P2P Systems and Blockchains
Spring 2019,
instructor: Laura Ricci
laura.ricci@unipi.it
Lesson 17:
ETHEREUM:
SMART CONTRACTS
13/05/2019
A **smart contract** is a computerized transaction protocol that executes the terms of a contract. The general objectives are to satisfy common contractual conditions (such as payment terms, liens, confidentiality, and even enforcement), minimize exceptions both malicious and accidental, and minimize the need for trusted intermediaries. Related economic goals include lowering fraud loss, arbitrations and enforcement costs, and other transaction costs.

- **Nick Szabo** “The Idea of Smart Contracts” [1994]
SMART CONTRACTS

- contract
  - formalizes a relationship and contains a set of promises made between principals.

- smart contract
  - based on the translation of contractual clauses into code
  - more functional compared to paper-based
    - can reduce costs
  - aim to remove the need for trusted intermediaries
    - make it more difficult for malicious parties to undermine compliance with the contract terms
  - uses cryptography and other security mechanisms
    - secure algorithmically specifiable relationships from being breached and ensure the agreed upon terms are satisfied.
Smart contract
- some code automating the “if this happens then do that” part of traditional contracts.

What is better with respect to normal contract?
- Computer code behaves in expected ways and doesn’t have the linguistic nuances of human languages.
SMART CONTRACTS: AN EXAMPLE

- a bank account may behave like a smart contract
  - my bank account has a balance
  - every month, I commission an automated payment that deducts a fixed amount and sends it to my landlady
  - if there is not enough money in my bank account, the payment fails
  - I get fined, and another procedure is triggered.

- these are the instructions I have set up which are associated with the account
  - a contract between me and the bank.

- my bank is the ultimate guardian of my bank account and has complete control:
  - can arbitrarily add money to my account (unlikely) or subtract it: it may happen!
  - even if coded, it sits on one computer and is executed by one party (the bank) which must be trusted
SMART CONTRACTS IN BITCOIN

• use scripting language of Bitcoin to write smart contracts?

• the scripting language has several limitations:
  • lack of Turing-completeness (e.g., no loops)
  • Bitcoin purposefully picked a non Turing-complete language
  • lack of arbitrary state variables:
    • contracts on Bitcoins cannot hold state
    • a variable may remember preceding events and user interactions
    • it can be modified by the contract.

• hard to build stateful general contracts on Bitcoin.

• Ethereum is based on the possibility of defining real smart contracts.
ETHEREUM VERSUS BITCOIN

Bitcoin

Ethereum
WHAT IS ETHEREUM?

- main idea: put “real” smart contracts on the blockchain
  - consensus mechanisms assure that all the nodes execute the contract
  - each node changes its own version of the ledger with the result of the smart contract evaluation
  - anything that doesn’t conform to pre-agreed rules is rejected

- Ethereum: decentralized smart contracts
  - smart contracts that are
    - replicated
    - processed on all the computers on the network, without a central coordinator
  - like a distributed computer to execute code

- a distributed state machine:
  - global state: status of all the smart contracts
  - transactions change global state
Transparency

• all participants in a blockchain run the same code, each verifying the other
• the logic of the smart contract is visible to all
• privacy may be an issue
  • solutions based on zero-knowledge proofs may be used in some cases

Flexibility

• smart contract are written in a “Turing complete” language
• can do anything that a normal computer can do
• but you need to pay for all nodes on the network to run the code in parallel.
  • the blockchain version will be more expensive to run than on a regular computer
Thanks to this guy!

