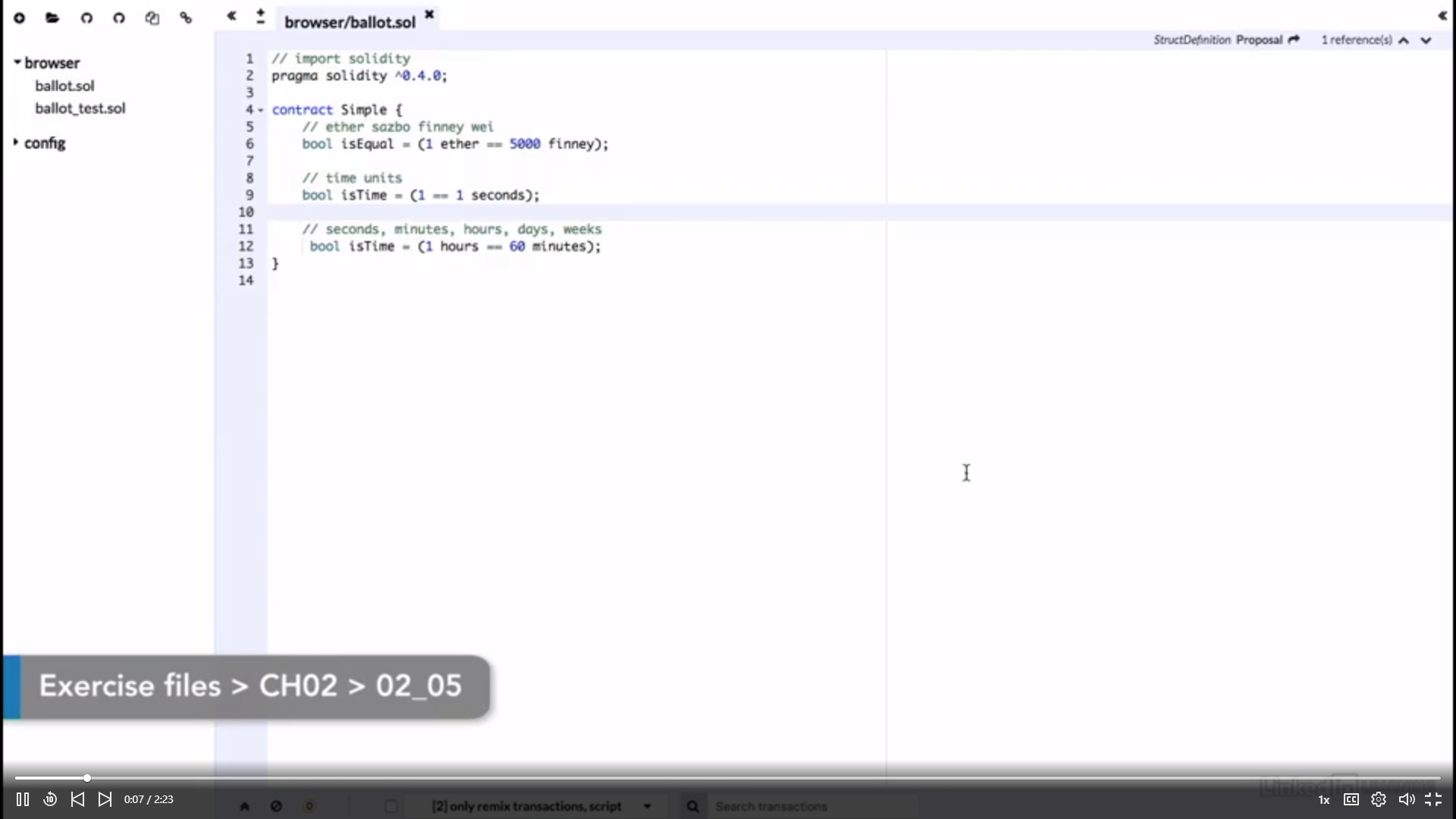
the coinbase transaction of a miner. – page 14

