Blockchains & Cryptocurrencies

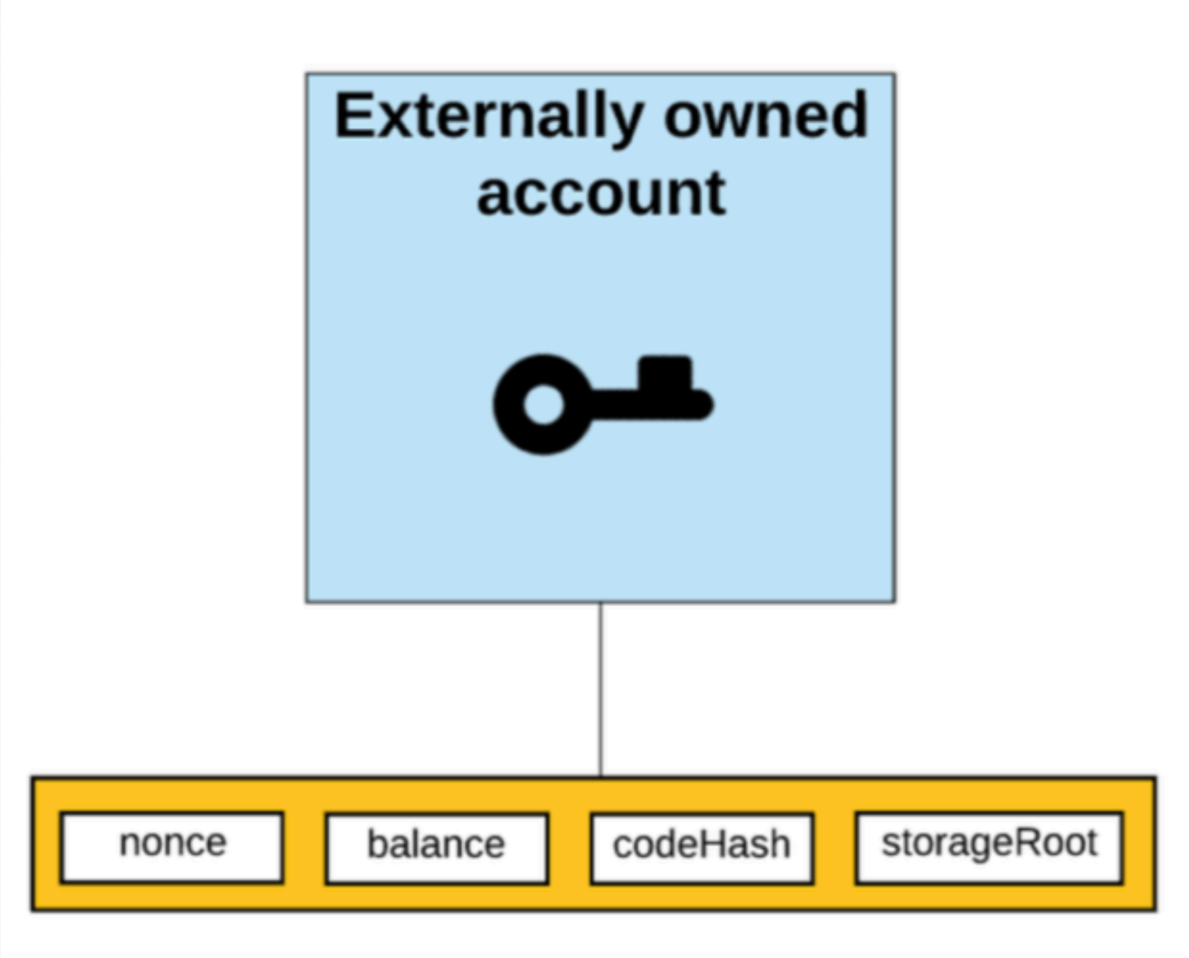
Smart Contracts / Ethereum In Detail (II)

Instructor: Matthew Green
Fall 2024

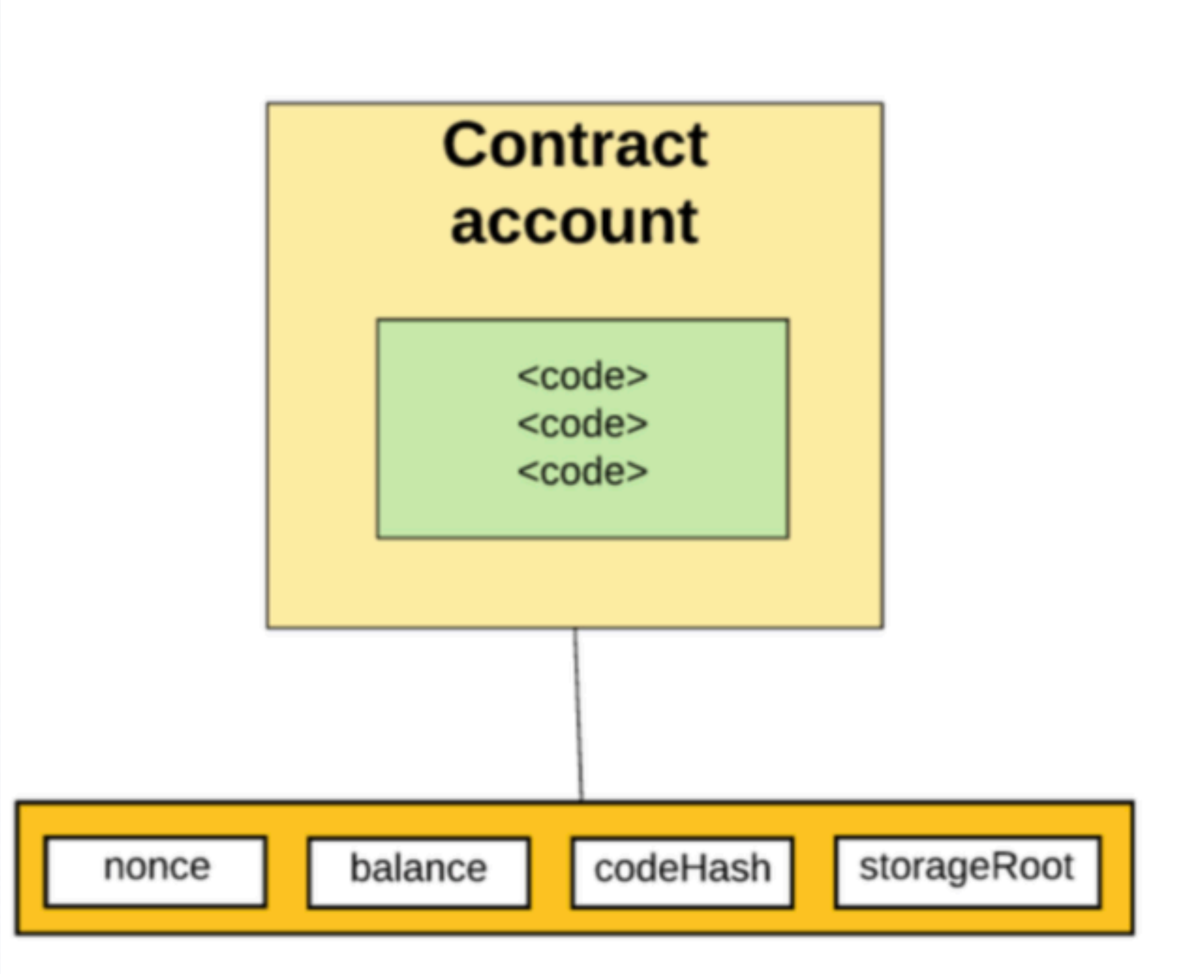
News?

Ethereum: Accounts

- Two types of account:
  - External (like Bitcoin), Contract accounts



Like Bitcoin, updates require an signature by an external private key



Anyone can call "methods" in the code, which trigger updates. Anyone can create.