Vitalik Buterin (January 31th 1994)
ETHEREUM: THE EVOLUTION

- Idea: Oct/2013
- Crowdfunding: Jul/2014
- Ethereum Foundation: Sep/2014
- First release: Jun/2015
- Official release: Mar/2016

January 2014
Vitalik Buterin announces Ethereum

July 2014
Ethereum Presale (18 Mio $)

March 2016
Second Release: Homestead

April 2016
Launch of the DAO

May 2016
DAO collects 160 Mio $

June 2016
DAO gets "hacked"

July 2016
First Hard Fork
ETHEREUM VS. BITCOIN

- Ethereum extends the blockchain concepts from Bitcoin
  - runs computer code equivalently on many computers around the world.
  - what Bitcoin does for distributed data storage, Ethereum does for distributed data storage + computations

- similarities
  - blockchain-based:
    - but in Ethereum blocks contain data and smart contracts
  - public and permissionless blockchain
  - has a Proof of Work mining (Proof of Stake proposed for Casper release)
  - has a native cryptocurrency: ether (ETH)

- differences
  - account-based
  - Turing complete smart contracts
  - uncles +Ethash
  - gas
Ether is the internal currency of Ethereum and is used:
- to transfer values in transactions
- to pay gas: computation fees

<table>
<thead>
<tr>
<th>Value (in wei)</th>
<th>Exponent</th>
<th>Common name</th>
<th>SI name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>wei</td>
<td>Wei</td>
</tr>
<tr>
<td>1,000</td>
<td>$10^3$</td>
<td>Babbage</td>
<td>Kilowei or femtoether</td>
</tr>
<tr>
<td>1,000,000</td>
<td>$10^6$</td>
<td>Lovelace</td>
<td>Megawei or picoether</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>$10^9$</td>
<td>Shannon</td>
<td>Gigawei or nanoether</td>
</tr>
<tr>
<td>1,000,000,000,000</td>
<td>$10^{12}$</td>
<td>Szabo</td>
<td>Microether or micro</td>
</tr>
<tr>
<td>1,000,000,000,000,000</td>
<td>$10^{15}$</td>
<td>Finney</td>
<td>Milliether or milli</td>
</tr>
<tr>
<td>$1,000,000,000,000,000,000$</td>
<td>$10^{18}$</td>
<td><strong>Ether</strong></td>
<td><strong>Ether</strong></td>
</tr>
<tr>
<td>$1,000,000,000,000,000,000,000,000$</td>
<td>$10^{21}$</td>
<td>Grand</td>
<td>Kiloether</td>
</tr>
<tr>
<td>$1,000,000,000,000,000,000,000,000,000,000$</td>
<td>$10^{24}$</td>
<td></td>
<td>Megaether</td>
</tr>
</tbody>
</table>

each name “a piece of computer science”
UTXO VS. ACCOUNTS

- Bitcoin user’s available balance
  - sum of unspent transaction outputs for which he/she owns the corresponding private keys
    - UTXO unspent transaction outputs: Bitcoin implicit state
    - Bitcoin scripts do not explicitly modify the state in UTXO
      - only limited modification performed through transactions
- Ethereum uses a different concept
  - account: like a bank account, keeps track of balance

![Diagram](https://example.com/diagram.png)

**Bitcoin's UXT0**

- 5 BTC ⇒ Bob
- 3 BTC ⇒ Bob
- 2 BTC ⇒ Bob

**Ethereum accounts**

- address: “0xfa38b...”
- balance: 10 ETH
- code: \( c := a + b \)
ETHEREUM ACCOUNTS

- the global shared state of Ethereum is comprised of
  - many accounts
  - small objects interacting with one another through a message-passing

- an account has state and 20 byte/160 bit byte identifier, like:
  0x91fff4cbd6159a527ca4dcce2e3937431086c662

- two types of accounts:
  - externally owned
    - controlled private keys and have no code associated with them.
  - contract accounts
    - have code associated with them and are controlled by this code.
EXTERNALLY OWNED ACCOUNTS

- owned by some external entity
- controlled by private keys (users)
  - having the private key allows access to funds and to contracts invocation
- can send transactions to transfer ether or to trigger a smart contract
- contain
  - address
  - ether balance
(SMART) CONTRACT ACCOUNTS

- the contract is the “owner” of the account: controlled by the logic of the contract code
  - recorded on the Ethereum blockchain at the contract account’s creation
  - executed by the Ethereum Virtual Machine (EVM)
- contains
  - address
  - contract code
  - persistent storage to store contract related variables
  - ether balance
  - does not have a private key
(SMART) CONTRACT ACCOUNTS

