

# Ethereum Smart contracts development

With Javascript (2020)



Andrea Lisi, [andrealisi.12lj@gmail.com](mailto:andrealisi.12lj@gmail.com)



# Smart contracts

A smart contract is a software stored and executed by all the full nodes of the network

In Bitcoin the smart contracts, implemented in Script, enable the execution of the transactions, for example verifying the signatures

In Ethereum the smart contracts, implemented in Solidity, may include additional logic, for example more complex conditions



# Smart contracts

As well as blocks and transactions, the smart contracts are:

- Immutable: its code cannot be changed
- Transparent: its (byte)code can be visualized and executed



# Smart contracts



Stack machine

No internal state

Intentionally simple



Virtual machine

Internal state

Turing complete



# Part 1

# Solidity overview

A brief summary of a Solidity smart contract





# Smart contracts: structure

A smart contract is similar to a Java class

It is composed by:

- Declaration
- A State (attributes)
- A list of functions (methods)

```
contract MyContract {  
    // State  
    uint public value;  
  
    // Functions  
    constructor() public {  
        value = 1;  
    }  
    function increase() public {  
        value = value+1;  
    }  
}
```



# Smart contracts: state

State variables determine the state of that smart contract

Solidity supports basic data types:

- Fixed length
  - bool, (u)int, bytes32, address
- Variable length
  - bytes, string
- array, mapping(key => value)



# Smart contracts: complex types



## Arrays

- Typical arrays, they can be either fixed length or dynamic length
- Pushing an element is ok, removing an element can be costly
  - Leave a blank hole, replace with last element (breaks ordering), shift

## Mappings

- Random access, all values has Zero value by default
- It is not possible to iterate over mappings unless you keep a list of all the keys with significant value





# Smart contracts: functions

Functions compose the code of the smart contract

Functions can be labeled in case they interact with the state:

- A **view** function **only reads** the state;
- A **pure** function does not read or write the state;
- Otherwise, the function writes (and reads) the state
  - The state modification will be placed in a transaction
  - It will be written on the blockchain
  - Therefore, it costs a fee to the user



# Fees and gas

Each writing function (and sending Ether) costs a fee to the user

- The fee is proportional to the required amount of computation (EVM OPCODES)
- Each OPCODE has a costs named **gas**



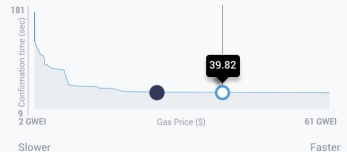
### Customize Gas Close

Advanced

New Transaction Fee	~Transaction Time
\$0.05	~32 sec

Gas Price (GWEI)	Gas Limit
<input type="text" value="26.117404834"/>	<input type="text" value="21000"/>

Live Gas Price Predictions



Send Amount	0 ETH
Transaction Fee	0.000548 ETH
New Total	<b>0.000548 ETH</b>
	\$0.05

[SAVE](#)

# Fees and gas

Each writing function (and sending Ether) costs a fee to the user

- Before each transaction a user can set in their wallet:
  - The **gas price**: i.e. how much Ether they are willing to pay for each unit of gas
  - The **gas limit**: i.e. how many units of gas they are willing to consume for that transaction



### Customize Gas Close

Advanced

New Transaction Fee ~Transaction Time  
\$0.05 ~32 sec

Gas Price (GWEI) Gas Limit

Live Gas Price Predictions

Send Amount 0 ETH  
Transaction Fee 0.000548 ETH  
New Total 0.000548 ETH  
\$0.05

[SAVE](#)



# Smart contracts: visibility

State variables and functions can have different visibilities

- **Private**
  - A private state variable or function is exposed only to the contract itself
- **Public**
  - A public function is exposed to other contracts; a public state is a shortcut creating a getter function with the name of the variable



# Smart contracts: visibility

State variables and functions can have different visibilities

- **Internal**
  - An internal state variable or function is exposed to child contracts and the contract itself
- **External**
  - (Only functions) An external function is exposed **only** to other contracts. They are more efficient with large array data
    - **Warning:** `foo()` does not work; `this.foo()` does
    - <https://ethereum.stackexchange.com/questions/19380/external-vs-public-best-practices>



# Accounts



In Ethereum any entity (account) has associated an address and a balance (in Ether)

The two types of accounts are:

- **Contract Accounts:** are controlled by code, and each received message activates its code
- **Externally Owned Accounts (EOA):** are controlled by private keys and send messages signing transactions



# Smart contracts: receive Ether

A smart contract function can be labelled as **payable** if it *\*expects\** to receive Ether

- Once received the Ether the contract's balance is automatically increased
- **msg.value** stores the received Ether (uint)

```
function foo() public payable {  
    uint payer = msg.sender; // Who sent the Ether  
    uint received = msg.value; // How much  
    uint current = address(this).balance; // The current balance of the contract  
}
```