Oraclize

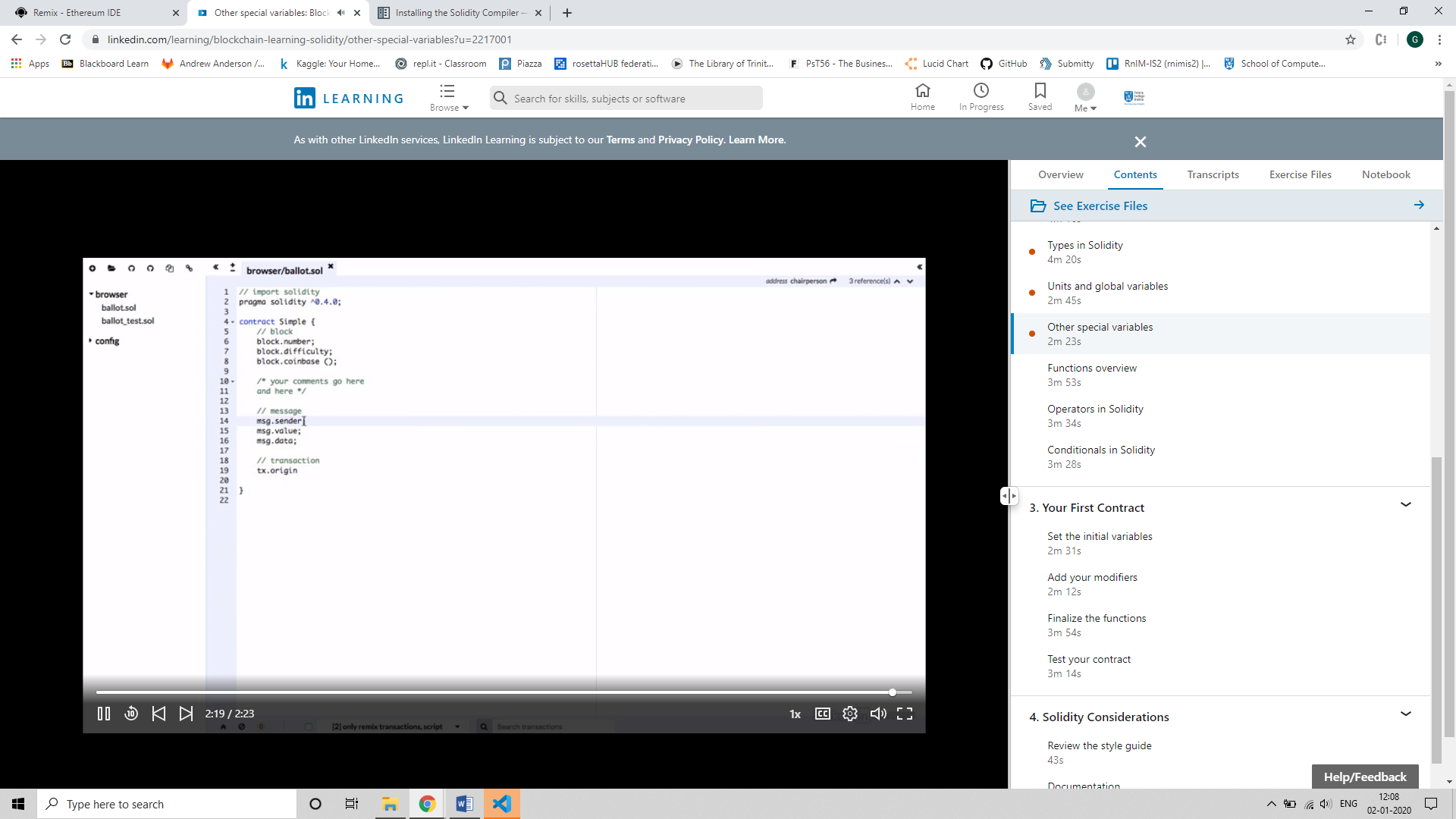
Dev.oraclize.it

**Solidity**

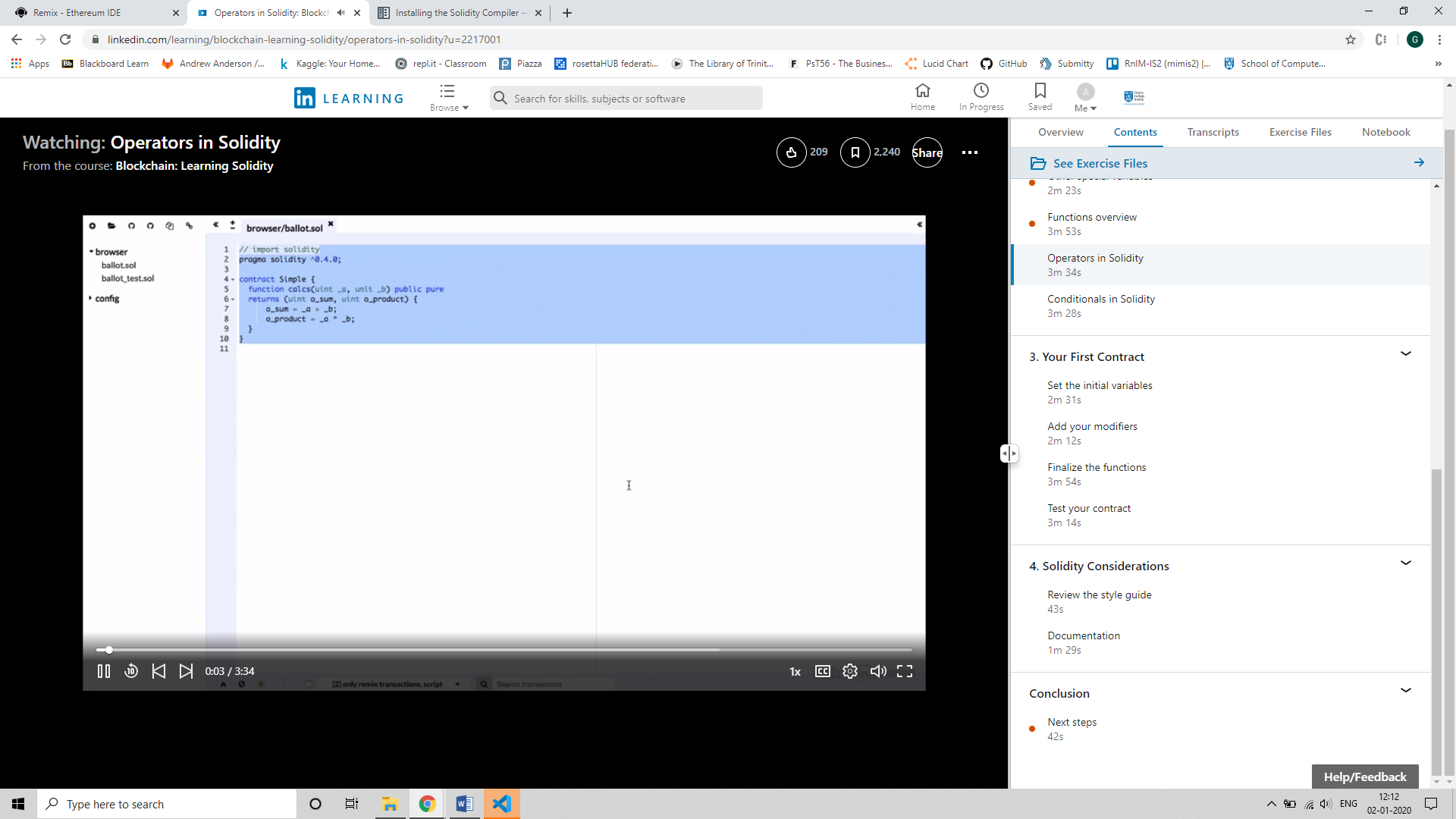
* Ethereum is a platform to build apps and solidity is the language.
* Use Remix to explore its syntax.
* IDE – Intellij, Visual Studio and VS Code.
* Ethereum is the environment, EVM executes the code, and Solidity is used to program smart contracts.
* 