- contracts may be triggered by **transactions** and **messages**
  - when a transaction destination is a contract address, it causes that contract to run in the EVM
- transactions may call functions inside a contract and may contain
  - ether to transfer to the contract
  - data, specifying parameters to pass to the function of the contract
- contracts can also send and receive ether, like EOAs
(SMART) CONTRACT ACCOUNTS

- Contract accounts do not have a private key and cannot initiate a transaction.
- Contracts:
  - Reacts to transactions and to messages.
  - May call other contracts, but not themselves, building complex execution paths.
- Can generate messages as a reply to transactions they have received from an EOA or from another contract account.
- In general, accounts interact with the network, other accounts, other contracts, and contract state through external or internal transactions.
SMART CONTRACT: A SIMPLE EXAMPLE

```
pragma solidity ^0.4.4;

contract Example1 {
    uint public variable;
    address public owner;

    function Example1 () {
        variable = 30;
        owner = msg.sender;
    }

    function double_it (uint value) returns (uint){
        var temp = value * 2;
        return temp;
    }
}
```

This function is called constructor. It is run only once when the contract is sent to the blockchain. It is not considered as a contract function and cannot be called, after the contract is created.

We can call `double_it()` from outside as:
```
instance.double_it.call(5, {from:"0x11"});
```

"instance" is just an instance of the contract.

This function doubles the value it receives and returns the result.

msg.sender here is the address of the account that sends the contract to the blockchain.

In this example, the contract function does not change the contract state, i.e. the value of "variable" is not changed by function: `double_it()`.
SMART CONTRACT LIKE A JUKEBOX

• can be activated and run, by funding it with some ETH

• to run create a transaction sending a payment of ETH to the contract, and possibly supplying some other input information

• each miner runs the smart contract

• if no one is behaving badly, each computer on the Ethereum network will produce the same output

• the winning miner will publish the block to the rest of the network,
  • the other computers validate that they get the same result
  • if so, they add the block to their own blockchains.
  • the state of Ethereum’s blockchain so gets updated.
any action occurring on the Ethereum blockchain is always set in motion by transactions fired from externally controlled accounts.

A transaction refers to a signed data package storing a message

- sent from an EOA to another account
- may trigger subsequent messages from contracts to other contracts
- serialized and sent on the network
- generates a change in the state of the blockchain if included in a block
- used also to generate new contracts

a bridge between the external world to the internal state of Ethereum.
EOA TRANSACTIONS

- a transaction between two EOAs
  - is simply a value transfer (transaction)

- a transaction from an EAO to a contract account activates the smart contract account’s code
  - allows a contract account to perform actions
    - write to internal storage
    - perform some calculation
    - send message to another smart contract, through an internal transaction
    - create new contracts
MESSAGES OR INTERNAL TRANSACTIONS

- can be sent from a contract to another contract
  - essentially function calls
- like a transaction, but produced by a contract and not a EOA
- like transactions, triggers the execution of the associated code is executed.
## External and Contract Accounts

<table>
<thead>
<tr>
<th></th>
<th>Externally owned account</th>
<th>Contract account</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identified by an address</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Holds the account’s Ether balance</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Hold contract code</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>Holds the account’s storage</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Associated private-public key</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>Can send signed transactions</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>Can create contracts</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Can send unsigned transactions</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>Holds a nonce</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

- Note that contract can send messages and create another contract(s)!
- A signed transaction is sent only by externally owned account and sending it will cost the sender.
THE HALTING PROBLEM FOR SMART CONTRACTS

• Turing completeness
  • implies that some programs take forever to run.

• Halting problem: it is not possible to tell, just by looking at the program, whether it will take forever or not to execute.
  • actually, the program must be executed to verify this
  • this is a huge problem for Ethereum if not properly addressed.