# Smart contracts: send Ether

If the contract has positive balance, then it can send Ether as well

- Solutions with a gas limit fixed to be 2300 for the caller
  - **address.send(amount)** Send amount to *address*, returns True if everything goes well, False Otherwise
  - **address.transfer(amount)** Throws exception if it fails
- Solution with settable gas limit (not very secure)
  - **address.call.value(amount)( )** Returns True or False
  - Set the gas limit: **address.call.value(msg.value).gas(20317)( )**





# Smart contracts: receive Ether

If a smart contract receives plain Ether without the means of a function, there are three possible outcomes:

- Trigger the **receive** function ( $\geq$  Solidity 0.6.\*)
  - Dedicated to receive plain Ether with a transaction without calldata (the data parameter of a transaction)
- Trigger the **fallback** function
  - This function is executed when no other function is matched
- Throws exception if none of them is provided



# Smart contracts: receive Ether

```
contract Example {
    address payable known_receiver;
    function forward() public payable {
        known_receiver.transfer(msg.value);
    }

    // All of them have in their body at most 2300 units of gas of computation if called by
    // send() or transfer()
    receive() external payable {} // strict syntax
    fallback() external payable {} // fallback f Solidity >= 0.6.*
    function() public payable {} // fallback f Solidity < 0.6.*
}
```



# Smart contracts: events



It is possible to declare an event in Solidity similarly to a function, and it can be fired with the **emit** keyword

- Events are placed in the transaction log, useful for Ethereum clients

```
contract Example {
    event click();
    event executed(address sender);

    function press_click() public {
        emit click();
        emit executed(msg.sender);
    }
}
```



# References

Solidity documentation V 0.6.6: <https://solidity.readthedocs.io/en/v0.6.6/contracts.html>

Accounts: <https://github.com/ethereum/wiki/wiki/White-Paper#ethereum-accounts>

Sending Ether:

<https://medium.com/daox/three-methods-to-transfer-funds-in-ethereum-by-means-of-solidity-5719944ed6e9>

Best practices: <https://consensys.github.io/smart-contract-best-practices/>

Data management: <https://blog.openzeppelin.com/ethereum-in-depth-part-2-6339cf6bddb9/>



# Smart contracts: development

It is possible to implement Ethereum smart contracts with the Solidity programming language

Smart contracts can be developed and executed within:

- The browser IDE Remix, <https://remix.ethereum.org/>
- The CLI tool Truffle, <https://www.trufflesuite.com/truffle>



# Part 2

# The Web3 library

An interface to interact with smart contracts





# Web3

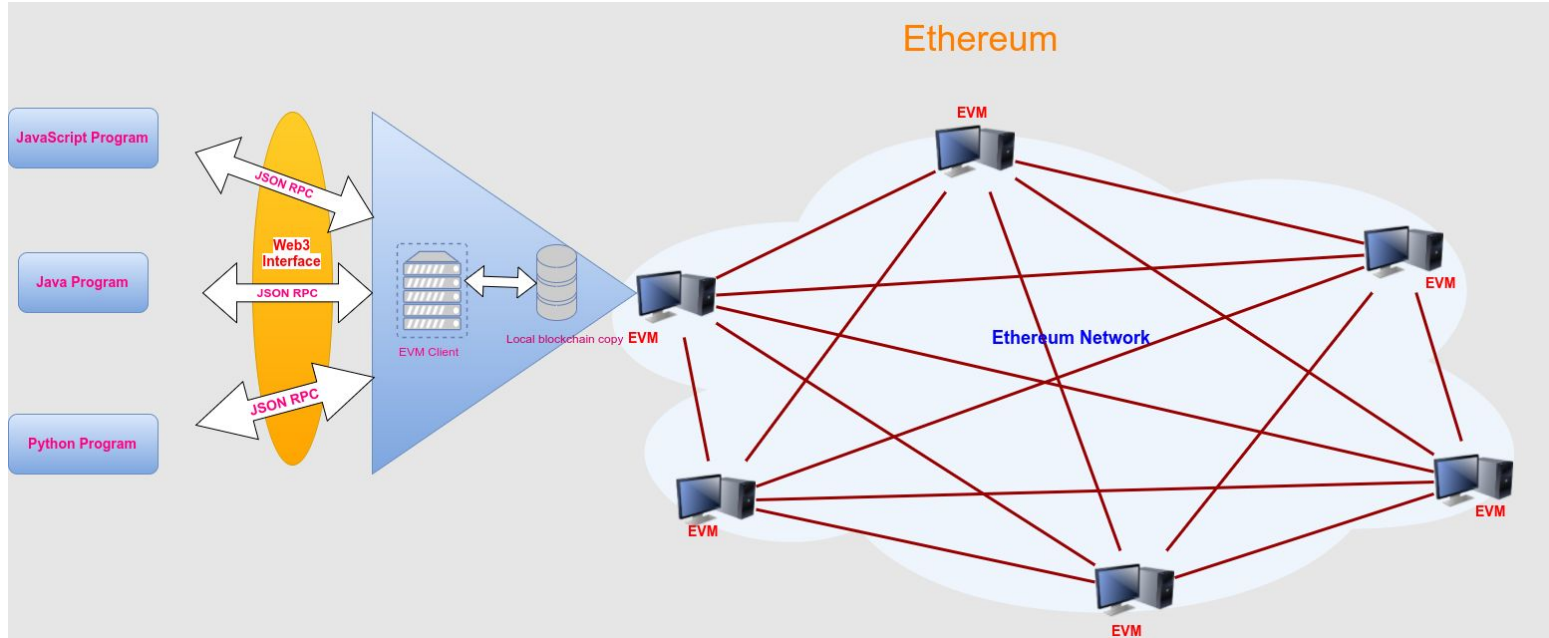
Web3 is an interface to the Ethereum network