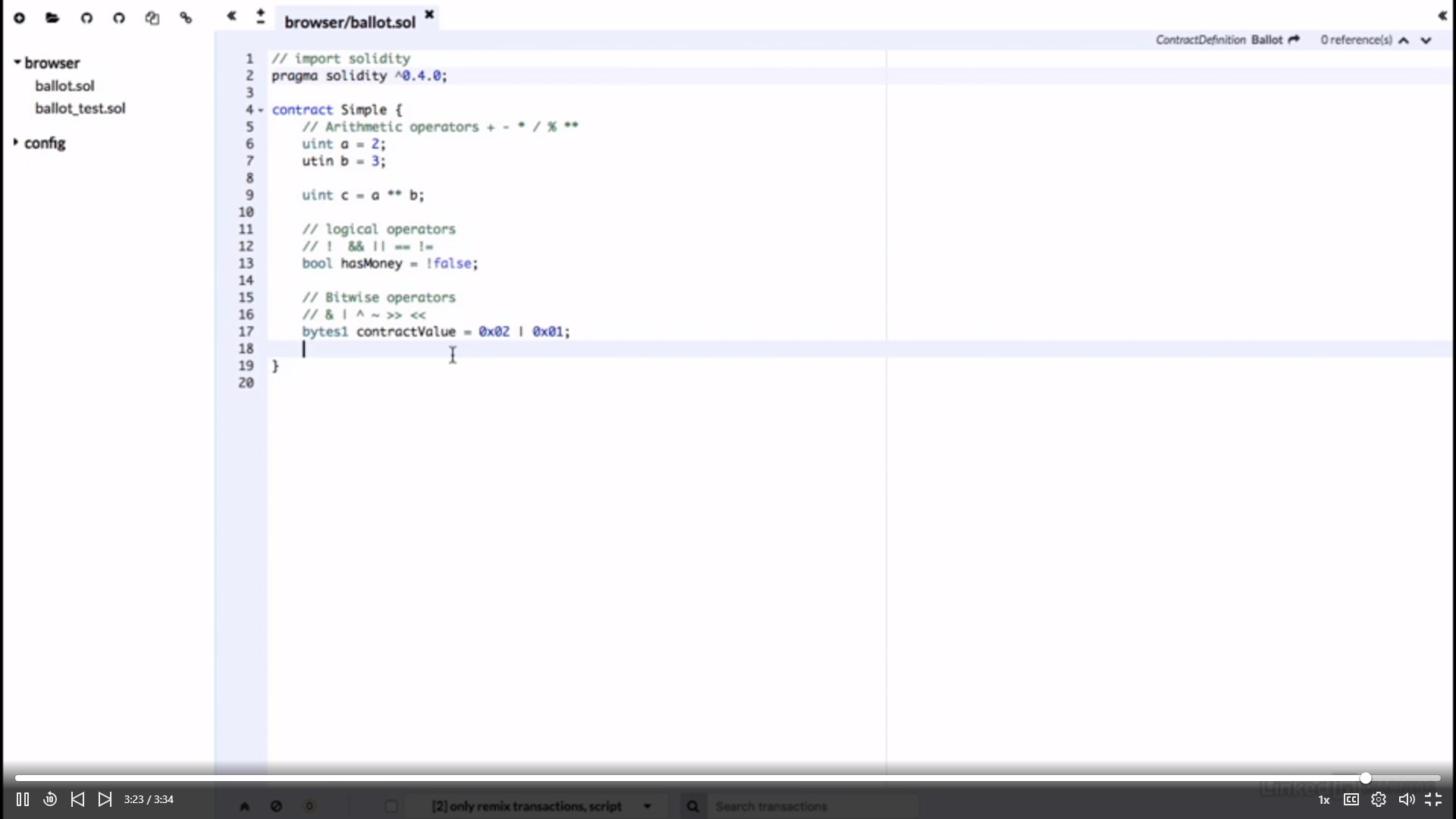
Note: access the miners address using the block.coinbase variable.