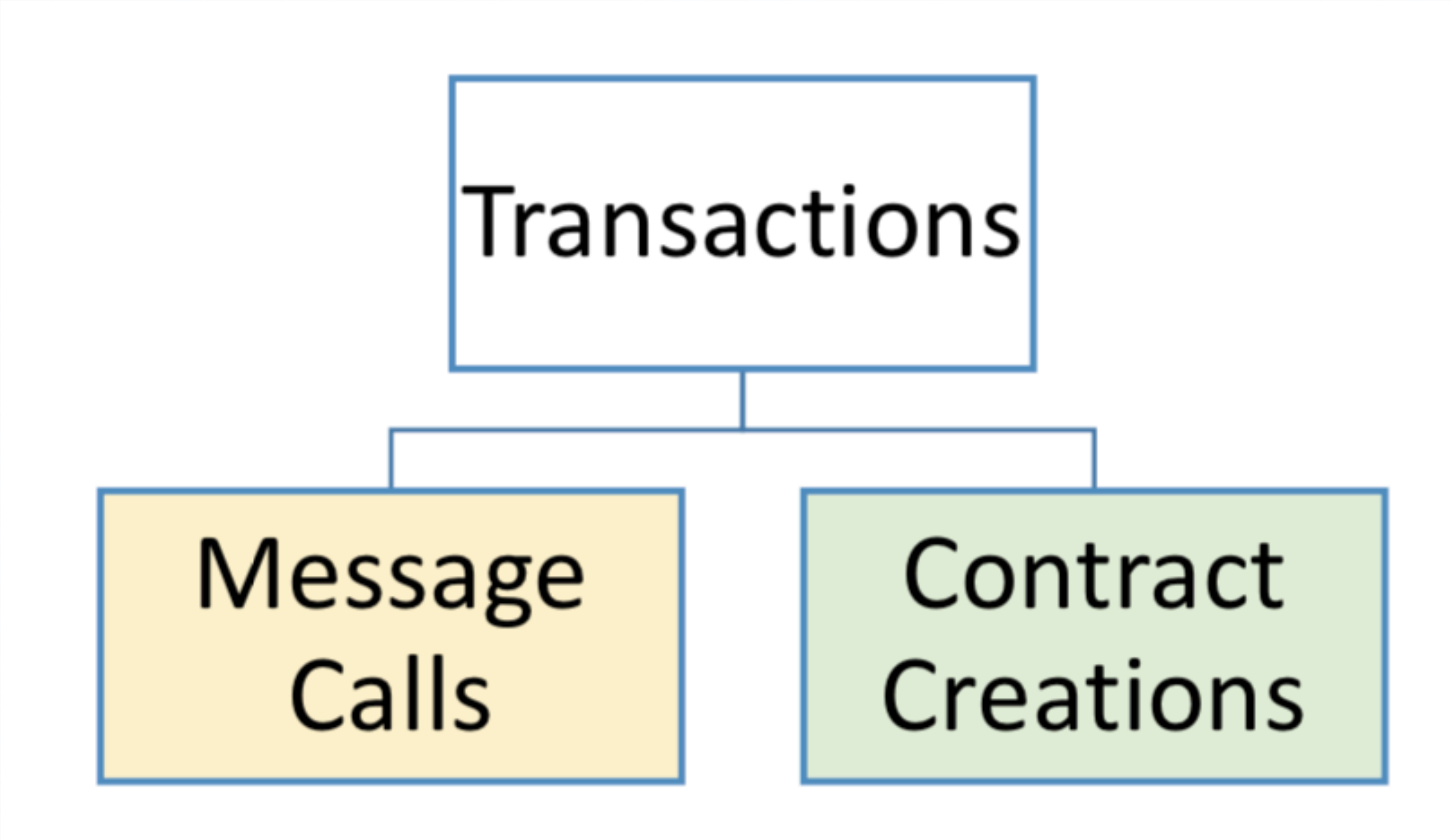
Ethereum: Accounts

nonce: # transactions sent/ # contracts created

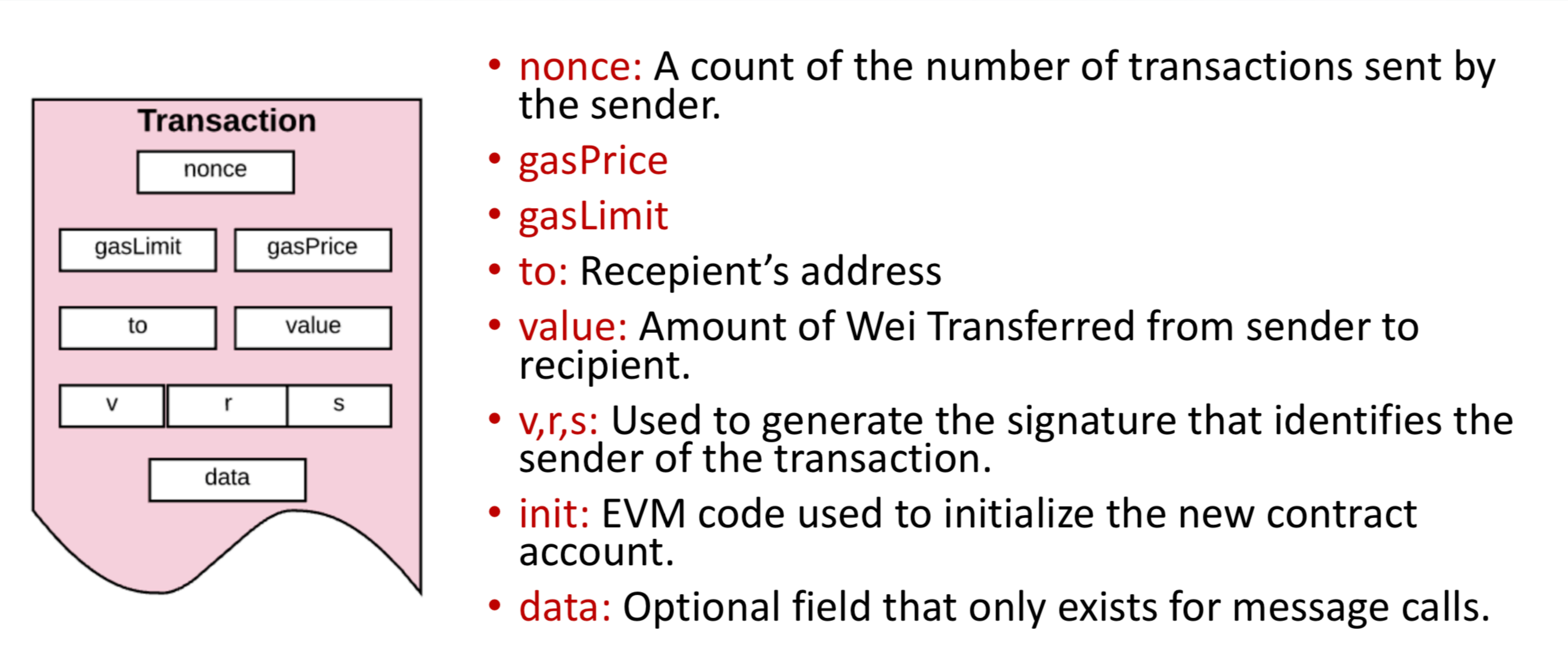
- balance: # Wei owned (1 ether=$10^{\#\$}$Wei)
- storageRoot: Hash of the root node of a Merkle Patricia tree. The tree is empty by defaul
- codeHash: Hash of empty string / Hash of the EVM (Ethereum Virtual Machine) code o this account

Ethereum Transactions

- Two types:
  - Message calls: update state in a given contract, by executing code (or simply transferring money)
  - Contract creations: make a new contract account, with new state



Ethereum Transactions



- nonce: A count of the number of transactions sent by the sender.
- gasPrice
- gasLimit
- to: Recepient’s address
- value: Amount of Wei Transferred from sender to recipient.
- v,r,s: Used to generate the signature that identifies the sender of the transaction.
- init: EVM code used to initialize the new contract account.
- data: Optional field that only exists for message calls.

Contracts

- Smart contracts are often written in a high-level object-oriented language (e.g., Solidity)
  - The "contract" object has "methods", which can be public or private (internal)
  - Public methods can be called by anyone, private methods can only be called from other methods within that contract
  - External transactions contain the contract address + the data (arguments) for a method call

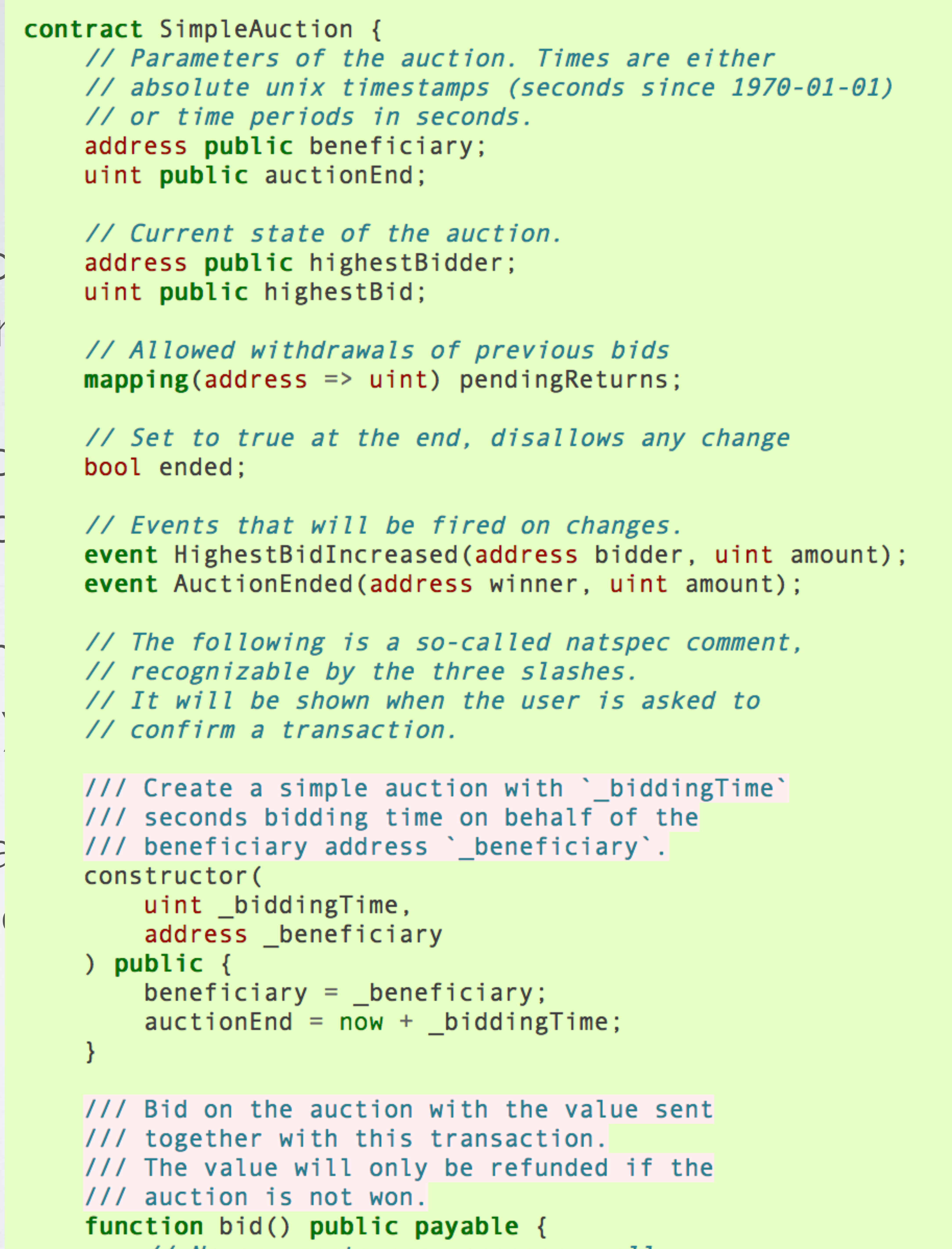
ns

- Smart co l
  object-or

  - The “co be
    public c

  - Public r methods
    can only

  - Externa ss + the data
    (argum



```
contract SimpleAuction {
    // Parameters of the auction. Times are either
    // absolute unix timestamps (seconds since 1970-01-01)
    // or time periods in seconds.
    address public beneficiary;
    uint public auctionEnd;

    // Current state of the auction.
    address public highestBidder;
    uint public highestBid;

    // Allowed withdrawals of previous bids
    mapping(address => uint) pendingReturns;

    // Set to true at the end, disallows any change
    bool ended;

    // Events that will be fired on changes.
    event HighestBidIncreased(address bidder, uint amount);
    event AuctionEnded(address winner, uint amount);

    // The following is a so-called natspec comment,
    // recognizable by the three slashes.
    // It will be shown when the user is asked to
    // confirm a transaction.

    /// Create a simple auction with `_biddingTime`
    /// seconds bidding time on behalf of the
    /// beneficiary address `_beneficiary`.
    constructor(
        uint _biddingTime,
        address _beneficiary
    ) public {
        beneficiary = _beneficiary;
        auctionEnd = now + _biddingTime;
    }

    /// Bid on the auction with the value sent
    /// together with this transaction.
    /// The value will only be refunded if the
    /// auction is not won.
    function bid() public payable {
```

