Solidity Part 1

Introduction

First, you need Remix (https://remix.ethereum.org/), a browser IDE to write and compile Solidity code.

- **How to define the Solidity version compiler ?**

```
// Any compiler version from the 0.8 release (= 0.8.x)
pragma solidity ^0.8.0;

// Greater than version 0.7.6, less than version 0.8.4
pragma solidity >0.7.6 <0.8.4;
```

- **How to define a contract ?**

Use the keyword contract followed by your contract name.

```
contract Score {

    // You will start writing your code here =)

}
```

- **How to write variable in Solidity ?**
  Contract would need some state. We are going to declare a new variable that will hold our score.

```
contract Score {

    uint score = 5;

}
```

Important !

Solidity is a statically typed language. So you always need to **declare the variable type** (here `uint`) before the variable name.

Do not forget to end your declaration statements with a semicolon `;`

`uint` defines an *unsigned integer* of 256 bits by default.
You can also specify the number of bits, by range of 8 bits. Here are some examples below:

Type	Number range
uint8	0 to 255
uint16	0 to 65,535
uint32	0 to 4,294,967,295
uint64	0 to 18,446,744,073,709,551,615
uint128	0 to 2^128
uint256	0 to 2^256

1) Getter and Setter

We need a way to write and retrieve the value of our `score` . We achieve this by creating a **getter** and **setter** functions.

In Solidity, you declare a `function` with the keyword function followed the *function name* (here `getScore()`).

```
contract Score {

    uint score = 5;

    function getScore() returns (uint) {
        return score;
    }

    function setScore(uint new_score) {
        score = new_score;
    }
}
```

Let’s look at both functions in detail.

1.1) Getter function using `return`

Definiton : In Solidity, a **getter** is a function that returns a value.

To return a value from a function (here our `score`), you use the following keywords:

- *In the function definition:* `returns` + variable type returned between parentheses for example (`uint`)
- *In the function body:* `return` followed by what you want to return for example `return score;` or `return 137;`

1.2) Setter function: pass parameters to our function

Definition : In Solidity, a **setter** is a function that modifies the value of a variable (**modifies the state of the contract**). To creates a **setter**, you must specify the **parameters** when you declare your function.

After your function name, specifies between parentheses 1) the **variable type** (`uint`) and 2) the **variable name** (`new_score`)

Compiler Error:

Try entering this code in Remix. We are still not there. The compiler should give you the following error:

```
Syntax Error: No visibility specified. Did you intend to add "public" ?
```

Therefore, we need to specify a visibility for our function. We are going to cover the notion of **visibility** in the next section.

2) Function visibility

2.1) Introduction

To make our functions work, we need to specify their *visibility* in the contract.

Add the keyword `public` after your function name.

```
contract Score {

    uint score = 5;

    function getScore() public returns (uint) {
        return score;
    }

    function setScore(uint new_score) public {
        score = new_score;
    }
}
```

What does the `public` keyword mean ?
There are four types of *visibility* for functions in Solidity : `public` , `private` , `external` and `internal` . The table below explains the difference.

Visibility	Contract itself	Derived Contracts	External Contracts	External Addresses
`public`	✓	✓	✓	✓
`private`	✓			
`internal`	✓	✓		
`external`			✓	✓

Learn More:

- Those keywords are also available for state variables, except for `external` .
- For simplicity, you could add the `public` keyword to the variable. This would automatically create a **getter** for the variable. You would not need to create a **getter** function manually. (see code below)

```
uint score public;
```

Try entering that in Remix. We are still not getting there ! You should receive the following Warning on Remix.

Compiler Warning:

```
Warning: Function state mutability can be restricted to view.
```

2.2) `View vs Pure ?`

- `view` functions can **only read** from the contract storage. They can't modify the contract storage. Usually, you will use `view` for getters.
- `pure` functions can **neither read nor modify** the contract storage. They are only used for *computation* (like mathematical operations).

Because our function `getScore()` only reads from the contract state, it is a `view` function.

```
function getScore() public view returns (uint) {
    return score;
}
```

3) Adding Security with Modifiers

Our contract has a security issue: **Anyone can modify the score.**

Solidity provides a global variable `msg` , that refers to the address that interacts with the contract's functions. The `msg` variables offers two associated fields:

- `msg.sender` : returns the address of the caller of the function.
- `msg.value` : returns the value in **Wei** of the amount of Ether sent to the function.

How to restrict a function to a specific caller ?

