P2P Systems and Blockchains
Spring 2018,
instructor: Laura Ricci
laura.ricci@unipi.it
Lesson 17:
ETHEREUM
SMART CONTRACTS
09/05/2018
SMART CONTRACTS BEYOND BITCOIN

- a first step forward with respect to Bitcoin: **smart contracts**
- **contract**
  - formalizes a relationship and contains a set of promises made between principals.
- **smart contract** (defined in the nineties by Nick Szabo, a cryptographer): a new way to formalize and secure digital relationships
  - based on the translation of contractual clauses into code and the use of this code in software.
  - aim to remove the need for trusted intermediaries and make it more difficult for malicious parties to undermine compliance with the contract terms
  - more functional compared to paper-based, can reduce costs, for either principals, third parties,...
  - uses cryptography and other security mechanisms, in order to secure algorithmically specifiable relationships from being breached and ensure the agreed upon terms are satisfied.
SMART CONTRACTS

• A simple 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

• a smart contract is similar to a real contract, but:
  • runs on a blockchain
  • is run by many parties rather than being controlled by a single one.
a smart contract is some code automating the “if this happens then do that” part of traditional contracts.

better than a normal contract
• computer code behaves in expected ways and doesn’t have the linguistic nuances of human languages.
• code is replicated on a blockchain all the nodes execute the contract and come to an agreement on the results of the code execution.

not limited to the automation of contracts
SMART CONTRACTS PROPERTIES

Centralized Contracts: my bank is the ultimate guardian of my bank account and has complete control:

- can arbitrarily add money to my account (ha...ha..), subtract it: it may happen!
- code sits on one computer and is executed by one party (the bank) which must be trusted

Smart contracts

Distributed smart contracts

- distributed blockchain and consensus mechanisms assure that multiple nodes are executing my contract
- results are compared by all nodes and they only change their own version of the ledger if they agree the results.
- anything that doesn’t conform to pre-agreed rules is rejected by other participants
CONTRACTS AND SMART CONTRACTS

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 can be 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
SMART CONTRACTS IN BITCOIN?

- use scripting language of Bitcoin to write smart contracts?

- the scripting language as implemented in Bitcoin has 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 and it can be changed by a program/contract.

- hard to build stateful general contracts on Bitcoin.

- Bitcoin state: implicit state in unspent transaction outputs (UTXO)
  - Bitcoin scripts do not explicitly modify the state in UTXO
    - only limited modification through transactions
    - they either succeed or fail.
• Ethereum is a decentralized platform designed to run smart contracts
  • like a distributed computer to execute code
  • defines an account-based blockchain
  • distributed state machine - transactions change global state
    • transactions == state transaction function

• Ethereum has a native asset called ether
  • basis of value in the Ethereum ecosystem
  • Used to pay code execution
  • given as mining rewards
ETHEREUM

• small computer programs called smart contracts are
  • replicated
  • processed on all the computers on the network, without a central coordinator

• 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 (smart contracts)

• similarities with Bitcoin:
  • manages a blockchain with blocks of data and smart contracts
  • the blockchain is public and permissionless
  • has a Proof of Work mining (even if Proof of Stake is proposed)
  • has a native cryptocurrency: ether (ETH)
Thanks to this guy!

Vitalik Buterin (January 31th 1994)
ETHEREUM AND SMART CONTRACTS

• live on the Ethereum blockchain

• have their own Ethereum address and balance
  • set up a smart contract by creating a new account with some code in it
  • upload it to the Ethereum blockchain in a transaction.

