Lesson 2

Solidity Review

- ### 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.

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

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  {
    DataStruct memory newRecord;
    newRecord.userAddress = _userAddress;
    newRecord.userID    = _userID;
  }

  function createRecord2(address _userAddress, uint _userID) public  {
     records.push(DataStruct({userAddress:_userAddress,userID:_userID}));
  }

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

Inheritence in Solidity

In object-oriented programming, inheritance is the mechanism of basing an object or class upon another object or class.

An object created through inheritance, a "child object", acquires some or all of the properties and behaviors of the "parent object"

In Solidity we use the *is* keyword to show that the current contract is inheriting from a parent contract, for example here Destructible is the child contract and Owned is the parent contract.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;


contract Owned {
    constructor() { owner = msg.sender; }
    address  owner;
}


// Use `is` to derive from another contract. Derived
// contracts can access all non-private members including
// internal functions and state variables. These cannot be
// accessed externally via `this`, though.
contract Child1 is Owned {
    // The keyword `virtual` means that the function can change
    // its behaviour in derived classes ("overriding").
    function doThings()) virtual public {
        .... ;
    }
}
```

See Solidity Documentation

Contract Components

Constructors

Every contract can be deployed with a `constructor`. It's optional to use and can be useful for initialising the contract's state i.e deploying an ERC20 contract with X tokens available.

The constructor is executed only when the contract is deployed.

Internal functions

Internal functions cannot be called externally. They are only visible in their own contract and its child contracts.

Further datatypes

Boolean

`bool`: The possible values are constants `true` and `false`.

Byte Arrays

Can be fixed size or dynamic
For fixed size : `bytes1, bytes2, bytes3, …, bytes32` are available
For dynamic arrays use : `bytes`

BYTES.CONCAT FUNCTION

A recent change (0.8.4)
You can concatenate a variable number of bytes or bytes1 ... bytes32 using bytes.concat.
The function returns a single bytes memory array
The simplest way to concatenate strings is now
`bytes.concat(bytes(s1), bytes(s2))`
where s1 and s2 are defind as string

string

Dynamically-sized UTF-8-encoded string
string is equal to bytes but does not allow length or index access.

Enums

See documentation
The keyword Enum can be used to create a user defined enumerations, similar to other languages.
For example

```
enum ActionChoices { GoLeft, GoRight, GoStraight, SitStill }
// we can then create variables
ActionChoices choice;
ActionChoices constant defaultChoice = ActionChoices.GoStraight;
```

Storage, memory and calldata

See documentation

STORAGE

Storage data is permanent, forms part of the smart contract's state and can be accessed across all functions. Storage data location is expensive and should be used only if necessary. The `storage` keyword is used to define a variable that can be found in storage location.

MEMORY

Memory data is stored in a temporary location and is only accessible within a function. Memory data is normally used to store data temporarily whilst executing logic within a function. When the execution is completed, the data is discarded. The `memory` keyword is used to define a variable that is stored in memory location.

CALLDATA

Calldata is the location where external values from outside a function into a function are stored. It is a non-modifiable and non-persistent data location. The `calldata` keyword is required to define a variable stored in the calldata location.

The difference between calldata and memory is subtle, calldata variables cannot be changed.
For example :

```
pragma solidity ^0.8.0;

contract Test {

function memoryTest(string memory _exampleString)
public pure
returns (string memory) {
    _exampleString = "example";  // You can modify memory
    string memory newString = _exampleString;
    // You can use memory within a function's logic
    return newString;  // You can return memory
}

function calldataTest(string calldata _exampleString) external
pure returns (string calldata) {
    // cannot modify _exampleString
    // but can return it
    return _exampleString;
}
}
```

Constant and Immutable variables

State variables can be declared as constant or immutable. In both cases, the variables cannot be modified after the contract has been constructed. For constant variables, the value has to be fixed at compile-time, while for immutable, it can still be assigned at construction time.

It is also possible to define constant variables at the file level.