We should have a feature that enables only certain addresses to change the score (your address). To achieve this, we will introduce the notion of **modifiers**.

Definition : A `modifier` is a special function that enables us to change the behaviour of functions in Solidity. It is mostly used to automatically check a condition before executing a function.

We will use the following modifier to restrict the function to only the *contract owner*.

```
address owner;

modifier onlyOwner {
    if (msg.sender == owner) {
       _;
    }
}

function setScore(uint new_score) public onlyOwner {
    score = new_score;
}
```

The `modifier` works with the following flow:

1. Check that the address of the caller (`msg.sender`) is equal to `owner` address.
2. If 1) is true, it passes the check. The `_;` will be replaced by the function body where the modifier is attached.

A `modifier` can receive arguments like functions. Here is an example of a modifier that requires the caller to send a specific amount of Ether.

```
modifier Fee(uint fee) {
    if (msg.value == fee) {
        _;
    }
}
```

However, we still haven't defined who the owner is. We will define that in the **constructor**.

4) Constructor

Definition : A **constructor** is a function that is **executed only once** when the contract is deployed on the Ethereum blockchain.

In the code below, we define the contract owner:

```
contract Score {

    address owner;

    constructor() {
        owner = msg.sender;
    }

}
```

Learn More:

Constructors are optional. If there is no constructor, the contract will assume the default constructor, which is equivalent to `constructor () {}`

Warning !

Prior to version 0.4.22, constructors were defined as function with the same name as the contract. This syntax was deprecated and is not allowed in version 0.5.0.

5) Events

Events are only used in Web3 to output some return values. They are a way to show the changes made into a contract.

Events act like a log statement. You declare **Events** in Soldity as follow:

```
// Outside a function
event myEvent(type1, type2, ... );

// Inside a function
emit myEvent(param1, param2, ... );
```

To illustrate, we are going to create an `event` to display the new score set. This `event` will be passed within our `setScore()` function. **Remember that you should pass the score after you have set the new variable.**

```
event Score_set(uint);

function setScore(uint new_score) public onlyOwner {
    score = new_score;
    emit Score_set(new_score);
}
```

You can also use the keyword `indexed` in front of the parameter's types in the `event` definition.

It will create an **index** that will enable to search for events via Web3 in your front-end.

```
event Score_set(uint indexed);
```

Note !

`event` can be used with any functions types (`public` , `private` , `internal` or `external`). However, they are **only visible outside the contract**.

`event` are are not visible internally by Solidity. You cannot read an `event` . So a function cannot read the `event` emitted by another function for instance.

Learn more !

When you call an `event` in Solidity, the arguments passed to it are stored in the transaction's log - a special data structure in the Ethereum blockchain. These logs are associated with the address of the contract, incorporated into the blockchain, and stay there as long as a block is accessible.

6) References Data Types: Mappings

Mappings are another important complex data type used in Solidity. They are useful for association, such as associating an address with a balance or a score. You define a mapping in Solidity as follow:

```
mapping(KeyType => ValueType) mapping_name;
```

You can find below a summary of all the datatypes supported for the key and the value in a mapping.

Type	Key	Value
`int/uint`	✓	✓
`string`	✓	✓
`bytes`	✓	✓
`address`	✓	✓
`struct`	✗	✓
`mapping`	✗	✓
`enums`	✗	✓
`contract`	✗	✓
`fixed-sized array`	✓	✓
`dynamic-size array`	✗	✓
`variable`	✗	✗

You can access the value associated with a key in a mapping by specifing the key name inside square brackets `[]` as follows: `mapping_name[key]` .

Our smart contract will store a mapping of all the user's addresses and their associated score. The function `getUserScore(address _user)` enables to retrieve the score associated to a specific user's address.

```
mapping(address => uint) score_list;

function getUserScore(address user) public view returns (uint) {
    return score_list[user];
}
```

Tips:

you can use the keyword public in front of a mapping name to create automatically a **getter** function in Solidity, as follows:

```
mapping(address => uint) public score_list;
```

Learn More:

In Solidity, mappings do not have a length, and there is no concept of a value associated with a key.

Mappings are virtually initialized in Solidity, such that every possible key exists and is mapped to a value which is the default value for that datatype.