• can send and receive messages

• are activated when they receive a transaction, and can be deactivated

• have a fee per CPU step and for storage unit
ETHEREUM AND SMART CONTRACTS

a 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.
externally owned accounts (EOA)

- owned by some external entity
- controlled by private keys (users)
- can send transactions to transfer ether or trigger a smart contract
- contain
  - address
  - ether balance
(smart) contract accounts

- “owned” by a contract
- controlled by contract code
- code execution triggered by transactions or messages
- contains
  - address
  - associated contract code
  - persistent storage
  - ether balance
A transaction refers to a signed data package that stores a message

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

a bridge between the external world to the internal state of Ethereum.
CONTRACT ACCOUNTS

- Contract Accounts: cannot send transactions in an autonomous way

- A contract account cannot invoke itself.
  - Must be invoked by another (externally owned or contract) account

- Can generate messages as a reply to transactions they have received from an EOA or from another contract account

- Any action occurring on the Ethereum blockchain is always set in motion by transactions fired from externally controlled accounts.
EXTERNALLY OWNED ACCOUNTS

- can send messages to other externally owned accounts OR to other contract accounts by creating and signing a transaction using its private key.

- a message between two externally owned accounts is simply a value transfer (transaction)

- a message from an externally owned account 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
    - create new contracts
MESSAGES OR INTERNAL TRANSACTIONS

- can be sent from a contract to another contract
  - essentially function calls
- a message is like a transaction, but produced by a contract and not an external actor
- when one contract sends an internal transaction to another contract, the associated code is executed.
- often referred as internal transactions
## EXTERNAL AND CONTRACT ACCOUNTS

<table>
<thead>
<tr>
<th>Feature</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>✓</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>✗</td>
</tr>
<tr>
<td>Can send signed transactions</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Can create contracts</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Can send unsigned transactions</td>
<td>✓</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.
transactions and messages

- contracts can talk to other contracts via “messages” or “internal transactions” to other contracts.

messages:
- similar to transactions, but they are not generated by externally owned account, they are generated by contracts.
- only exist within the Ethereum execution environment.
STATE MACHINES

- 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, and
  - \(\sigma\): a state-transition function defined as \(\sigma: S \times I \rightarrow S\)
  - \(F\): a set of final states \(F \subset S\)

- 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 BITCOIN STATE MACHINE

• Bitcoin is a specialised version of a state machine

• The ledger of Bitcoin can be thought of as a state defined as the ownership status of all unspent bitcoin.

• The initial state is the genesis block (i.e. 1st block in the blockchain).

• State-transition function takes the state and a transaction and outputs a new state
THE ETHEREUM STATE MACHINE

- in Ethereum, the state includes all accounts’ information (e.g. state/storage, Ether balances, contracts code, etc).
- the state-transition function takes the state and a transaction and outputs a new state.
- miners keep track of accounts Ether balances and update them.
**BLOCK MINING**

- Miners receive transactions (Tx-1, Tx-2, ..., Tx-n).
- They create a block containing:
  - Previous block
  - A set of TXs
  - New State Root
  - Receipt Root
  - Nonce

- Verify transactions & execute all code to update the state.
- SHA3(Block) < D

- Broadcast Block
ETHEREUM GAS AND FEES

• an immediate issue with smart contracts: what if our contract has an infinite loop?
  
  function foo()
  {while (true) {
    /* Loop forever! */
  }

• every node on the network will get stuck executing the loop forever!

• by the halting problem, it is impossible to determine ahead of time whether the contract will ever terminate
  ⇒ Denial of Service Attack!

• solution: gas and fees
ETHEREUM GAS AND FEES

- executing a smart contract is not free, incurs a fee!
- fee is paid in gas
  - some gas unit for every Ethereum Virtual Machine opcode
- gas price: amount that each user is willing to spend for every unit of gas, measured in “gwei.”
  - “Wei” is the smallest unit of Ether, gwei = 1,000,000,000 Wei.
Ether is the internal currency of Ethereum and is used both to pay transactions and computation fees.

Ethereum has a list of denominations each of which has its own name.