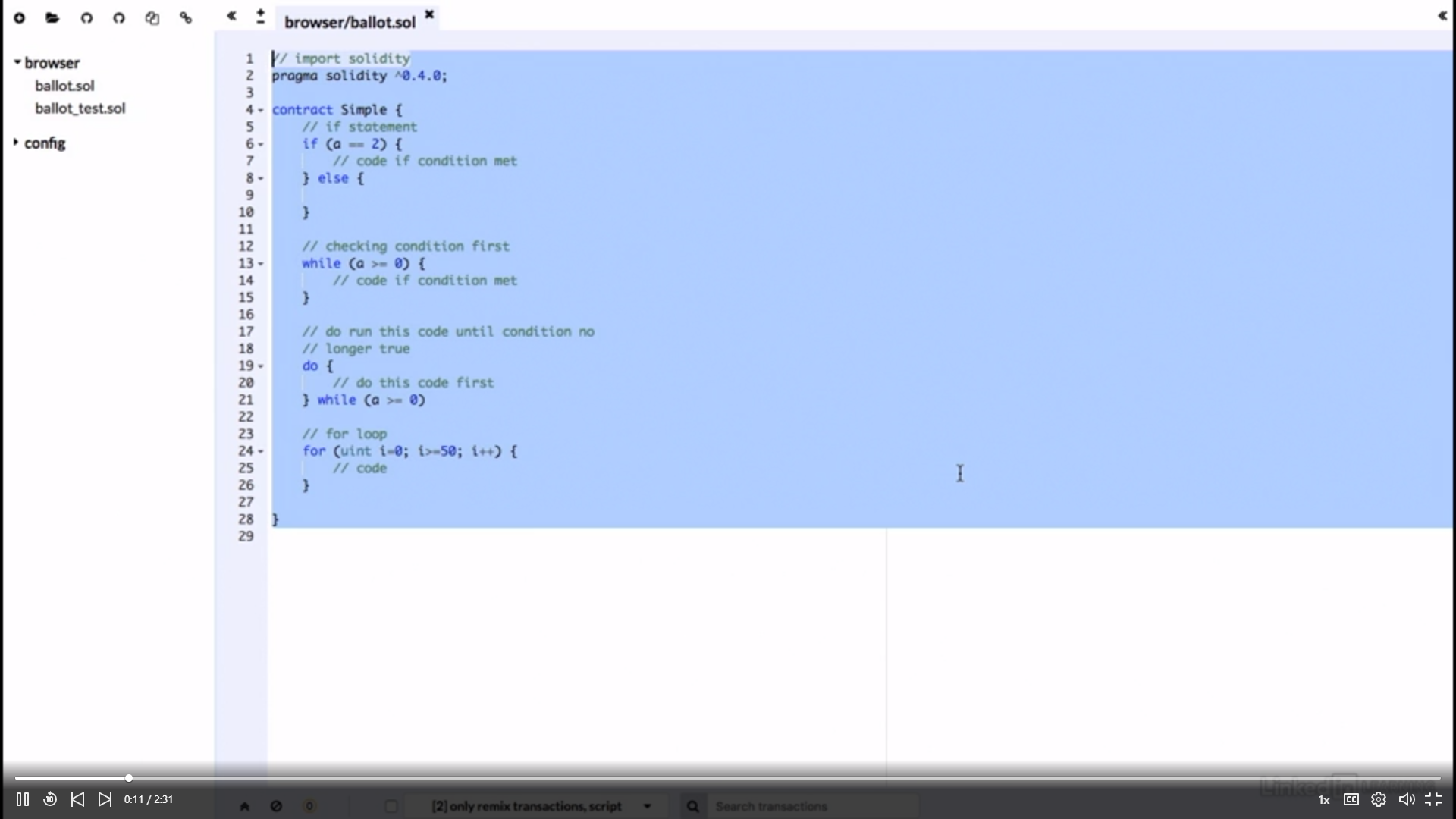
Basics of functions:

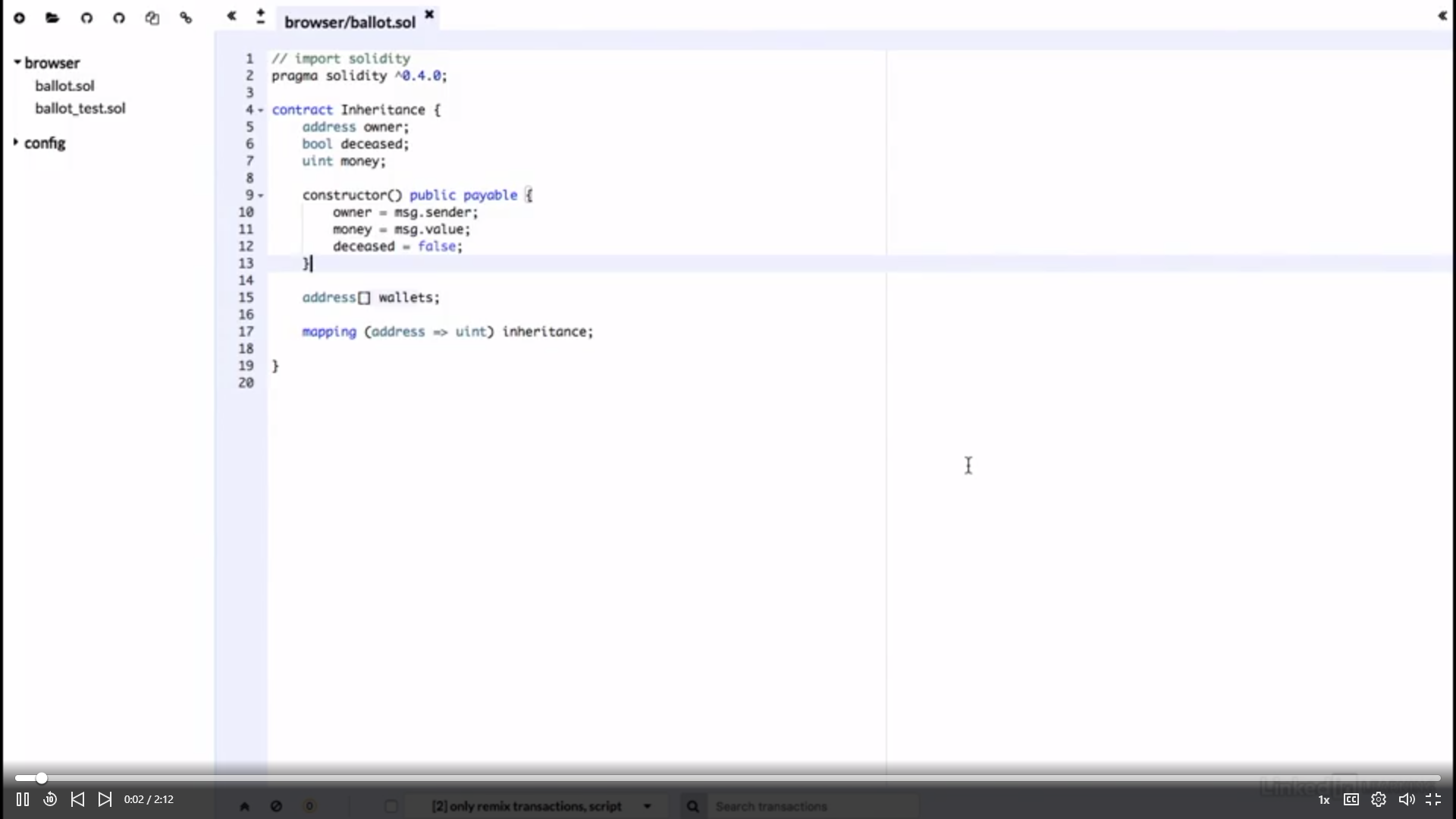


Operators in Solidity



Loops:





Example code:

pragma solidity ^0.4.22;

import "github.com/provable-things/ethereum-api/provableAPI\_0.4.25.sol";

contract WeatherContract is usingProvable {

string public sunsetTime;

event LogConstructorInitiated(string nextStep);

event LogPriceUpdated(string price);

event LogNewProvableQuery(string description);

function Constructor() payable {

LogConstructorInitiated("Constructor was initiated. Call 'updatePrice()' to send the Provable Query.");

}

function \_\_callback(bytes32 myid, string result) {

if (msg.sender != provable\_cbAddress()) revert;

sunsetTime = result;

}

function update() payable {

LogNewProvableQuery("Provable query was sent, standing by for the answer..");

provable\_query("URL", "json(https://api.pro.coinbase.com/products/ETH-USD/ticker).price");

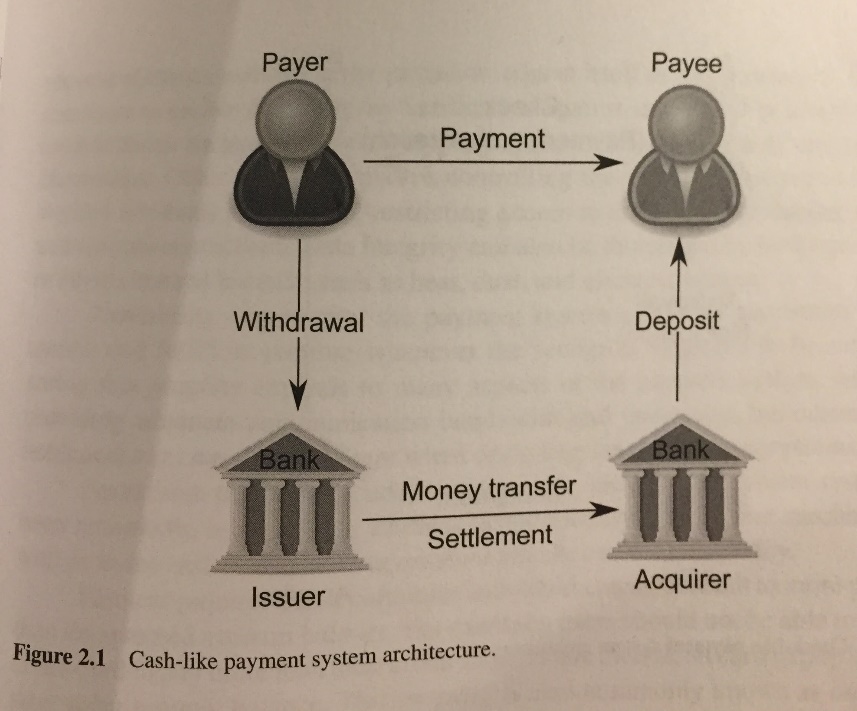
}

}

**Book on Bitcoin and Blockchain Security by Ghassan Karame**

**Digital Payments:**

**Architecture**



**Security** aspects are – fairness, non-repudiation (user should not be able to deny a transaction, resistance to impersonation attacks and accountability.

**Methods** – Authentication (passwords), Authorization (role based access), Encryption, Tokens

**Privacy** – transaction anonymity (one cannot link a particular transaction to a specific identity) and transaction unlinkability (two transactions of a user cannot be linked as such).

**Bitcoin Protocol Specification:**

* Loosely connected P2P network. (Computers connected together in a network – no central servers, every computer acts both as a server and a client)
* Bitcoin nodes are connected to the Overlay network
* Peers bootstrap to network by requesting peer address information from Domain Name System seeds that provide a list of current Bitcoin node IP addresses.
* A Bitcoin client establishes a maximum of 125 TCP connections.
* Each address maps to a public/private key pair used to transfer ownership of BTCs among addresses.
* Each Bitcoin address is computing using Elliptic Curve Digital Signature Algorithm (ECDSA) public key – for which the address owner knows the corresponding private key using a transformation based on hash functions.
* Hash functions are one-way functions. Meaning it is possible to compute an address using a public key, but it is infeasible to retrieve the public key solely from the Bitcoin address.