Contract Creation

- The same piece of code ("contract") can be deployed multiple times, by different people
  - Contract "addresses" refer to a specific instance of a contract (combines contract code and a "nonce")
  - Contracts are compiled into EVM byte code and sent to the network
  - To deploy the code, you send the code (as data) to the special Ethereum address ("0")
    - (Contracts have a specialized opcode for this function…)

EVM

- Contracts are compiled into a type of Bytecode and run on a VM

To prevent cheating, the network works like Bitcoin:

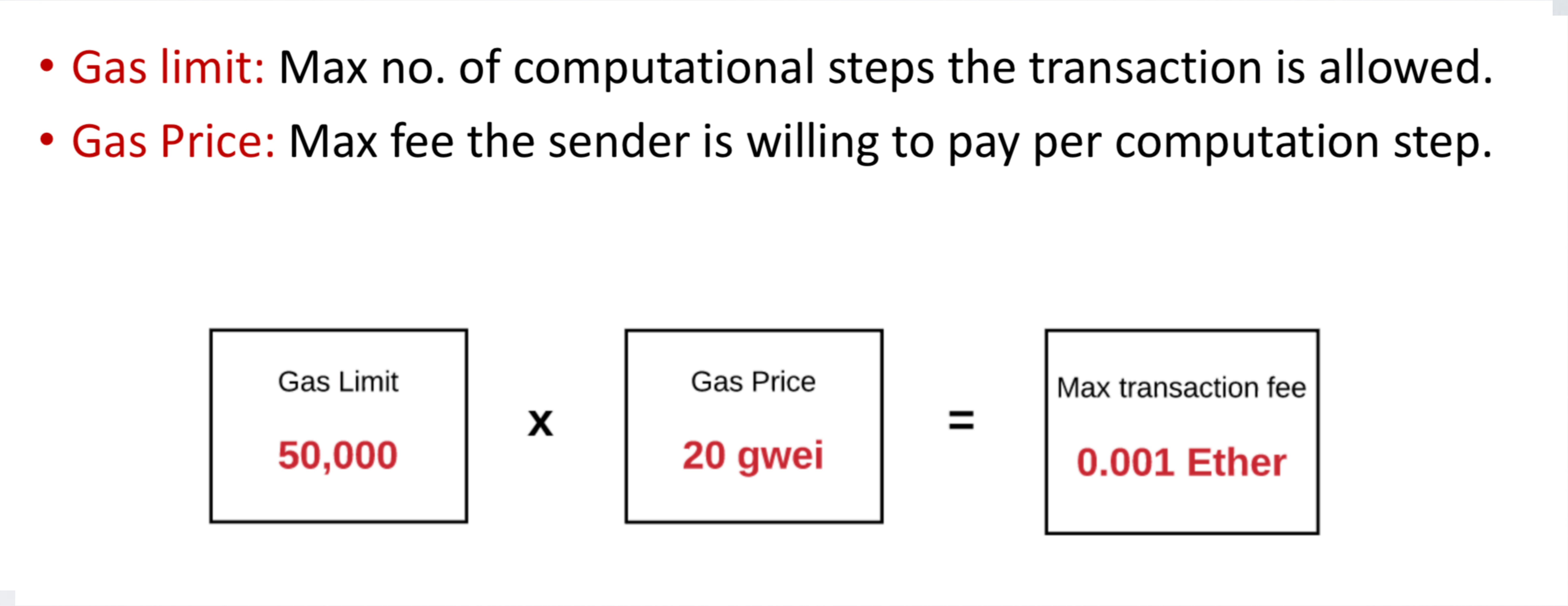
Every single node in the network must also run the EVM machine instructions and inputs for each transaction in a received block, and only accepts the block if the EVM outputs (in the block) match their local computations.

Verification through repeated computing:

Each contract execution is "replicated" across the entire Ethereum network!

- What if that node cheats?

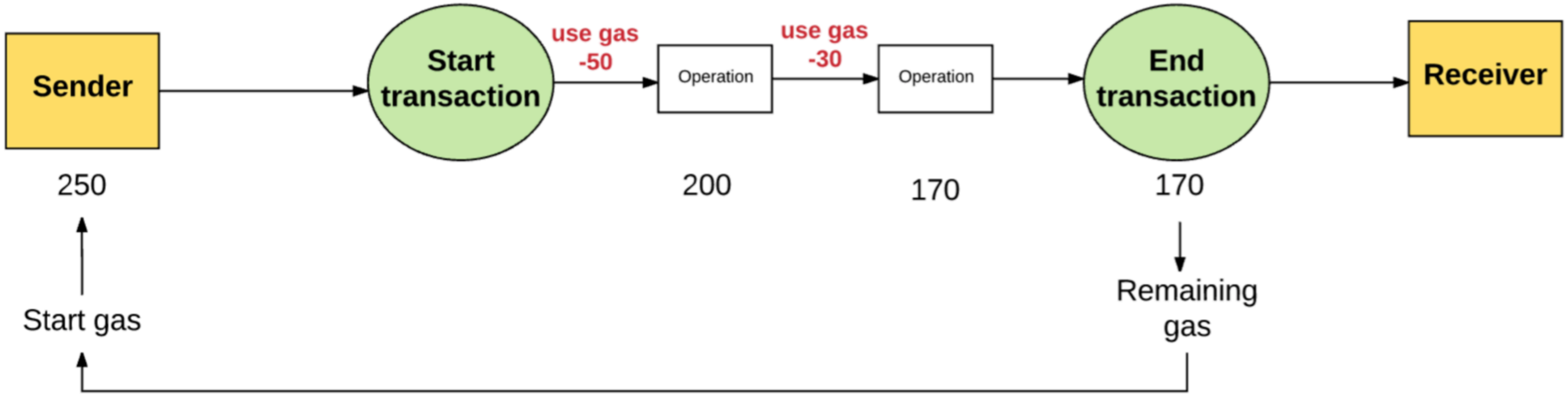
Gas limits/price



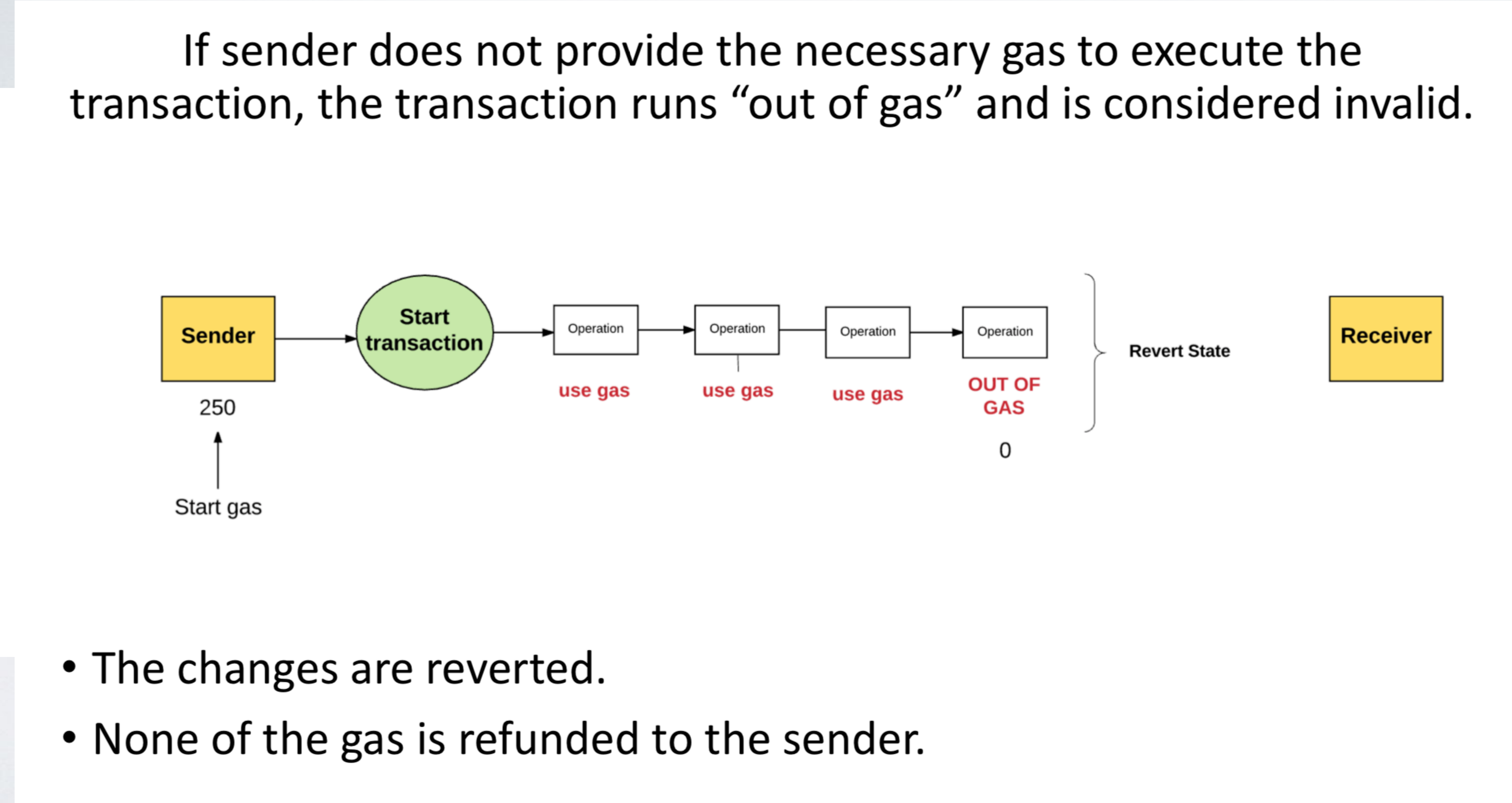
- Gas limit: Max no. of computational steps the transaction is allowed.
- Gas Price: Max fee the sender is willing to pay per computation step.

