2 May 2019

ONBOARDING-TUTORIAL

A SMALL TUTORIAL TO GET STARTED WITH THE PROJECT

To get started right away I have listed some important website and basic tutorials. I also wrote some important concept that might be useful while hands-on with the real project.

1. SOLIDITY INTRODUCTION

Solidity is a high-level programming language, that is used to implement smart contracts. The contracts created using Solidity can be used for multi-signature purposes.

It has Python, C++ and JavaScript influences and is used for Ethereum Virtual Machine (EVM). As a result, it is fairly convenient and easy to grasp for those that are already familiar with the Python, C++ or JavaScript. This provides a good opportunity for newcomers to learn how to work with blockchain applications.

For the purposes of learning, Remix is currently the most optimal way of trying out Solidity

2. SOLIDITY INTRODUCTION RESOURCES

The main resources for using Solidity consist of available integrations, such as IDEs, tools for various purposes, parsers and grammar.

Let's begin with the integrations since they will be required to get started with practicing Solidity

3. SOLIDITY INTRODUCTION INTEGRATIONS

Remix

This browser-based IDE provides an inbuilt compiler as a well as run-time environment without server-side components. This makes it fairly convenient for using and learning Solidity. Click here to have a look on the remix ide.

Solidity plugins

There are also packages that optimize compilers and code editors for Solidity.

Here's a list of these integrations:

IntelliJ IDEA plugin - plugin for IntelliJ IDEA, as well other JetBrains IDEs

Visual Studio Extension - plugin for Microsoft Visual Studio, includes a compiler

Package for SublimeText — Solidity language syntax - provides Solidity syntax highlighting.

4. SOLIDITY EXAMPLE BASIC SMART CONTRACT

A contract in terms of Solidity is a data, that can be referred to as its state, and code, which can be referred to as its functions, collection, that resides on a specific address in the Ethereum blockchain.

Let's try to create a simple contract that creates, sets the value for, and returns a variable called storageData.

Example:

```
pragma solidity ^0.4.0;
contract SampleContract {
    uint storageData;
    function set (uint x) {
        storageData = x;
    }
    function get () constant returns (uint) {
        return storageData;
    }
}
```

The first line of code sets a pragma.

pragma is a keyword, which is used to instruct the compiler how the source code should be treated. These instructions are what we refer to as pragmas.

In this case, the pragma specified that the source code is written for the Solidity version 0.4.0 and above (if it does not break functionality).

The first line inside the contract itself declares an unsigned integer (unsigned 256-bit integer) variable called **storageData**.

To specify that it will be an unsigned variable, we use the keyword **uint** before the name of the new variable.

This integer can be seen as a database slot, which is used by calling the appropriate functions to set and return the value.

In Ethereum's case, this is always the owning contract.

- **Note:** In Solidity, all identifiers are restricted to ASCII character set. The string values, however, can contain UTF-8 characters.
- **Warning:** be careful with Unicode characters since similarly looking characters may have different code points, resulting in them being encoded as different byte arrays.

To modify and return the variable value, you will need to use the functions **set** and **get**. To use either of these, you need to specify the type and name of the variable you want to use between parentheses (in this case it's **uint storageData**).

In the case of **set**, next up is simply calling the name of the variable inside the functions and assigning a value. In the case of **get**, you simply need to write the keyword return before your variable name. The problem is though, that in this case just about anyone could call **set** again and set another value.

5. SOLIDITY IMPORTING LIBRARIES

Importing in Solidity uses statements that very similar to JavaScript past version ES6. It should be noted though, that Solidity does not support export statements. Globally, import statements like these works:

```
import "importedFile";
```

The statement shown above will import all symbol from, and imported into, importedFile, into the global scope. This is different from ES6, but it's backwards-compatible to Solidity. The line shown below will created a symbol called importedSymbol, containing the global symbols from importedFile.

```
import * as importedSymbol from "importedFile";
```

Another line of syntax for Solidity, that is rather convenient:

```
import "importedFile" as importedSymbol;
```

6. SOLIDITY CONTRACT STRUCTURE

This tutorial will shortly describe the structure of a contract written in Solidity. Every contract contains declarations of elements such as functions, function modifiers, state variables, struct and enum types, events as needed for its purpose.

Functions

Functions are blocks of code that are executed when called. Functions have varying visibility to other contracts and can be called both internally and externally.

Example:

```
function MeterReadings () payable {// Script to execute
// ...}
```

Function Modifiers

Function modifiers in Solidity are used in a declarative way to amend function semantics.

Example:

```
address public seller;

    modifier sellerOnly () {// Modifier itself

        require (msg. sender == seller);

    }

    function abort () sellerOnly {// Usage of modifier

        // ...

    }
```

State Variables

State variables are values, permanently stored in the contract's storage.

Example:

```
uint storageData;
```

Struct Types

Struct types are custom defined types, containing a set of variables.

Example:

```
struct Voter { // Struct
        uint mrterID;
        bool MeterCounted;
        address delegateAddress;
        uint meterReadings;
    }
```

Enum Types

Enums' use is to create custom types with finite value sets.

Example:

```
enum contractState {Created, Locked, Inactive} // Enum type
```

Events

Events in Solidity are a convenience interface for EVM logging facilities, making them excellent for keeping track of what is happening with a contract.

Example:

```
event toMeterReadings (uint meterID, uint current_readings);
// Event
    function info () payable {
        // ...
        toMeterReadings (msg. sender, msg.value); // Trigger
event
    }
```

7. SOLIDITY FUNCTION CALLS

In Solidity you can call functions either internally or externally. Only the same contract's functions of the same contract may be called internally, whereas external function calls refer to other contracts

The current contract's functions are possible to call directly, to which we also refer to as calling functions internally, as well as recursively, as shown in this example:

```
Example:

pragma solidity ^0.4.0;

contract cont {

    function func1(uint x) returns (uint returnValue) {

        return func2();

}

    function func2() returns (uint returValue) {

        return func1(7) + func2();

        }

}
```

Those function calls get translated to simple jumps in the EVM. The effect of this is the current memory not being cleared, for example, passing references of memory to internally-called functions is extremely efficient. You can only call functions that are in the same contract internally though.

Solidity Cheat sheet Function Visibility Specifiers

```
function functionName
() <visibility specifier> returns (bool) {return true;}
```

- **public**: externally and internally visible (will create a getter function in case of storage/state variables)
- **private**: visible inside the current contract exclusively
- **external**: visible externally exclusively (for use on functions only) - i.e. may only be message-called (using **this. functionName**)
- **internal**: visible exclusivley internally

Solidity Cheatsheet Modifiers

- **pure** for functions: Does not allow modifying or access of state - not enforced yet.
- **view** for functions: Does not allow modifying of state - not enforced yet.
- **payable** for functions: Allow function to receive Ether via calls.
- **constant** for state variables: Does not allow assignment (except initialization), will not occupy storage slot.

- **constant** for functions: Similar to the **view** modifier.
- **anonymous** for events: No event signature storage as topic.
- **indexed** for event parameters: Store parameter as topic.

This tutorial covers a basic idea about the solidity programming.

References:

1. *https://solidity.readthedocs.io/en/v0.5.8/*
2. *https://bit.ly/2vWbNX4*
3. *https://www.codementor.io/learn/blockchain/solidity-tutorials*
4. *https://bit.ly/2VLwHX6*
5. *www.bitdegree.org*