• what happens if malicious users or accidents ask Ethereum to execute a never stopping code?

```javascript
function foo()
{
    while (true) {
        /* Loop forever! */
    }
}
```

• EVM is a single-threaded machine, without any scheduler
  • if it becomes stuck in an infinite loop, all the world nodes will get stuck executing the loop forever!
THE HALTING PROBLEM FOR SMART CONTRACTS

- if after a maximum amount of computation has been performed, the execution has not ended
  - the EVM halts the execution of the program

- EVM: a quasi-Turing-complete machine
  - can run any program, but only if the program terminates within a particular amount of computation.
  - at any one time, there is a defined computational limit
    - not pre-fixed in Ethereum.

- Ethereum introduces the concept of gas: a unit for measuring both
  - the computational and
  - the storage resources required to perform actions on the Ethereum blockchain

- executing a smart contract is not free, incurs a fee!
  - more complex system with respect to Bitcoin
GAS AND GAS PRICE

• gas is the unit used to measure the fees for a particular computation
  • adding two numbers : 3 gas
  • computing a Keccak-256 hash : 30 gas + 6 gas for each 256 bits of data being hashed
  • sending a transaction : 21,000 gas

• gas price is the amount of Ether you are willing to spend on every unit of gas
  • measured in “gwei” 1 gwei = 1,000,000,000 wei

• purchasing gas like purchasing distributed, trustless computational power.
ETHEREUM GAS AND FEES

- with every transaction, a sender sets a **gas limit** and a **gas price**

  - **gas price** \( \times \) **gas limit** = max amount of wei sender is willing to pay for transaction

  \[
  \text{Gas Limit} \quad \times \quad \text{Gas Price} \quad = \quad \text{Max transaction fee}
  \]

  \[
  \begin{array}{c}
  \text{Gas Limit} \\
  50,000 \\
  \text{Gas Price} \\
  20 \text{ gwei} \\
  \text{Max transaction fee} \\
  0.001 \text{ Ether}
  \end{array}
  \]

- transaction is executed
  - the gas limit is enough to execute the transaction, transaction will execute
  - if they have enough Ether in their account balance to cover the gas limit

- the fees are rewards for miners for the effort to run computations and validate transactions

- the higher the gas price the sender is willing to pay, the more likely miners will be to select its transaction.
GAS AND PAYMENTS

- the users must have enough Ether in their account balance to cover this maximum,
- the sender is refunded for any unused gas at the end of the transaction, exchanged at the original rate.
OUT OF GAS TRANSACTIONS

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

- the transaction processing aborts and any state changes that occurred are reversed.

- since the machine has spent the effort to run the calculations before running out of gas, none of the gas is refunded to the sender.
ETHEREUM FEES AND GAS: RECAP

At the start of the transaction:

- \( \text{gaslimit} \times \text{gasprice} \) ether are subtracted from the sender’s account.
- if the contract successfully executes
  - the remaining gas is refunded to the sender.
- if the contract execution runs out of gas before it finishes
  - then execution reverts
  - account is not refunded.
ETHEREUM TRANSACTIONS

- A signed data package that stores a message sent from an externally owned account.

- The general structure of a transaction:

- The transaction is serialized using the Recursive Length Prefix (RLP) encoding scheme.
ETHEREUM TRANSACTIONS: NONCE

• from the yellow paper: a scalar value equal to the number of transactions sent from this address or, in the case of accounts with associated code, the number of contract-creations made by this account.

• an attribute of the originating address, not of the transaction

• transaction nonce different from the Bitcoin nonce
  • in Ethereum two types of nonces!
    • transaction nonce (different from Bitcoin nonce)
    • block nonce (like Bitcoin nonce)

• useful for
  • recording the order of the transaction
  • protection against transaction duplication
    • against double spending to the same address
ACCOUNT NONCES

- each time an account sends a transaction, the nonce increases by 1.
- the nonce is used to enforce the rules to consider a transaction valid.
  - transactions must be in order
    - a transaction with a nonce of 1 cannot be mined before one with a nonce of 0.
  - transaction cannot be skipped.
    - a transaction with a nonce of 2 cannot be mined if the miner has not already sent transactions with a nonce of 1 and 0.
ACCOUNT NONCES: PREVENTING ATTACKS