Gas limits/price

The sender is refunded for any unused gas at the end of the transaction.



Gas limits/price

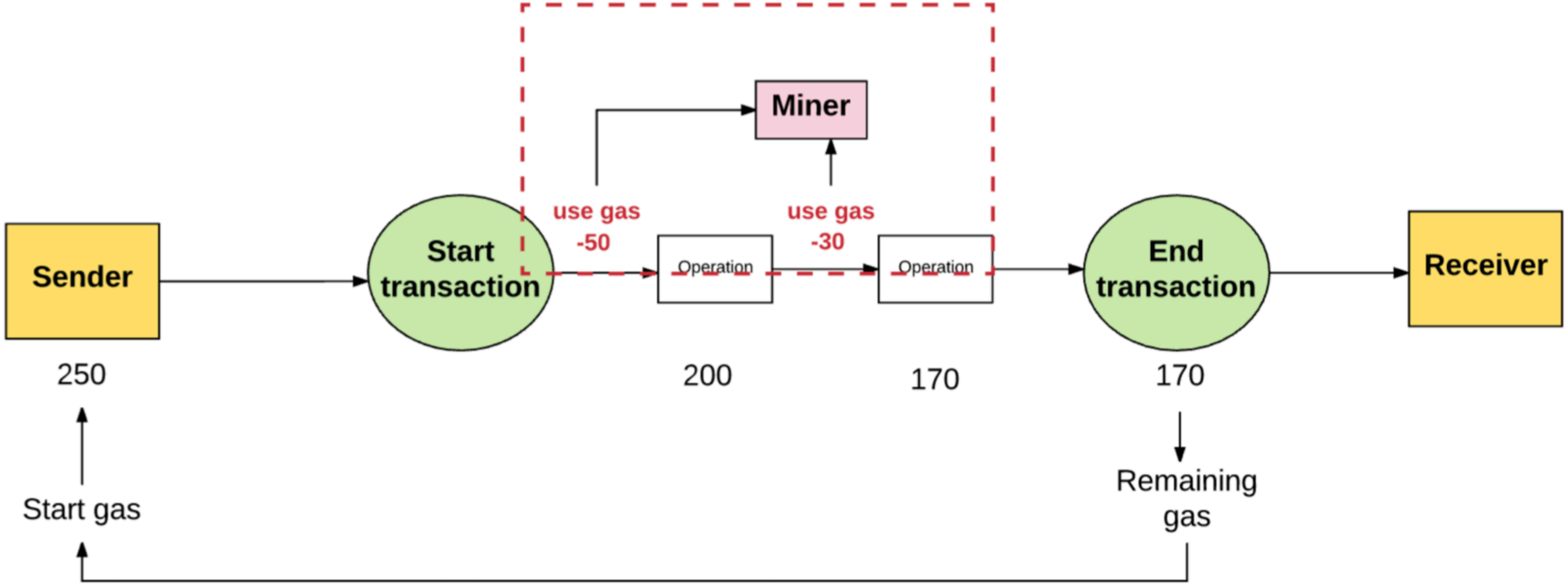


If sender does not provide the necessary gas to execute the transaction, the transaction runs “out of gas” and is considered invalid.

- The changes are reverted.
- None of the gas is refunded to the sender.

Gas limits/price

All the money spent on gas by the sender is sent to the miner's address.



Incentive problems?

Does ETH need an internal currency?

How is the chain built?

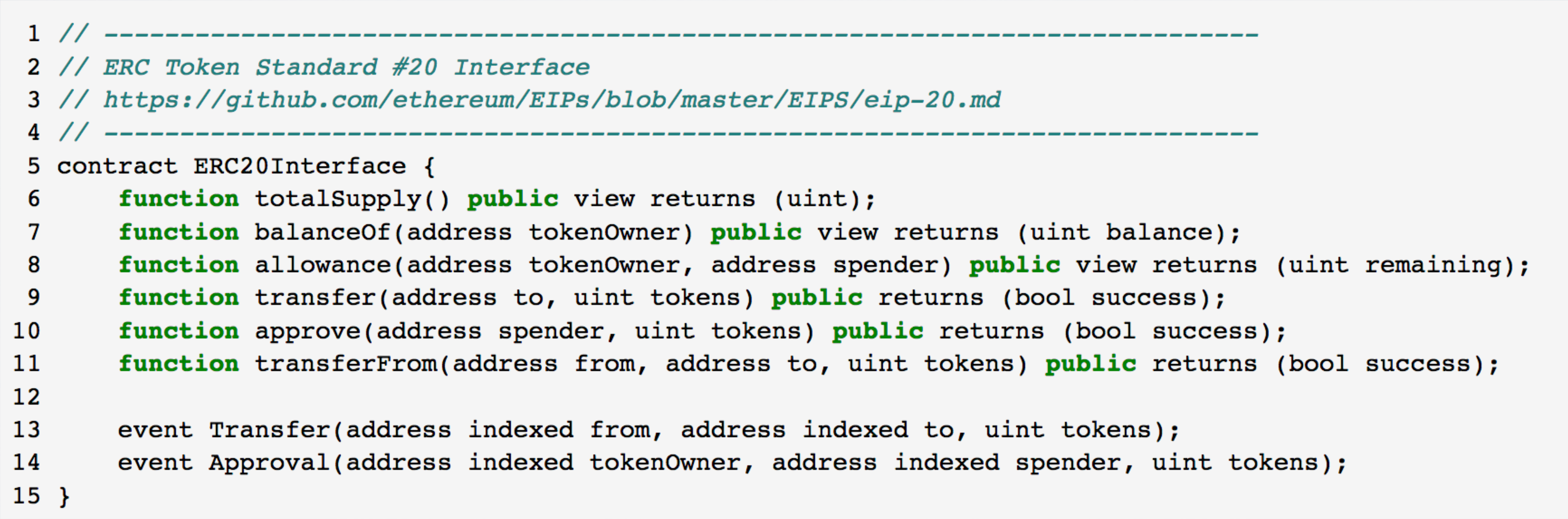
- Early versions of Ethereum worked just like Bitcoin
    - Nodes used a Proof-of-Work (based on EthHash) to mine new blocks of transactions
    - Block time was much faster (a few seconds)
    - This raised throughput but increased the rate of short-lived forks
        - To fight this, Ethereum made some changes to consensus

New consensus: proof of stake!

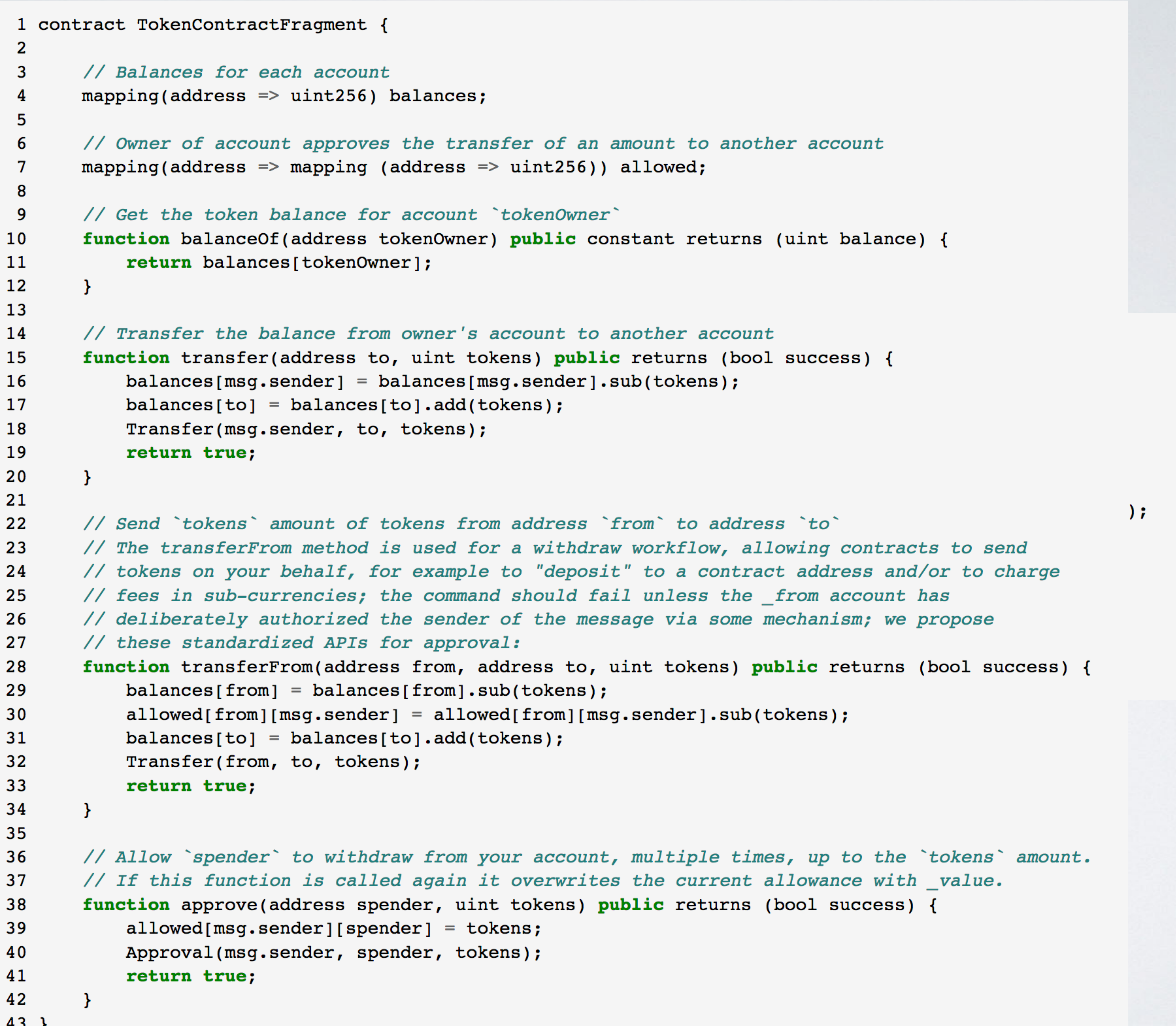
- Early versions of Ethereum worked just like Bitcoin
  - Now Ethereum uses **proof of stake**
  - Nodes must “stake” a big chunk of money (ETH)
  - This gives them a chance to propose blocks (in slots)
    - We will cover the algorithm in much greater detail later