7) Reference Data Types: Arrays

Arrays are also an important part of Solidity. You have two types of arrays (`T` represents the data type and `k` the maximum number of elements):

- **Fixed size** array : `T[k]`
- **Dynamic size** array : `T[]`

```
uint[] all_possible_number;
uint[9] one_digit_number;
```

In Solidity, arrays are ordered numerically. Array indices are zero based. So the index of the 1st element will be **0**. You access an element in an array in the same way than you do for a mapping:

```
uint my_score = score_list[owner];
```

You can also used the following two methods to work with arrays:

`array_name.length` : returns the number of elements the array holds.
`array_name.push(new_element)` : adds a new element at the end of the array.

8) Structs

We can build our own datatypes by combining simpler datatypes together into more complex types using structs.
We use the keyword **struct**, followed by the structure name , then the fields that make up the structure.
For example:

```
struct Funder {
    address addr;
    uint amount;
}
```

Here we have created a datatype called Funder, that is composed of an address and a uint. We can now declare a variable of that type

```
Funder giver;
```

and reference the elements using dot notation

```
giver.addr = address (0xBA7283457B0A138890F21DE2ebCF1AfB9186A2EF);
giver.amount = 2500;
```

The size of the structure has to be finite, this imposes restrictions on the elements that can be included in the struct.

Example of a contract using reference datatypes

```
pragma solidity ^0.8.0;

contract ListExample {

  struct DataStruct {
    address userAddress;
    uint userID;
  }

  DataStruct[] public records;

  function createRecord1(address _userAddress, uint _userID) public returns(uin
    DataStruct memory newRecord;
    newRecord.userAddress = _userAddress;
    newRecord.userID    = _userID;
    return records.push(newRecord)-1;
  }

  function createRecord2(address _userAddress, uint _userID) public returns(uin
    return records.push(DataStruct({userAddress:_userAddress,userID:_userID}))-
  }

  function getRecordCount() public view returns(uint recordCount) {
    return records.length;
  }
}
```

Solidity Exercises - VolcanoCoin part 1

(1) VolcanoCoin contract

Additional Guidance:

Please use the Score contract tutorial as guidance, it has plenty of tips for building your ERC20 contract.
At each point where you change your contract you should re deploy to the JavascriptVM and test your changes.

1. In Remix, create a new file called `VolcanoCoin.sol` .
2. Define the pragma compiler version to `^0.8.0` .
3. Before the pragma version, add a license identifer `// SPDX-License-Identifier: UNLICENSED` .
4. Create a contract called VolcanoCoin.
5. Create a variable to hold the total supply of 10000.
6. Make a public function that returns the total supply.
7. Make a private function that can increase the total supply. Inside the function, add 1000 tokens to the current total supply.
8. We probably want users to be aware of functions in the contract for greater transparency, but to make them all public will create some security risks (e.g. we don’t want users to be able to change the total supply).

   Initialise an `address` variable called `owner` with your address.
9. Next, create a `modifier` which only allows an owner to execute certain functions.
10. Make your change total supply function `public` , but add your modifier so that only the owner can execute it.
11. The contract owner’s address should only be updateable in one place. Create a constructor and within the constructor, store the owner’s address.
12. It would be useful to broadcast a change in the total supply. Create an event that emits the new value whenever the total supply changes. When the supply changes, emit this event.

13. In order to keep track of user balances, we need to associate a user’s address with the balance that they have.
    What is the best data structure to hold this association ?
    Using your choice of data structure, set up a variable called `balances` to keep track of the number of volcano coins that a user has.

14. We want to allow the balances variable to be read from the contract, there are 2 ways to do this.
    What are those ways ?
    Use one of the ways to make your balances variable visible to users of the contract.

15. Now change the constructor, to give all of the total supply to the owner of the contract.

16. Now add a public function called transfer to allow a user to transfer their tokens to another address.

17. This function should have 2 parameters :

- the amount to transfer
- the recipient address.

Why do we not need the sender’s address here ?
What would be the implication of having the sender’s address as a parameter ?

18. Add an `event` to the transfer function to indicate that a transfer has taken place, it should record the amount and the recipient address.

19. We want to keep a record for each user’s transfers. Create a `struct` called Payment that stores the transfer amount and the recipient’s address.

20. We want to reference a payments array to each user sending the payment. Create a `mapping` which returns an array of Payment structs when given this user’s address.

Useful Links and further resources

Official Solidity Documentation (https://docs.soliditylang.org/en/v0.8.6/)
Globally Available Variables (https://docs.soliditylang.org/en/v0.8.6/units-and-global-variables.html)
Open Zeppelin Libraries (https://github.com/OpenZeppelin/openzeppelin-contracts)