The **InterPlanetary File System (IPFS)** is a protocol and peer-to-peer network for storing and sharing data in a distributed file system. IPFS uses content-addressing to uniquely identify each file in a global namespace connecting all computing devices.

**Working of Oracles in Ethereum Blockchain:**

* Create a smart contract that inherits oraclize/provable.sol that follows the Ethereum protocol. The contract that is created has to be ‘payable’ since we have to send a little bit of Ether to pay the Oraclize service.
* The program has function that queries the outside world or an external source to get data that is used by another smart contract. The program is then called from the outside world.
* Then, another function is created called - \_\_callback() that has two parameters viz., bytes32 \_provableID and string \_result. This is called by the Oraclize service in our contract with two parameters – provableID and result. ProvableID can be used to check which particular request/query is getting called back.

**Note:** \_\_callback() function can be called multiple times by several requests. Requests can be identified by the unique id stored/passed in the provableID variable.

Steps to deploy the contract:

* Compile the contract. Copy the byte code.

E.g.,

{

"linkReferences": {},

"object": "

Use a wallet. Deploy the contract (which includes pasting the byte code, adding Ether and signing the transaction)

* Once deployed, gives a TX hash. The deployed contract is an address which can be called into.
* Declare var deployed = eth.web.contract(ABI) ; ABI is got from the solidity compiler.

E.g.,

[

{

"constant": false,

"inputs": [

{

"name": "myid",

"type": "bytes32"

},

{

"name": "result",

"type": "string"

}

],

"name": "\_\_callback",

"outputs": [],

"payable": false,

"stateMutability": "nonpayable",

"type": "function"

},

{

"constant": true,

"inputs": [

{

"name": "",

"type": "address"

}

],

"name": "balances",

"outputs": [

{

"name": "",

"type": "uint256"

}

],

"payable": false,

"stateMutability": "view",

"type": "function"

}

]

* Deployed.result.call() helps call the result variable in the contract at this stage returns null.
* Go to the section in the wallet that helps us interact with the contract where we enter the contract address, the ABI and the private key.

// Private Key

23e4dfb937932332a3cf17f1fc0541e7dc2b7d0bef81289bfa04a251443b5666

// Address

0x08Cf02070Bb9F167556C677DA58e6678bbE871fc

MEW password: Four0992$

// Etherscan URI

https://testnet.etherscan.io/address/0x08cf02070bb9f167556c677da58e6678bbe871fc

* We get a list of methods/function we could call in the contract. Call the function which has your required query in it. E.g., flip a coin. This operation asks for the amount of Ether you would like to send.
* After which the transactions are recorded in the logs – 1. Called function and 2. \_\_callback() function which is called by the Oraclize service.
* Now call the Deployed.result.call() function. We see the result stores the response from the Oraclize service.

**25-2-2020**

The main solution is to provide an Oracle contract on the blockchain, which serves off-chain data requested by user smart contracts. While deploying smart contract with Oracle, original user smart contract needs to inherit extra smart contract, namely Oracle resolver, with a predefined standard on data format. The Oracle resolver is responsible for interacting with Oracle contract, which is designed to present a simple API to a relying user contract for its requests to external data source. As shown in Fig. 1, Oracle contract accepts query datagram from Oracle resolver and generates event log to external agent for fetching off-chain data. At the end, external agent will launch a callback and return corresponding data for user contract.

**Codius**

<https://github.com/codius/codius-wiki/wiki/White-Paper>

**Chainlink, Komodo and Augur are various other oracle creation projects**

Smart oracles build on the idea of oracles, or entities that provide smart contracts with information about the state of the outside world, and combine information gathering with contract code execution. In such a system, rules can be written in any programming language and contracts can interact with any service that accepts *cryptographically signed commands*.

Some smart contracts systems, including the one built into Bitcoin, are strictly deterministic. In order to interact with the real world, these systems rely on *cryptographic signatures submitted by outside systems called "oracles."*

**Contracting Parties**

The contracting parties are the people or businesses who agree to use a smart contract to carry out an arrangement. Other relevant, but possibly distinct entities, are the contract author and the contract owner. The contract author is the one who wrote the code, though they may not be involved in the particular arrangement at all. For example, the author could be a developer or group of developers that have published an open source auction contract. The contract owner is the entity that sets it up to be executed by the smart oracle(s).

Note that the contract host should not be one of the contracting parties or affiliated with any of them.

**Public/Private Key Cryptography**

Public/private key cryptography enables messages to be encrypted, or translated into a seemingly random set of characters. When a message is encrypted with the public key, only the holder of the corresponding private key can decrypt or decipher it.

Public/private key cryptography also enables the holder of the private key to cryptographically "sign" messages. Anyone can definitively verify that the signature could only have been created by the holder of that private key.

Public/private key cryptography underpins some of the common use cases for smart oracles, so it may be useful to have a basic understanding of how asymmetric encryption and cryptographic signatures work.

**From Oracles to Smart Oracles**

The concept of smart contracts is widely attributed to Nick Szabo who, in the late 1990s, argued that formalizing relationships and encoding them in software and hardware would simplify and secure business logic and functionality. He wrote of embedding contractual clauses, such as bonds and property rights. Szabo used the example of the vending machine as a "primitive ancestor" of smart contracts because its hardware and software enforce a simple contractual agreement. Anyone who inserts money will receive a snack in return, even though no explicit contract was ever made with the machine’s owner. Wei Dei also wrote about digital contracts in his B-money proposal of the late 1990s, describing self-enforcing, cryptography-based contracts not too dissimilar to Szabo’s ideas.