Contract examples

- Simple “custom token” contract (ERC20)



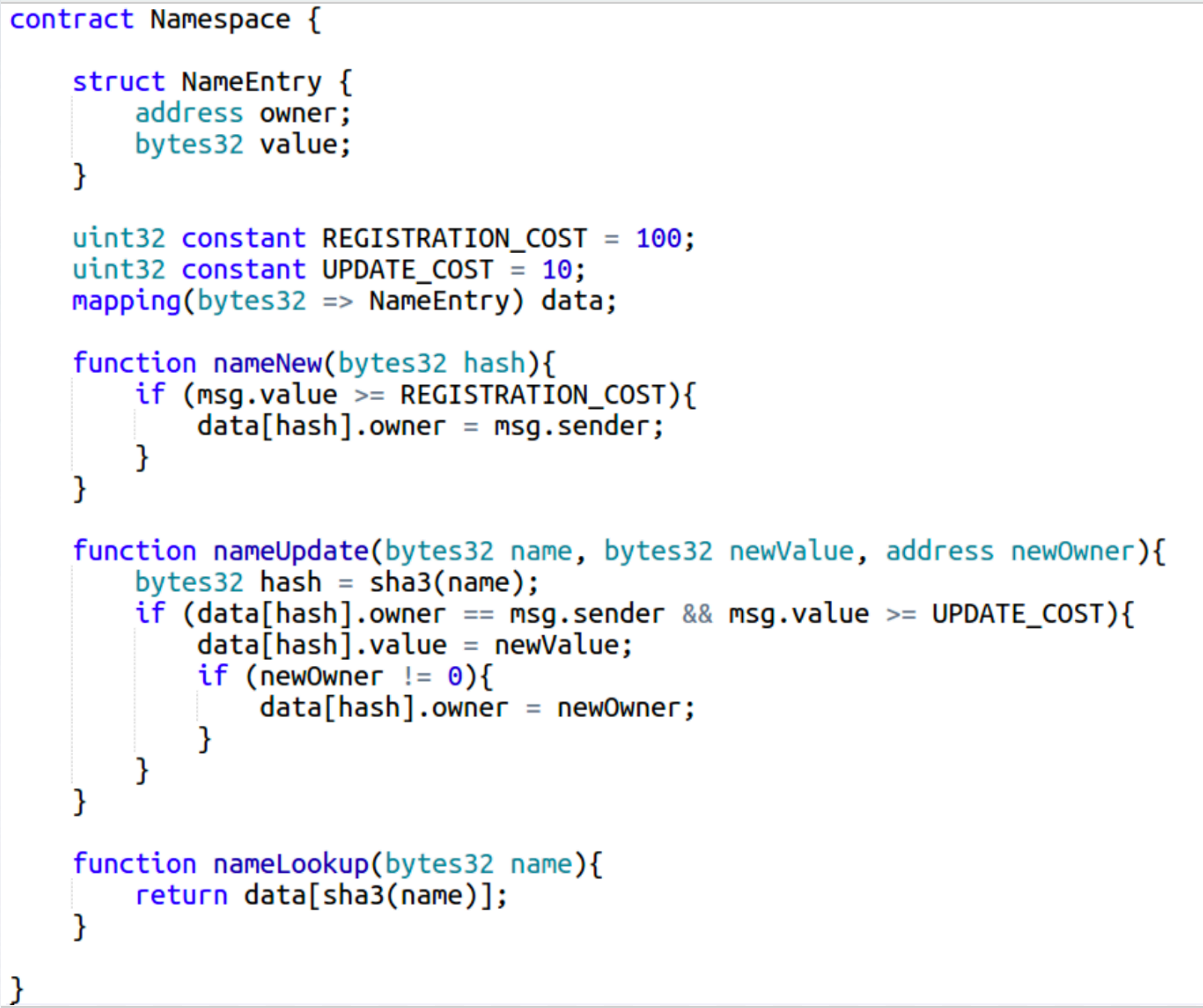
```
// ----------------------------------------------------------------------------
// ERC Token Standard #20 Interface
// https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md
// ----------------------------------------------------------------------------
contract ERC20Interface {
    function totalSupply() public view returns (uint);
    function balanceOf(address tokenOwner) public view returns (uint balance);
    function allowance(address tokenOwner, address spender) public view returns (uint remaining);
    function transfer(address to, uint tokens) public returns (bool success);
    function approve(address spender, uint tokens) public returns (bool success);
    function transferFrom(address from, address to, uint tokens) public returns (bool success);

    event Transfer(address indexed from, address indexed to, uint tokens);
    event Approval(address indexed tokenOwner, address indexed spender, uint tokens);
}
```



```
contract TokenContractFragment {

    // Balances for each account
    mapping(address => uint256) balances;

    // Owner of account approves the transfer of an amount to another account
    mapping(address => mapping (address => uint256)) allowed;

    // Get the token balance for account `tokenOwner`
    function balanceOf(address tokenOwner) public constant returns (uint balance) {
        return balances[tokenOwner];
    }

    // Transfer the balance from owner's account to another account
    function transfer(address to, uint tokens) public returns (bool success) {
        balances[msg.sender] = balances[msg.sender].sub(tokens);
        balances[to] = balances[to].add(tokens);
        Transfer(msg.sender, to, tokens);
        return true;
    }

    // Send `tokens` amount of tokens from address `from` to address `to`
    // The transferFrom method is used for a withdraw workflow, allowing contracts to send
    // tokens on your behalf, for example to "deposit" to a contract address and/or to charge
    // fees in sub-currencies; the command should fail unless the _from account has
    // deliberately authorized the sender of the message via some mechanism; we propose
    // these standardized APIs for approval:
    function transferFrom(address from, address to, uint tokens) public returns (bool success) {
        balances[from] = balances[from].sub(tokens);
        allowed[from][msg.sender] = allowed[from][msg.sender].sub(tokens);
        balances[to] = balances[to].add(tokens);
        Transfer(from, to, tokens);
        return true;
    }

    // Allow `spender` to withdraw from your account, multiple times, up to the `tokens` amount.
    // If this function is called again it overwrites the current allowance with _value.
    function approve(address spender, uint tokens) public returns (bool success) {
        allowed[msg.sender][spender] = tokens;
        Approval(msg.sender, spender, tokens);
        return true;
    }
}
```

);

"NameCoin" in Ethereum



```
contract Namespace {

    struct NameEntry {
        address owner;
        bytes32 value;
    }

    uint32 constant REGISTRATION_COST = 100;
    uint32 constant UPDATE_COST = 10;
    mapping(bytes32 => NameEntry) data;

    function nameNew(bytes32 hash){
        if (msg.value >= REGISTRATION_COST){
            data[hash].owner = msg.sender;
        }
    }

    function nameUpdate(bytes32 name, bytes32 newValue, address newOwner){
        bytes32 hash = sha3(name);
        if (data[hash].owner == msg.sender && msg.value >= UPDATE_COST){
            data[hash].value = newValue;
            if (newOwner != 0){
                data[hash].owner = newOwner;
            }
        }
    }

    function nameLookup(bytes32 name){
        return data[sha3(name)];
    }

}
```

Credit: Andrew Miller and Joe Bonneau for this code

Multisig and filters

- Can create “filter” contracts that execute another contract and/or pay out money if complex conditions are satisfied
    - E.g., If k-out-of-N signers sign (in Bitcoin this is called “multisig”)
    - Verify that a certain number of blocks have elapsed
    - Check that another contract executed

Prediction Markets

- Remember that contracts can “control” a balance (in ETH)
  - (They can control balances in e.g., ERC20 tokens as well)
  - Can make payouts of ETH conditional on certain events — e.g., signed by a notary
    - Requires: method to “place bet”
    - Method to “claim bet” (verify sig/conditions), pay to an address
    - Relies on a centralized notary! See Augur….