It interacts with the Ethereum nodes by means via RPC protocol, Remote Procedure Call

- Communications are asynchronous

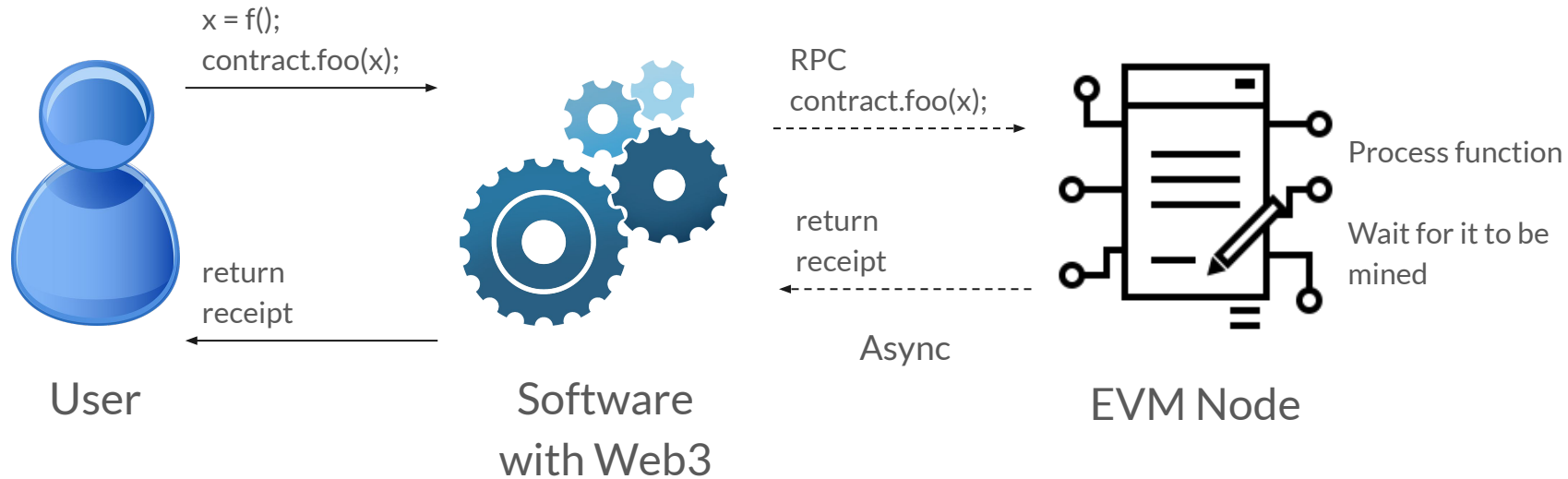
Software importing Web3 are able to communicate with smart contracts

# Web3





# Web3





# Web3 implementations

web3Js: JavaScript [W1]

web3J: Java [W2]

web3py: Python [W3]

web3.php: Php [W4]

hs-web3: Haskell [W5]





# NodeJs and Npm

In this tutorial we are going to use an environment based on Javascript  
We need **NodeJs** and **Npm** (Node Package Manager)





# Requirements: NodeJs

NodeJs is an environment to execute Javascript code on your machine instead on the browser:

- Write server-side Javascript code
- Modern frameworks for web development (ReactJs, AngularJs etc...)
- And Javascript desktop applications (ElectronJs)
- Install NodeJs
  - <https://nodejs.org/en/docs/>



# Requirements: Npm



Npm (Node Package Manager) is the tool to install NodeJs packages

- Local packages are installed in the `./node_modules/` directory
  - Libraries and utilities for a single project
- Global packages are all installed in a single folder in your system
  - CLI tools to be reused among many projects
- It is installed with NodeJs
  - <https://www.npmjs.com/get-npm>
  - <https://docs.npmjs.com/>



## References, Web3

[W1] Web3Js: <https://github.com/ethereum/web3.js>

[W2] Web3J: <https://github.com/web3j/web3j>

[W3] Web3Py: <https://github.com/ethereum/web3.py>

[W4] Web3.php: <https://github.com/sc0Vu/web3.php>

[W5] hs-Web3: <https://github.com/airalab/hs-web3>