**Challenges**:

Cryptocurrency developers have found it challenging to design a system that encompasses both a powerful smart contracts language and a robust consensus system. Bitcoin scripts allow for simple logic to be encoded and executed on the Bitcoin network. However, encoding advanced logic and executing untrusted code have proven more complicated to integrate.

**Forking the blockchain**, or distributed ledger, means creating multiple competing states of the network, which is a highly undesirable outcome for a consensus-based system.

**Codius:**

Codius and Google Native Client

Google’s Native Client is a sandbox for running untrusted x86 code, the low-level commands used by most computer processors. Native Client was developed to run compiled binary code on the web, as opposed to the HTML/CSS/Javascript that websites are normally limited to. Native Client makes a number of improvements on top of software fault isolation (described above) to provide a constrained execution environment that protects users from potentially malicious code.

Native Client can be used to run any programming language and currently supports C, C++, Python, V8 JavaScript, Ruby, Go, Mono, and Lua. Recent versions of NaCl support x86-32 and x86-64 architectures, as well as ARM and MIPS. Google uses Native Client for computationally intensive web apps, such as Hangouts Video and QuickOffice, among others, as well as ChromeOS apps and datacenter hosting of untrusted code. The latest benchmarks have shown that Portable Native Client modules run only 10-25% slower than LLVM-compiled native code, so Native Client is not only efficient to start up but it also provides performant execution.

*Codius uses Native Client because it provides a unique combination of security, performance, and flexibility.* Native Client is lighter weight than a VM and provides a much smaller attack surface than an entire operating system managing containers. VMs and containers may develop a better balance of performance and security in the future, but they do not meet our requirements at present.

We argue that requiring contracts to be written in a specific capability-based programming language would needlessly hamper the adoption of the system. Rather, if some authors or hosts prefer contracts to be written in a custom or capability-based language they can also do so within the Native Client sandbox and it will provide yet another security layer.

Software fault isolation and Native Client rely on a minimal trusted code base while being flexible enough to support all programming languages and allow for the reuse of already-developed and widely-used modules.

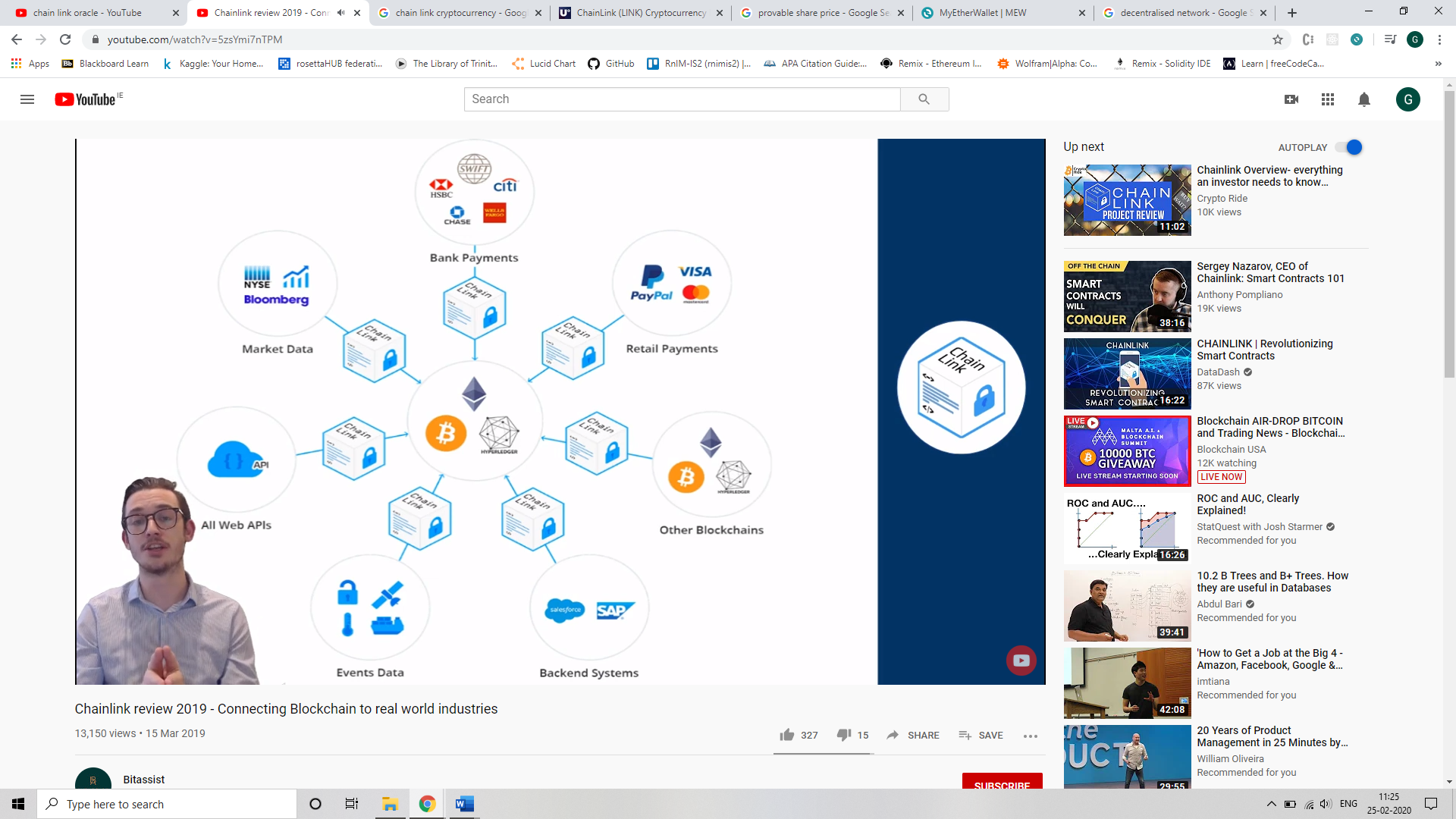
Reference: <https://github.com/codius/codius-wiki/wiki/White-Paper>