- imagine an adversary taking any existing transaction and resending it to be processed more times

- example:
  - Alice signed a transaction to send 10 ETH to Bob
  - Bob takes the transaction signed by Alice and repeatedly submits it for processing until all of Alice’s funds had been transferred to Bob

- with the account nonce
  - when the transaction is processed, Alice’s account’s nonce is incremented
  - when Bob resubmits the transaction, it is rejected by the miners because the nonce doesn’t match.
  - Bob is unable to change the nonce on the transaction without invalidating Alice’s signature
    - the transaction can only be applied once, exactly as Alice intended
ETHEREUM TRANSACTIONS: RECIPIENT

- The `to` field
  - a 20-byte Ethereum address
  - can be an EOA or a contract address

- no validation of the `to` field
  - any 20-byte value is valid.
  - if it is invalid, the ether sent is burnt.

- Bitcoin comparison
  - only one output address and no script in the output addresses
  - value directly inserted in the transaction, no reference to a previous transaction output
the main "payload" of a transaction contains the fields
- value and data

fields be significative or null
- only value significative: a payment
- only data significative: a function invocation
- both value and data significative: payment and invocation

value trasmitting a value to:
- EOA adds value to the balance of the target address account
- contract accounts:
  - execute the function named in the data payload of the transaction or a fallback function,
    - the function must be payable: can accept Ether from the caller.
  - if no function is found, increase the balance of the contract
ETHEREUM TRANSACTIONS: DATA AND VALUE

• Transmitting data payload to an EOA or contract account
  • EOA: allowed, but not specified in the Ethereum protocol
  • data payload for contracts accounts:
    • function selector:
      • the first 4 bytes of the Keccak-256 hash of the function’s prototype
      • allows to unambiguously identify which function to invoke.
    • function arguments:
      • encoded according to the rules for the various elementary types defined in the ABI (Application Binary Interface) specification
  • contract creation transactions
    • deploy a new contract on the blockchain, for future use
    • data payload contains the compiled bytecode which will create the contract
• **Gas Price**: the price of gas (in wei) the originator is willing to pay

• **Gas Limit**: The maximum amount of gas the originator is willing to buy for this transaction

• **v, r, s**: three components of an ECDSA digital signature of the originating EOA
  - allow to compute the address of the account sending the transaction
MESSAGES OR INTERNAL TRANSACTIONS

- same structure of transactions, but no gas limit
  - why no limit?

- gasLimit is determined by the external creator of the original transaction, by some EOA
  - must be enough to carry out all the transactions
    - including any sub-executions that occur as a result of that transaction, such as contract-to-contract messages.
  - if, in the chain of transactions and messages, a particular message execution runs out of gas,
    - that message’s execution will revert, along with any subsequent messages triggered by the execution.
  - however, the parent execution may not need to revert.
WHAT IS STORED IN AN ACCOUNT STATE?

- **nonce** number of transactions sent from external account, number of contracts created if contract account
- **balance** number of Wei owned by this address
- **storageRoot**
  - hash of the root node of a Merkle Patricia tree
  - the tree encodes the hash of the storage contents of this account, empty by default.
- **codeHash**
  - hash of the code of contract account account. Hash(“ ”) for external accounts
The state of the Ethereum network is the state of all its accounts

- the entire Ethereum network agrees on the current balance, storage state, contract code, etc. of every single account

- Ethereum network state is updated when a new block is mined
  - block mining takes the previous state and produces a new network state
  - every node has to agree upon new network state
the Ethereum network uses a flooding protocol to propagate the transactions

transaction propagation starts with the originating Ethereum node creating a signed transaction.
  • the transaction is validated and then transmitted to all the other Ethereum nodes that are directly connected to the originating node.
  • on average, each Ethereum node maintains connections to at least 13 other nodes

  • each neighbor node validates the transaction as soon as it receive it.

