**The Solidity Contract-oriented Programming Language**

Solidity documentation: <http://solidity.readthedocs.io/en/develop/index.html>

Solidity can be used to program a smart contract. This contract can be deployed to the Ethereum blockchain and can then be interacted with. The deployment of a contract will cost gas. Interacting with a contract can also cost gas. Gas is an internal pricing mechanism used to prevent spam on the network and as an incentive to miners. Ether is the “value token” of the Ethereum network; it is a cryptocurrency and can be transferred between participants and contracts. Contract files have a “.sol” extension.

**Functions**

Solidity is a function based programming language. Functions programmed in a contract can be called. The scope of these functions can be specified as external, public (default), internal or private. Everything inside a contract is always visible for external observers. Defining a function as private for example does not change this.

The *constructor* function has to be declared with the same name as the contract name. This function is only executed on deployment of the contract to the Ethereum network (can be useful for example to set the owner of the contract).

The *payable* keyword in a function definition enables this function to send/receive ether.

Function can have *modifiers* in their function definition to easily change the behavior of functions, for example to check a condition before executing the function (preconditions).

**Storage**

Solidity is a strongly typed programming language. This means variables have to be declared with a type. Variables in solidity can also have different scopes. Declaring a variable inside a function will make then local for the function. Declaring variables at the top of your contract will make them globally available in your contract. When adding the *public* addition to these global variables, it will make them readable from the outside.

**Types**

When variable type “var” is used, the type of the variable will be chosen based on the content given to it. Available types in solidity are:

* Int/Uint
* Bytes32
* Address
* Bool

All the normal arithmetic operators can be used when working with integer types.

All the normal comparison operators can be used when working with variables.

A structure is declared using “struct” and an array can be created using a mapping of this structure.

**Events**

Events can be triggered in a contract and allow the front-end to react on this. They have to be declared at the top of the contract and can be triggered in a function.

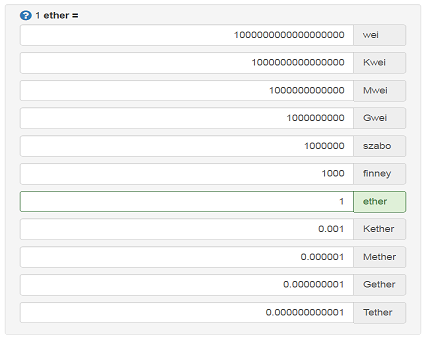
**Global variables**

A smart contract has a few global variables always available. These variables are:

* Block (information about the current block)
* This (information about the contract)
* Msg (information about the current message)
* Now (current block timestamp)
* Tx (information about the transaction)

**Units**

Ether is the main currency of the Ethereum network, but currency inside contracts is measured in Wei. This is a much smaller unit and converts to ether in the following way:

****

**Oracles**

Smart contracts live like in a walled garden they cannot fetch external data on their own. Oraclize is there to help since it acts as a data carrier, a reliable connection between Web APIs and the contract. Oraclize main goal as a company is to provide a way for smart contracts to break free of their constrains and provide them the ability to access all the data they need from the web, without compromising their trustless nature. Currently, the following data sources can be called using Oraclize:

* URL (any public web API or webpage on the internet)
* WolframAlpha
* Blockchain (blockchain related data like bitcoin difficulty, litecoin hashrate)

<http://www.oraclize.it>

**Constraints**

Now that we have looked at the basics of the solidity programming language and some example code, we have also experienced some constraints and technical difficulties along the way.

*Deterministic approach* – A solidity contract should always be deterministic. It should always give the same result based on the same input. This is needed because when a transaction is validated by different nodes, it should always return the same result. If this is not the case, consensus cannot be possible and the transaction will not be processed.

A random number is easy in any other environment, but when dealing with the blockchain this becomes a notable problem. For any real randomization, the use of an oracle is probably

*Gas costs* – Based on the actions executed in a contract, gas costs will become greater. This forces contracts to be as simple as possible. A developer should be very careful around loops since these can make gas costs explode.

**Example contract**