**Chainlink:**

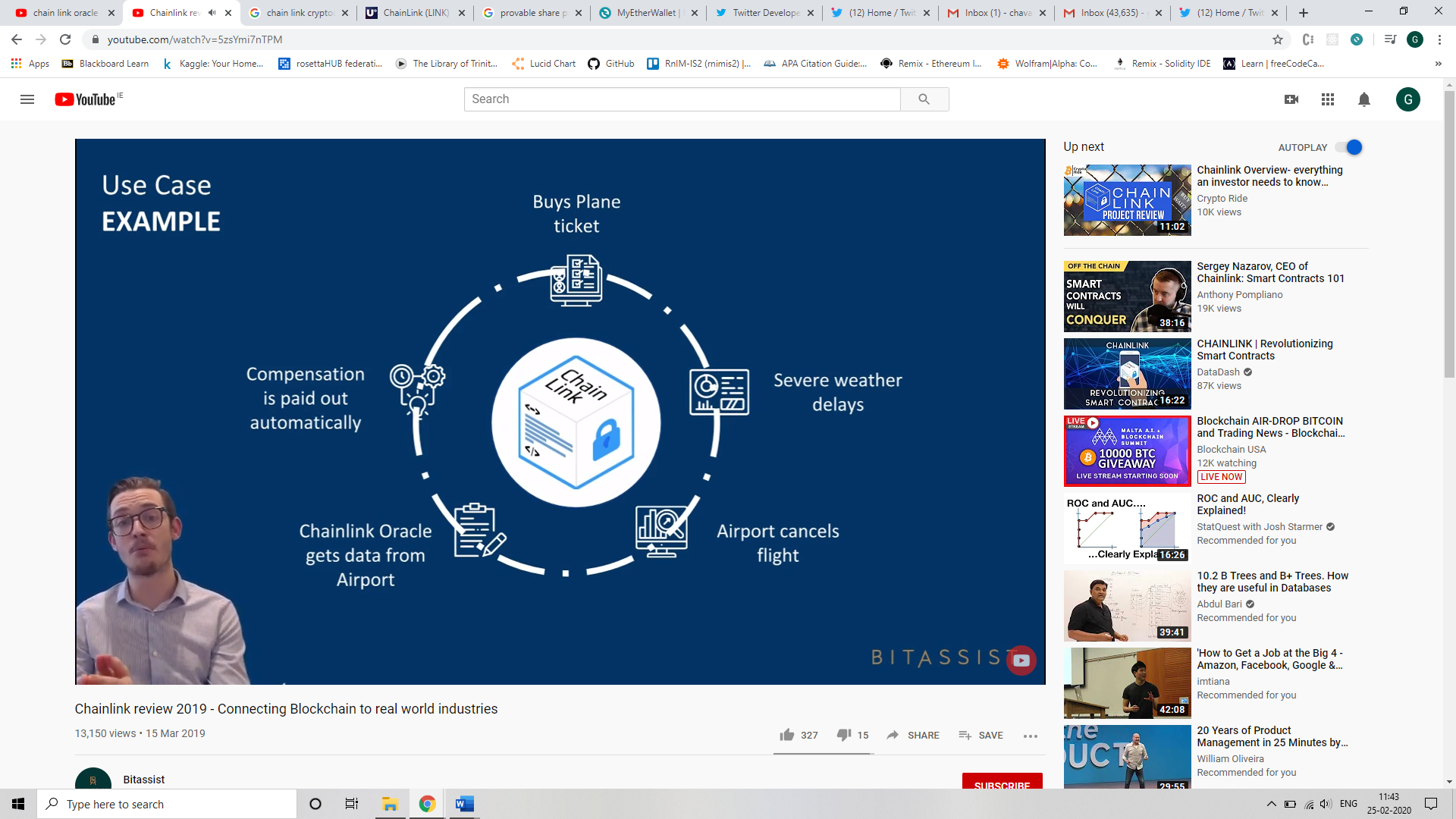
Key differences:

It has a decentralised network. A decentralized network is a “trustless environment,” where there is no single point of failure. The nodes connected in the network are not dependent on a single server point and each node holds the entire copy of the network configurations.

* Inbuilt reputation system, independent oracle service providers and bidding system.
* You earn link tokens.
* The link/oracle provider that is most profitable is the one that has the best reputation.
* Bad actors are removed.



* Chainlink takes data from external source, converts it into a right format and settles it within the blockchain.
* Usecases: flight statistics, sporting events, exchange rate data.



**Ropsten:**

Transaction hash: 0x93a3f9e2be0a720a005274bba0e7e6287c8d096b913be7f5738a4627fb32c622

<https://ropsten.etherscan.io/tx/0x93a3f9e2be0a720a005274bba0e7e6287c8d096b913be7f5738a4627fb32c622>