Decentralized Exchanges

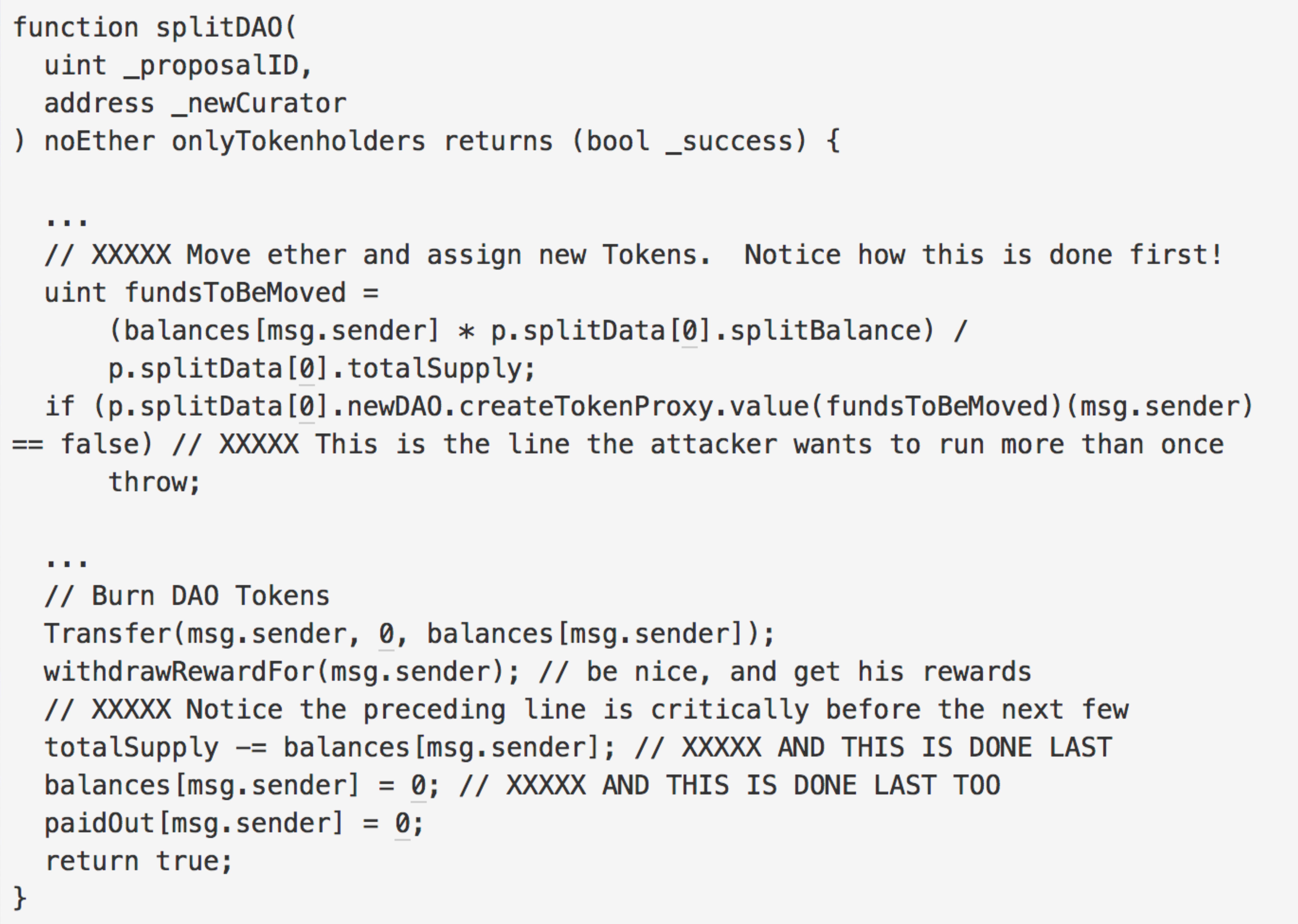
- How do we build this?

Frontrunning

DAO disaster

- Decentralized Autonomous Organization
  - "Like a VC fund" but decentralized
  - Implementation: a contract that controls money, and directs its disbursement according to "shareholder votes"
  - Shareholders buy in, pool their ETH (sending to contract)
  - Then vote on investments, which are made together
  - Users can "split" a DAO

“The DAO”



```
function splitDAO(
  uint _proposalID,
  address _newCurator
) noEther onlyTokenholders returns (bool _success) {

  ...
  // XXXXX Move ether and assign new Tokens.  Notice how this is done first!
  uint fundsToBeMoved =
      (balances[msg.sender] * p.splitData[0].splitBalance) /
      p.splitData[0].totalSupply;
  if (p.splitData[0].newDAO.createTokenProxy.value(fundsToBeMoved)(msg.sender)
== false) // XXXXX This is the line the attacker wants to run more than once
      throw;

  ...
  // Burn DAO Tokens
  Transfer(msg.sender, 0, balances[msg.sender]);
  withdrawRewardFor(msg.sender); // be nice, and get his rewards
  // XXXXX Notice the preceding line is critically before the next few
  totalSupply -= balances[msg.sender]; // XXXXX AND THIS IS DONE LAST
  balances[msg.sender] = 0; // XXXXX AND THIS IS DONE LAST TOO
  paidOut[msg.sender] = 0;
  return true;
}
```

How to upgrade a contract?

How to upgrade a contract?

- Contracts are default immutable
- If there is no useful ongoing state, don’t: just replace it
- If there is ongoing state (e.g., account balances) then:
  - Don’t allow upgrades — and pray you got the code right
  - Call upgradeable/replaceable library code
  - Create a complex mechanism to transfer state from an old contract instance to a new contract instance

Future of Ethereum

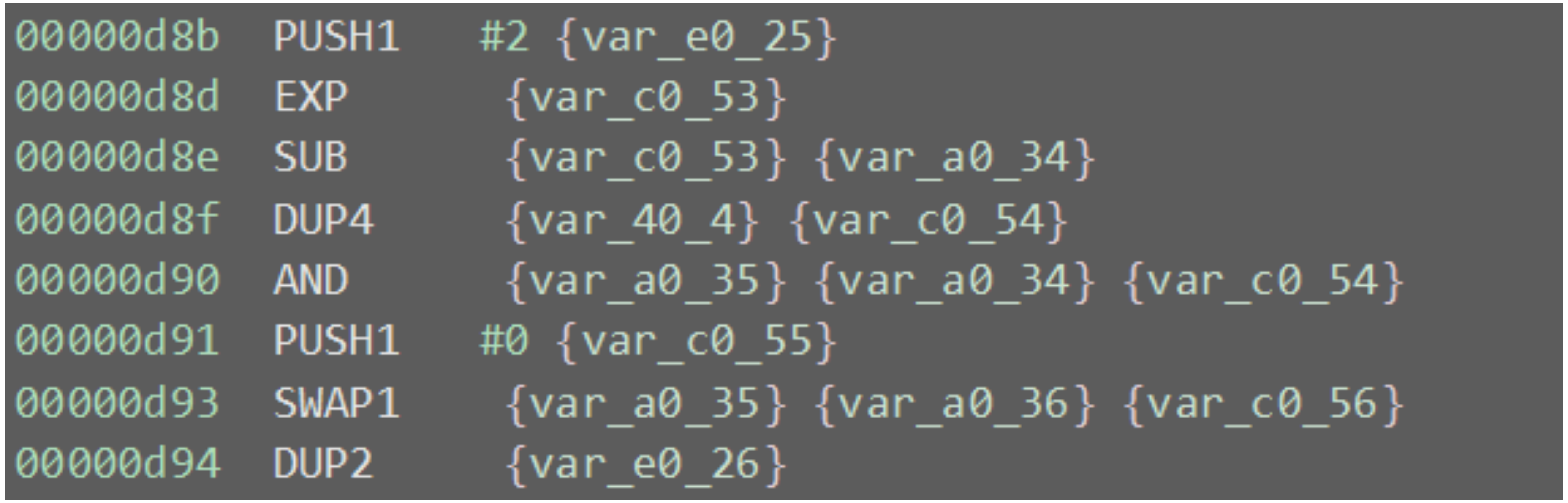
- Proof of stake
- Rollups
- Sharding

From concept to practice

- **How does a developer see Ethereum?**
  - So far we have talked about:
    - Init function (at deploy, creation)
      - (A note: contracts can ‘spawn’ new contracts!)
    - A single stateUpdate function (triggered by message Tx)
    - Databases and VMs

From concept to practice

- Ethereum programs are in "EVM byte code"
  - This is great for running things in a VM, works across platforms
  - Not made for human comprehension



```
00000d8b  PUSH1   #2 {var_e0_25}
00000d8d  EXP      {var_c0_53}
00000d8e  SUB      {var_c0_53} {var_a0_34}
00000d8f  DUP4     {var_40_4} {var_c0_54}
00000d90  AND      {var_a0_35} {var_a0_34} {var_c0_54}
00000d91  PUSH1   #0 {var_c0_55}
00000d93  SWAP1    {var_a0_35} {var_a0_36} {var_c0_56}
00000d94  DUP2     {var_e0_26}
```

Source: https://blog.ret2.io/2018/05/16/practical-eth-decompilation/

From concept to practice

- **How does a developer see Ethereum?**
  - So far we have talked about:
    - Init function (at deploy)
    - A single stateUpdate function (triggered by message Tx)
    - Databases and VMs

From concept to practice

- **How does a developer see Ethereum?**
  - So far we have talked about:
    - Init function (at deploy)
    - A single stateUpdate function (triggered by message Tx)
    - Databases and VMs
  - But this sucks for software developers
    - Let’s instead think of contracts as object-oriented programs