  • if it is valid, they store a copy and propagate it to all their neighbors (except the one it came from).
RECORDING ON THE BLOCKCHAIN

• valid transactions will eventually be included in a block of transactions by the miners

• when a miner insert the transaction into a block
  • transactions also modify the state of the Ethereum singleton
    • modify the balance of an account (in the case of a simple payment)
    • invoking contracts that change their internal state.

• transaction receipts
  • records the change made, including generated events
formally, a (deterministic) state machine is defined as \((I, S, s_0, \sigma, F)\), where:

- \(I\): a set of inputs.
- \(S\): a set of states.
- \(s_0\): an initial state
- \(\sigma\): a state-transition function defined as \(\sigma: S \times I \rightarrow S\)
- \(F\): a set of final states \(F \subseteq S\)

- at any given time, a state machine can be in exactly one state.
- in a distributed setting
  - a state machine for each replica
  - all machine replicas must be in an identical state, given the same input.
THE ETHEREUM STATE MACHINE

- the state of Ethereum includes all accounts’ information
  - state/storage
  - Ether balances
  - contracts code

- the state-transition function takes the state and a transaction and outputs a new state.

![Diagram of Ethereum state machine](image)
Miners update the state of the distributed state machine
A simplified account state is shown in the figure:

- account address
- balance
- code and state of the variables for contract accounts
AN ETHEREUM TRANSACTION

A simplified transaction is shown in the figure:

Transaction

From: 14c5f88a
To: bb75a980
Value: 10
Data: 2, CHARLIE
Sig: 30452fdeeb3d
      f7959f2ceb8a1


- an example of state transition fired by a transaction

**State**

| 14c5f8ba: |
| - 1024 eth |
| bb75a980: |
| - 5202 eth |
| if t.contract.storage[tx.data[0]]: |
| contract.storage[tx.data[0]] = tx.data[1] |
| [0, 235235, 0, ALICE .. | |
| 892bf92f: |
| - 0 eth |
| send(tx.value / 3, contract.storage[0]) |
| send(tx.value / 3, contract.storage[1]) |
| send(tx.value / 3, contract.storage[2]) |
| [ALICE, BOB, CHARLIE ] |
| 4096ad65: |
| - 77 eth |

**Transaction**

- From: 14c5f8ba
- To: bb75a980
- Value: 10
- Data: 2, CHARLIE
- Sig: 30452fdeeb3d f7959f2ceb8a1

**State'**

<p>| 14c5f8ba: |
| - 1014 eth |
| bb75a980: |
| - 5212 eth |
| if t.contract.storage[tx.data[0]]: |
| contract.storage[tx.data[0]] = tx.data[1] |
| [0, 235235, CHARLIE, ALICE .. | |
| 892bf92f: |
| - 0 eth |
| send(tx.value / 3, contract.storage[0]) |
| send(tx.value / 3, contract.storage[1]) |
| send(tx.value / 3, contract.storage[2]) |
| [ALICE, BOB, CHARLIE ] |
| 4096ad65: |
| - 77 eth |</p>
<table>
<thead>
<tr>
<th></th>
<th>Bitcoin</th>
<th>Ethereum</th>
</tr>
</thead>
<tbody>
<tr>
<td>First issued</td>
<td>January 2009</td>
<td>July 2015</td>
</tr>
<tr>
<td>Distributed Consensus</td>
<td>Proof-of-work</td>
<td>Proof-of-work, moving to proof-of-stake (Casper)</td>
</tr>
<tr>
<td>Block mining rate</td>
<td>≈ 10 minutes</td>
<td>10 - 20 seconds</td>
</tr>
<tr>
<td>Mining reward</td>
<td>12.5 BTC / block mined, halves every 210k blocks (≈ 4 years)</td>
<td>5 ETH / block mined, constant¹</td>
</tr>
<tr>
<td>Total Supply</td>
<td>Finite; capped at 21 M</td>
<td>Infinite</td>
</tr>
<tr>
<td>Monetary Policy</td>
<td>deflationary</td>
<td>inflationary</td>
</tr>
<tr>
<td>Tx. Processing</td>
<td>Rests on miners (+ tip)</td>
<td>Rests on everyone (+ tip)</td>
</tr>
<tr>
<td>Price</td>
<td>1 BTC = 6262.05 EURO</td>
<td>1 ETH = 188,24 EURO</td>
</tr>
</tbody>
</table>
SMART CONTRACT PROGRAMMING LANGUAGES