- the smallest denomination is called Wei

<table>
<thead>
<tr>
<th>Unit</th>
<th>Wei</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wei</td>
<td>1</td>
</tr>
<tr>
<td>Kwei</td>
<td>1,000</td>
</tr>
<tr>
<td>Mwei</td>
<td>1,000,000</td>
</tr>
<tr>
<td>Gwei</td>
<td>1,000,000,000</td>
</tr>
<tr>
<td>Szabo</td>
<td>1,000,000,000,000</td>
</tr>
<tr>
<td>Finney</td>
<td>1000,000,000,000,000</td>
</tr>
<tr>
<td>Ether</td>
<td>1000,000,000,000,000,000</td>
</tr>
</tbody>
</table>
ETHEREUM GAS AND FEES

- The sender of a transaction sets in the transaction:
  - a gas limit (startgas, gaslimit)
  - a gas price

- The product of gas price and gas limit is the maximum amount of Wei that the sender is willing to pay for executing a transaction.

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

\[
\begin{align*}
\text{Gas Limit} & = 50,000 \\
\text{Gas Price} & = 20 \text{ gwei} \\
\text{Max transaction fee} & = 0.001 \text{ Ether}
\end{align*}
\]

- 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.
ETHEREUM GAS AND FEES

- 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:

- startgas * 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
  - startgas * gasprice are not refunded.

- What about the infinite loop?
  - Ethereum still allows the infinite loop
  - however, the attacker attempting to DoS the network has to pay enough ether to fund the DoS

- purchasing gas like purchasing distributed, trustless computational power.
ETHEREUM ACCOUNTS: DETAILS

- Each account is identified by an address
  - 20-byte address, 160-bit identifier

- State of an account (EOA or contract)
  - nonce: (not PoW nonce!)
    - number of transactions sent from the EOA’s address or number of contracts created by the contract account
  - ether balance
    - number of Wei owned by this address.
  - storageRoot:
    - hash of the root node of a Merkle Patricia tree. This tree encodes the hash of the storage contents of this account, empty by default.
  - codeHash:
    - hash of the code of this account, empty for EOAs.
ACCOUNT NONCES

- in Ethereum, every transaction has a nonce.

- The nonce is the number of transactions sent from a given address.

- each time you send 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: you cannot have a transaction with a nonce of 1 mined before one with a nonce of 0.
  - transaction cannot be skipped. You cannot have a transaction with a nonce of 2 mined if you have not already sent transactions with a nonce of 1 and 0.
ACCOUNT NONCES

- Account nonce inserted to prevent double-spending

- In order for a double-spend to be "successful", you typically...
  - Send a transaction to one party (nonce 1)
  - Wait for it to register
  - Collect something in return from this first transaction
  - Quickly send another transaction with a high gas price (nonce 2)
  - Have this second transaction mined first, therefore invalidating the first transaction.

- But this is not possible:
  - the previous procedure will not work as the second transaction (nonce 2) cannot be mined before the first transaction (nonce 1).

- Different from Bitcoin, because Ethereum has accounts!
THE ETHEREUM STATE

- need an efficient representation of the Ethereum state.
- to be seen in the next slides

External Account

Contract Account

Contract Account

External Account

State

14c5f8ba:
- 1024 eth

bb75a980:
- 5202 eth
  if !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
ETHEREUM AND TRANSACTIONS

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

- transactions contains (different fields for message calls or contract creation)
  - `nonce`
    - a count of the number of transactions sent by the sender.
  - `gasLimit`
  - `gasPrice`
  - `to`: the recipient of the message
  - `v,s,r`: to generate the signature identifying the sender
  - `value`: amount of Wei to be transferred
  - `data`: the input data (i.e. parameters) of a message call
    - can be read by the contract code
    - only for message call
  - `init` (only for contract-creation)
    - to initialize the new contract account.
A simplified transaction
TRANSACTION EXECUTION: AN EXAMPLE

- An example of state transition fired by a transaction
MESSAGES OR INTERNAL TRANSACTIONS

- same structure of transactions, but no gas limit
- why no GasLimit?
- GasLimit is determined by the external creator of the original transaction, by some EOA
  - must be enough to carry out all the transaction, 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, then that message’s execution will revert, along with any subsequent messages triggered by the execution.
  - however, the parent execution may not need to revert.
Ethereum (defined also a planetary scale computer) with respect to Bitcoin:

- time between blocks is around 14 seconds, compared with Bitcoin’s ~10 minutes

- A larger amount of forks
  - Introduction of uncles

- has smaller blocks

- built-in Turing complete programming language

- consume gas to execute contracts pay gas with ether
### COMPARISON BITCOIN ETHEREUM

<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, constant(^1)</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 = 9102.00 $</td>
<td>1 ETH = 728.51 $</td>
</tr>
</tbody>
</table>
SMART CONTRACT PROGRAMMING LANGUAGES

- 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.
SOLIDITY

- high level language whose syntax is similar to JavaScript
- It is like object oriented language and supports inheritance.
- however, it has some new types as well.
- special variables and functions mainly used to get information about the blockchain. For instance:
  - `msg.sender`: provides the message sender’s address.
  - `msg.value`: provides the number of Ether/Wei sent with the message.
  - `now`: current/head block timestamp.
- Remix (remix.ethereum.org.):
  - a web based environment to write, compile and debug smart contracts.
  - you can “play” with smart contract there
SOLIDITY DATA TYPES

- **bool**: true and false.
- **integers**: int (int8,…,int256) and uint (uint8,…,uint256).
  - signed and unsigned integers of various sizes
- **address**: Ethereum address 20 bytes value. It has several members:
  - **balance**: balance of the address in Wei
  - …
- **byte arrays**: bytes1,…, bytes32, bytes.
  - fixed sized byte arrays
- **dynamic arrays**.
  - dynamically sized byte-arrays
SOLIDITY DATA TYPES

- String.
- Enum.
- Struct.
- mapping (KeyType => ValueType).

It is like a hash table:
  - store a value using a key:
    - reputation[0x111] = 5
  - later, given the key, you can retrieve the value:

```
var val = reputation[0x111]
```

5 will be assigned to it
we can create an (externally owned) account and address in Ethereum by using one of the following methods:

- **Wallet/software**: MetaMask, MyEtherWallet.
  The wallet also produces a private key. A password is used to encrypt the private key. MetaMask generates password for us, but we have to provide our own password to MyEtherWallet.

- **command line interface** (e.g. geth) in Ethereum client node,
  ```
  geth new account.
  ```
  It also generates a private key and allows us to provide a password.

- a new contract can be sent to the chain by executing a transaction. An account and address for the contract are generated by the network and provided to the sender.
SMART CONTRACT: EXAMPLE 1

• How does a smart contract look like?

• Let’s design a contract that:
  • holds its owner/creator address.
  • assigns an arbitrary value to its state.
  • after it is deployed to the blockchain, each time a function of it is called and a value is passed to it, it doubles the value and returns the result.
```solidity
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:

```solidity
instance.double_it.call (5, {from:"0x11"});
```

“instance” is just an instance of the contract.

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 ().
TOKEN SYSTEM: EXAMPLE 2

- Token systems have a variety of applications (e.g. subcurrencies, secure unforgeable coupons, or reputation systems).

- Let’s design 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.
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().
Special variables mainly used to provide information about the blockchain or are general-use utility functions.

- `block.coinbase(address)`: current block miner’s address
- `block.difficulty(uint)`: current block difficulty
- `block.gaslimit(uint)`: current block gaslimit
- `block.number(uint)`: current block number
- `block.timestamp(uint)`: current block timestamp as seconds since unix epoch
- `gasleft() returns (uint256)`: remaining gas
- `msg.data(bytes)`: complete calldata
- `msg.gas(uint)`: remaining gas - deprecated in version 0.4.21 and to be replaced by `gasleft()`
- `msg.sender(address)`: sender of the message (current call)
- `msg.sig(bytes4)`: first four bytes of the calldata (i.e. function identifier)
- `msg.value(uint)`: number of wei sent with the message
- `now(uint)`: current block timestamp (alias for `block.timestamp`)
- `tx.gasprice(uint)`: gas price of the transaction
- `tx.origin(address)`: sender of the transaction (full call chain)
- **require** can be used to check for conditions and throw an exception if the condition is not met.

  - the **require** function should be used to ensure valid conditions, such as inputs, or contract state variables are met,

- **modifiers** can be used to easily change the behaviour of functions.

  - for example, they can automatically check a condition prior to executing the function.
where and how are `msg.sender` and `msg.value` defined?

they are defined in the transaction that invokes the function. Function `buyToken()` in a transaction is invoked as follows:

```
contractInstance.buyToken.sendTransaction ({from: "0x111", value: 10, gas: 4200000});
```

Note: we did not define these parameters, in our original function, but EVM allows us to include them in any function.
Let’s design a contract that:

• sets token rate (value of each token in Ether).
• sets threshold.
• keeps track of the number of token it sold.
• 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 an account address (e.g. “0x7dff4ad270b09c5847072796ee7e7c6c6ae209df”).
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 = 1000000000000000000000000000000000000000000000000000000000000000000;
        total_token_sold = 0;
        Ether_recipient = 0x7d4f4ad270b09c5947072796ee7e7c6c6ae209df;
    }

    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;
    }
}
What does a bank need to do?
- allow deposits.
- allow withdrawals.
- balance checks

```solidity
contract SimpleBank {
    mapping (address => uint) private balances;
    address public owner;
    event LogDepositMade(address accountAddress, uint amount);

    function SimpleBank() {
        owner = msg.sender;
    }

    function deposit() public returns (uint) {
        balances[msg.sender] += msg.value;
        LogDepositMade(msg.sender, msg.value);
        return balances[msg.sender];
    }

    function withdraw(uint withdrawAmount) public returns (uint remainingBal) {
        if(balances[msg.sender] >= withdrawAmount) {
            balances[msg.sender] -= withdrawAmount;
            if(!msg.sender.send(withdrawAmount)) {
                balances[msg.sender] += withdrawAmount;
            }
        }
        return balances[msg.sender];
    }

    function balance() constant returns (uint) {
        return balances[msg.sender];
    }

    function () {
        throw;
    }
}
```
THE BANK CONTRACT: EXAMPLE 4

• **contract** similar to **classes** in other languages (class variables, inheritance, etc.)

• declare state variables outside function, persist through life of contract

• **mapping**: dictionary that maps addresses to balances
  • **private** other contracts can't directly query balances

• **event** publicize actions to external listeners

```
1: contract SimpleBank {

  mapping (address => uint) private balances;

  address public owner;

  event LogDepositMade(address accountAddress, uint amount);

```
• **constructor** can receive one or many variables here; only one allowed

• **msg** provides details about the message that has been sent to the contract

• **msg.sender** is contract caller (address of contract creator)

```solidity
function SimpleBank() {
    owner = msg.sender;
}
```
Deposit( ):

- takes 0 parameters, but takes Ether from the transaction!
- **public** makes externally readable (not writeable) by users or contracts
- returns user’s balance as an unsigned integer (**uint**)
- **balances[msg.sender]** no **this** or **self**. required with state variable
- all values set to data type's initial value by default
- event fired
Balance( ): constant prevents function from editing state variables

• returns user’s balance
• allows function to run locally/off blockchain
• ( ) : Fallback function

• Typically, called when invalid data is sent or on a call to the contract if none of the other functions match the given function identifier
• Added so ether sent to this contract is reverted if the contract fails otherwise, the sender's money is transferred to contract
• throw : throw reverts state to before call
CONCLUSIONS: BLOCKCHAINS BEYOND BITCOIN

Blockchain 1.0: currency, Bitcoin, Monero

Blockchain 2.0:
- Introduction of smart contracts to realize decentralized applications, Ethereum
- Further financial applications

Blockchain 3.0:
- Non cryptocurrencies related distributed ledgers
  - Electronic voting
  - Health care
  - Identity management systems
  - Decentralized notary
  - Supply chains Management
  - IoT