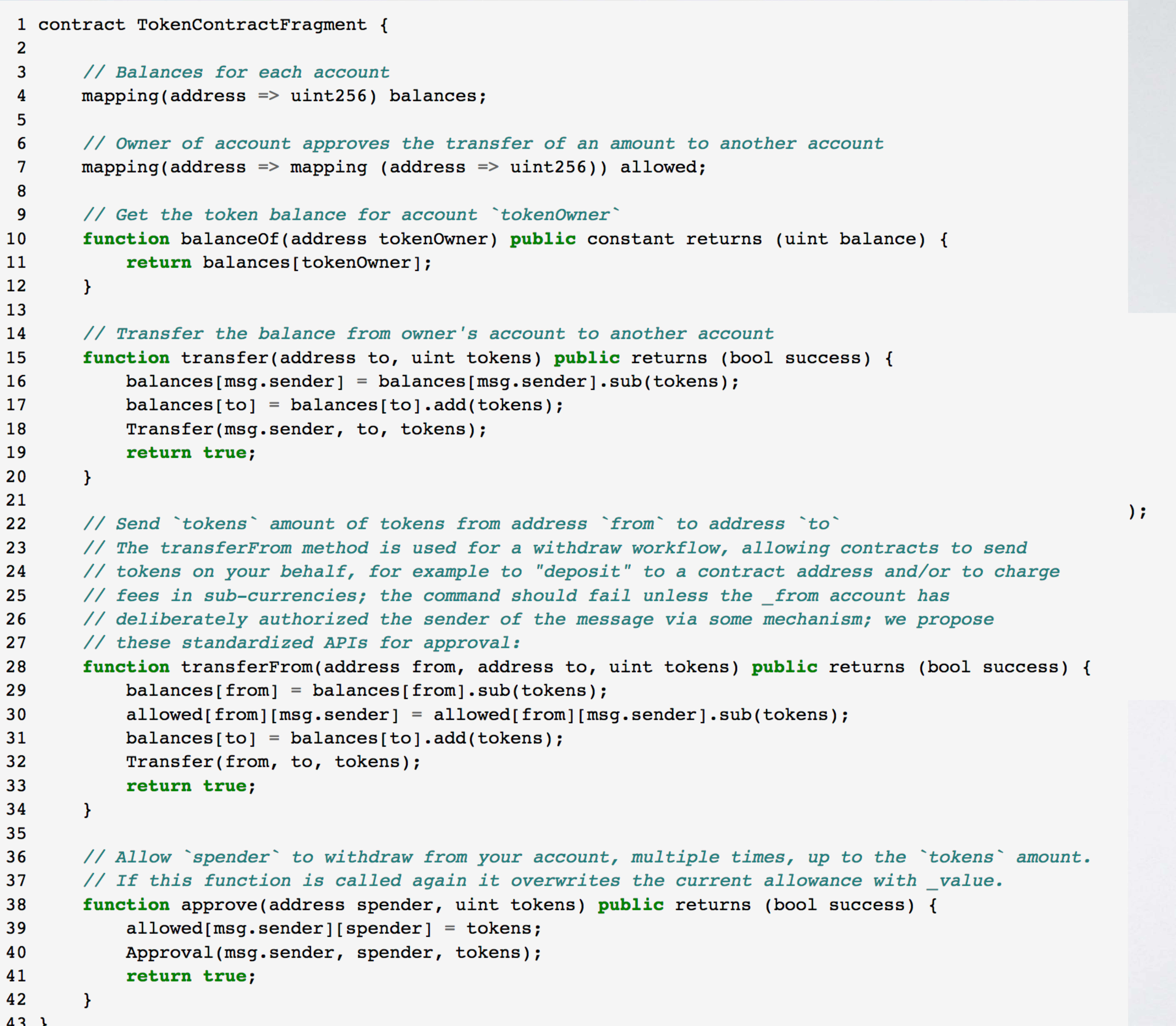
From concept to practice

- Developers typically write programs in a high-level language
  - Technically any language can compile to EVM bytecode
  - And there are a few: Agoric (Javascript), Vyper
    - Some other chains (e.g., Solana) use rust
  - However, most Ethereum smart contracts are written in **Solidity**

Source: https://blog.ret2.io/2018/05/16/practical-eth-decompilation/

Solidity

- **How does a Solidity developer see Ethereum?**
  - Solidity is object-oriented
    - "contract" programs are like classes, with methods and variables
    - Each contract will have a constructor method that initializes any state variables
    - There are "view" (read-only) methods, and methods that (may) change state
    - Methods can have modifiers attached, that execute specific checks



```
contract TokenContractFragment {

    // Balances for each account
    mapping(address => uint256) balances;

    // Owner of account approves the transfer of an amount to another account
    mapping(address => mapping (address => uint256)) allowed;

    // Get the token balance for account `tokenOwner`
    function balanceOf(address tokenOwner) public constant returns (uint balance) {
        return balances[tokenOwner];
    }

    // Transfer the balance from owner's account to another account
    function transfer(address to, uint tokens) public returns (bool success) {
        balances[msg.sender] = balances[msg.sender].sub(tokens);
        balances[to] = balances[to].add(tokens);
        Transfer(msg.sender, to, tokens);
        return true;
    }

    // Send `tokens` amount of tokens from address `from` to address `to`
    // The transferFrom method is used for a withdraw workflow, allowing contracts to send
    // tokens on your behalf, for example to "deposit" to a contract address and/or to charge
    // fees in sub-currencies; the command should fail unless the _from account has
    // deliberately authorized the sender of the message via some mechanism; we propose
    // these standardized APIs for approval:
    function transferFrom(address from, address to, uint tokens) public returns (bool success) {
        balances[from] = balances[from].sub(tokens);
        allowed[from][msg.sender] = allowed[from][msg.sender].sub(tokens);
        balances[to] = balances[to].add(tokens);
        Transfer(from, to, tokens);
        return true;
    }

    // Allow `spender` to withdraw from your account, multiple times, up to the `tokens` amount.
    // If this function is called again it overwrites the current allowance with _value.
    function approve(address spender, uint tokens) public returns (bool success) {
        allowed[msg.sender][spender] = tokens;
        Approval(msg.sender, spender, tokens);
        return true;
    }
}
```

```
);
```

Concurrency and re-entrancy

- Ethereum transactions run sequentially and atomically
  - In principle this is good: there **appear** to be no concurrency issues (no threads) and your methods always run to completion or don’t complete at all!

Example (ok)

```
function transfer(uint amount, address recipient) … {
   if (balances[msg.sender] < amount) {
      // insufficient balance
      revert(‘Something bad happened’);
   }

   // Transfer the money
   balances[msg.sender] -= amount;
   balances[recipient] += amount;
}
```

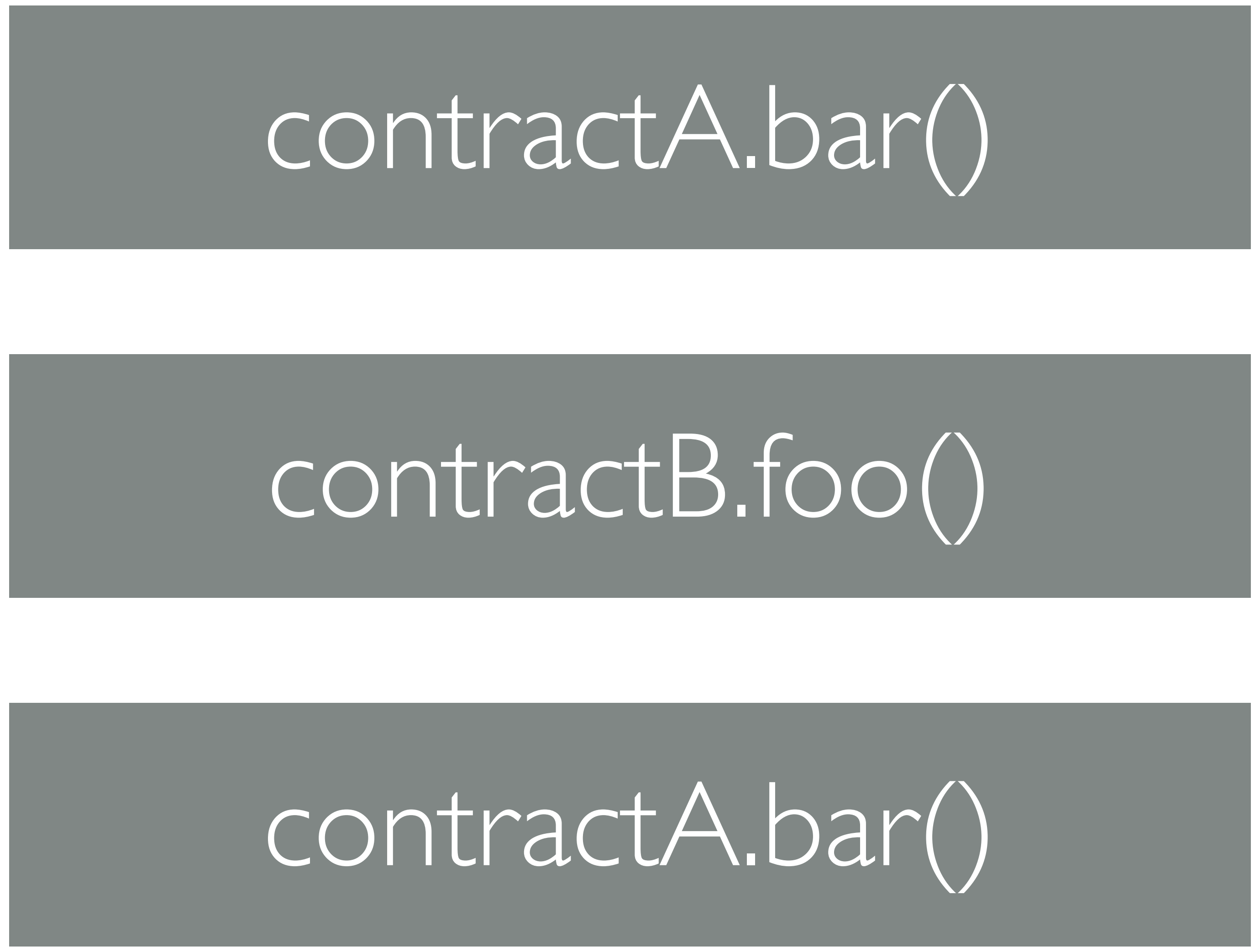
Re-entrancy

```
function transferAndNotify(uint amount, address notifyContract) … {
   if (balances[msg.sender] < amount) {
      // insufficient balance
      revert(‘Something bad happened’);
   }

   // Call the specified contract to notify it that a deposit is coming
   notifyContract.notify(amount, “You are getting a deposit!”);

   // Transfer the money
   balances[msg.sender] -= amount;
   balances[notifyContract] += amount;
}
```

Re-entrancy

- There is still one scary “gotcha”!