- language for smart contracts development:
  - **Solidity**
    - the primary language for writing smart contracts.
    - object oriented programming language.
    - looks like Javascript.
  - Serpent.
  - LLL.
  - Mutant.

- they are all compiled into EVM byte code before being deployed to the blockchain.
SMART CONTRACT AT A GLANCE

• how does a smart contract look like?
• in the next slides some examples giving the flavour of smart contracts
• a more rigorous presentation of Solidity in the next lessons
A TOKEN CONTRACT

- token systems have a variety of applications
  - subcurrencies
  - secure unforgeable coupons
  - reputation systems

- in the following slide: a contract that:
  - sets token rate (value of each token in Ether).
  - allows people to buy tokens by paying Ether within a predefined period of time.
  - maintains an unforgeable public token balance.
A TOKEN CONTRACT IN SOLIDITY

```
pragma solidity ^0.4.4;

contract Example2 {
    mapping (address=> uint) public token_balances;
    uint exchange_Rate;
    uint public end;
    uint public validity_period;

    function Example2 () {
        exchange_Rate = 2;
        validity_period = 10000;
        end = now + validity_period;
    }

    modifier notExpired {
        require (now <= end);
        -;
    }

    function buyToken () payable notExpired external {
        require (msg.value >= exchange_Rate);
        uint amount = (msg.value) / exchange_Rate;
        token_balances[msg.sender] += amount;
    }
}
```

In this example, the contract function does change the contract state, i.e. the value of “token balances” is updated by function: buyToken().

- **require():** is a function for error handling. When the condition is not met, it will undo all changes made to the state in the current call (but it will keep the gas).

- **The tag: payable** allows the function to receive Ether; otherwise, it cannot receive Ether.

- **The tag: notExpired** allows the function to accept calls only within a period of time. We can defined it by using a modifier.

- **msg.sender** here is the address of the account that calls buyToken().
In the following slide, a contract that:

- sets token rate = value of each token in Ether.
- sets threshold.
- keeps track of the number of sold tokens.
- allows people to buy and transfer tokens.
- maintains an unforgeable public token balance.
- when the value of token sold exceeds the threshold, it automatically sends the value of sold tokens (in Ether) to a special account address.
pragma solidity ^0.4.4;

contract Example4 {

    mapping (address => uint) public token_balance;
    uint exchange_Rate;
    uint public threshold;
    uint public total_token_sold;
    address Ether_recipient;
    uint converter;

    function Example4 (){
        exchange_Rate = 2;
        threshold = 8;
        converter = 1000000000000000000000000;
        total_token_sold = 0;
        Ether_recipient = 0x7dffe4d270b09c5847072796e7e7e6c6ae209df;
    }

    function buyToken () payable external {
        uint amount = (msg.value) / exchange_Rate;
        token_balance[msg.sender] += amount / converter;
        total_token_sold += amount / converter;
        if (total_token_sold * exchange_Rate > threshold) {
            Ether_recipient.transfer (converter * total_token_sold);
            total_token_sold = 0;
        }
    }

    function transfer_token (address recipient, uint amount){
        require (token_balance[msg.sender] >= amount);
        token_balance[msg.sender] -= amount;
        token_balance[recipient] += amount;
    }

}
• Solidity in depth

• Ethereum Block Structure:
  • logs
  • owners and uncles in Ethereum
    • GHOST/SPECTRE protocols
  • Merkle Patricia Tries
  • block difficulty
  • Ethereum Proof of Work and Proof of Stake
  • Ethereum Execution Model via the EVM