- Ethereum is not re-entrancy “safe”. *You can still have multiple calls to the same routine within any given call-stack.*



contractA.bar()

contractB.foo()

contractA.bar()

Re-entrancy

```
function transferAndNotify(uint amount, address notifyContract) … {
   if (balances[msg.sender] < amount) {
      // insufficient balance
      revert(‘Something bad happened’);
   }
   balances[notifyContract] += amount;

   // Call the specified contract to notify it that a deposit is coming
   notifyContract.notify(amount, “You are getting a deposit!”);

   // Transfer the money
   balances[msg.sender] -= amount;
}
```

Re-entrancy

```
function transferAndNotify(uint amount, address notifyContract) … {
  if (balances[msg.sender] < amount) {
    // insufficient balance
    revert('Something bad happened');
  }
  balances[notifyContract] += amount;

  // Call the specified contract to notify it that a deposit is coming
  notifyContract.notify(amount, "You are getting a deposit!");

  // Transfer the money
  balances[msg.sender] -= amount;
}
```

What if this contract call calls us?

Re-entrancy

```
function transferAndNotify(uint amount, address notifyContract) … {
   if (balances[msg.sender] < amount) {
      // insufficient balance
      revert(‘Something bad happened’);
   }
   balances[notifyContract] += amount;

   // Call the specified contract to notify it that a deposit is coming
   notifyContract.notify(amount, “You are getting a deposit!”);

   // Transfer the money
   balances[msg.sender] -= amount;
}
```

What if this contract call calls us?

Re-entrancy

```
function transferAndNotify(uint amount, address notifyContract) … {
    if (balances[msg.sender] < amount) {
        // insufficient balance
        revert(‘Something bad happened’);
    }
    balances[notifyContract] += amount;

    // Call the specified contract to notify it that a deposit is coming
    notifyContract.notify(amount,“You are getting a deposit!”);

    // Transfer the money
    balances[msg.sender] -= amount;
}
```

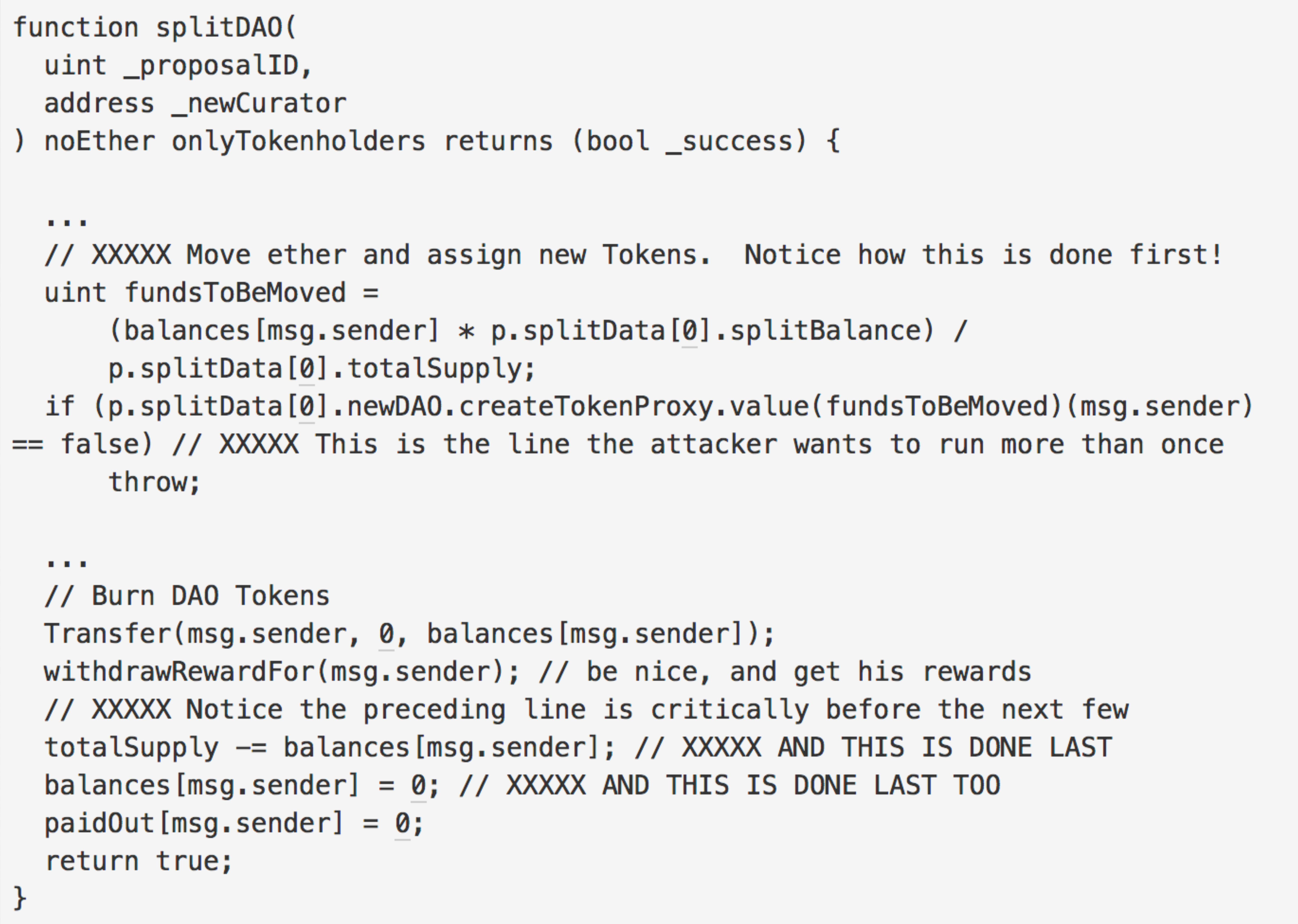
What if this contract call calls us?

Re-entrancy

```
function transferAndNotify(uint amount, address notifyContract) … {
  if (balances[msg.sender] < amount) {
    // insufficient balance
    revert(‘Something bad happened’);
  }
  balances[notifyContract] += amount;

  // Call the specified contract to notify it that a deposit is coming
  notifyContract.notify(amount, “You are getting a deposit!”);

  // Transfer the money
  balances[msg.sender] -= amount;
}
```

What if this contract call calls us?

“The DAO”



```
function splitDAO(
  uint _proposalID,
  address _newCurator
) noEther onlyTokenholders returns (bool _success) {

  ...
  // XXXXX Move ether and assign new Tokens.  Notice how this is done first!
  uint fundsToBeMoved =
      (balances[msg.sender] * p.splitData[0].splitBalance) /
      p.splitData[0].totalSupply;
  if (p.splitData[0].newDAO.createTokenProxy.value(fundsToBeMoved)(msg.sender)
== false) // XXXXX This is the line the attacker wants to run more than once
      throw;

  ...
  // Burn DAO Tokens
  Transfer(msg.sender, 0, balances[msg.sender]);
  withdrawRewardFor(msg.sender); // be nice, and get his rewards
  // XXXXX Notice the preceding line is critically before the next few
  totalSupply -= balances[msg.sender]; // XXXXX AND THIS IS DONE LAST
  balances[msg.sender] = 0; // XXXXX AND THIS IS DONE LAST TOO
  paidOut[msg.sender] = 0;
  return true;
}
```

Solutions?

```
function transferAndNotify(uint amount, address notifyContract) … {
  if (globalLock == true) {
    revert(“Locked.”);
  }
  globalLock = true;

  if (balances[msg.sender] < amount) {
    // insufficient balance
    revert(‘Something bad happened’);
  }

  // Call the specified contract to notify it that a deposit is coming
  notifyContract.notify(amount, “You are getting a deposit!”);

  // Transfer the money
  balances[msg.sender] -= amount;
  balances[notifyContract] += amount;

  globalLock=false;
}
```

```
function transferAndNotify(uint amount, address notifyContract) … {
  if (globalLock == true) {
    revert(“Locked.”);
  }
  globalLock =

  if (balances[msg.s
    // insufficient b
    revert(‘Someth
  }

  // Call the specifi
  notifyContract.nc

  // Transfer the m
  balances[msg.sen
  balances[notifyContract] += amount;

  globalLock=false;
}
```

Drawbacks of (global) locks:

1. Extra gas (due to stores/loads)

2. Can get “stuck” if you’re careless

3. Sometimes re-entrant calls are useful!

Check-Effects-Interaction pattern

- Most common solution is to follow a code pattern:
  - First perform all contract checks (CHECKS)
  - Second, update contract state (EFFECTS)
  - Finally, make any contract calls (INTERACTION)

Check-Effects-Interaction pattern

```
function transferAndNotify(uint amount, address notifyContract) … {
   // CHECK
   require (amount < balances[msg.sender]);

   // EFFECTS
   balances[msg.sender] -= amount;
   balances[notifyContract] += amount;

   // INTERACTION
   notifyContract.notify(amount,“You are getting a deposit!”);
}
```

Contract upgrades

- Ethereum contracts are not (natively) upgradeable
  - Once a contract is deployed, it can self-destruct
  - But its code cannot be changed
  - But some contracts need to be upgraded (bug fixes, etc.)
    - How are we going to handle this?