```
// define a constant a file level
uint256 constant X = 32**22 + 8;

contract C {
    string constant TEXT = "abc";
    bytes32 constant MY_HASH = keccak256("abc");
    uint256 immutable decimals;
    uint256 immutable maxBalance;
    address immutable owner = msg.sender;

    constructor(uint256 _decimals, address _reference) {
        decimals = _decimals;
        // Assignments to immutables can even access the environment.
        maxBalance = _reference.balance;
    }
```

Interfaces

Interfaces in Solidity work the same way as in other languages.
The interface specifies the function signatures, but the implementation is specified in child contracts.
Use the *interface* keyword to declare an interface
For example

```
interface DataFeed {
function getData(address token) external returns (uint value);
}
```

Fallback and Receive functions

receive() external payable { … }
Called when the contract receives ether

fallback () external [payable]
Called if a function cannot be found matching the required function signature.
It also handles the case when ether is received but there is no receive function

Checking inputs and dealing with errors

require / assert / revert / try catch

See Error handling

"The **require** function either creates an error without any data or an error of type **Error(string)**.

It should be used to ensure valid conditions that cannot be detected until execution time.

This includes conditions on inputs or return values from calls to external contracts."

Example

```
require(_amount > 0,"Amount must be > 0");
```

The **assert** function creates an error of type **Panic(uint256)**.

Assert should only be used to test for internal errors, and to check invariants.

Properly functioning code should never create a Panic, not even on invalid external input.

Example

```
assert(a>b);
```

The ***revert*** statement acts like a throw statement in other languages and causes the EVM to revert.

The require statement is ofen used in its place.

It can take a string as an error message, or a Error object.

For example

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract VendingMachine {
    address owner;
    error Unauthorized();
    function buy(uint amount) public payable {
        if (amount > msg.value / 2 ether)
            revert("Not enough Ether provided.");
        // Alternative way to do it:
        require(
            amount <= msg.value / 2 ether,
            "Not enough Ether provided."
        );
        // Perform the purchase.
    }
    function withdraw() public {
        if (msg.sender != owner)
            revert Unauthorized();

        payable(msg.sender).transfer(address(this).balance);
```

```
    }
}
```

try / catch statements can be used to catch errors in calls to external contracts.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.1;

interface DataFeed {
function getData(address token) external returns (uint value);
}

contract FeedConsumer {
    DataFeed feed;
    uint errorCount;
    function rate(address token) public
    returns (uint value, bool success) {
        // Permanently disable the mechanism if there are
        // more than 10 errors.
        require(errorCount < 10);
        try feed.getData(token) returns (uint v) {
            return (v, true);
        } catch Error(string memory /*reason*/) {
            // This is executed in case
            // revert was called inside getData
            // and a reason string was provided.
            errorCount++;
            return (0, false);
        } catch Panic(uint /*errorCode*/) {
            // This is executed in case of a panic,
            // i.e. a serious error like division by zero
            // or overflow. The error code can be used
            // to determine the kind of error.
            errorCount++;
            return (0, false);
        } catch (bytes memory /*lowLevelData*/) {
            // This is executed in case revert() was used.
            errorCount++;
            return (0, false);
        }
    }
}
```

Adding Other Contracts and Libraries

When thinking about interacting with other contracts / libraries, it is useful to think of what happens at compile time, and what hapens at runtime.

Compile time

If your contract references another contract or library, whether for inheritence, or for an external function call, the compiler needs to have the relevant code available to it.
You use the `import` statement to make the code available in your compilation file, alternatively you could copy the code into your compilation file it has the same effect.
Sometimes you need to gather all the contracts into one file, for example when getting you contract verified on etherscan. This process is known as flattening and there are plugins in Remix and Truffle to help with this.

If you inherit another contract, for example the Open Zeppelin Ownable contract, on compilation, the functions and variables from the parent contract (except those marked as private) are merged into your contract and become part of the resulting bytecode. From that point on the origin of the functions, are irrelevant.

Run time

There are 2 ways that your contract can interact with other deployed bytecode at run time.

1. External calls
   Your contract can make calls to other contract's functions during a transaction, to do so it needs to have the function signature available (this is checked at compile time) and the other contract's address available.

```
pragma solidity ^0.8.0;

contract InfoFeed {
    uint256 price;
    function info() public view returns (uint256 ret_) {
        return price;
        }
    // other functions
}

contract Consumer {
    InfoFeed feed;

    constructor(InfoFeed _feed){
        feed = _feed;
    }
```

```
    function callFeed() public view returns (uint256) {
        return feed.info();
    }
}
```

2. Using libraries

```
A library is a type of smart contract that has no state, instead their
functions run in the context of your contract.
```

See Documentation

```
For example we could use the Math library from Open Zeppelin
https://github.com/OpenZeppelin/openzeppelin-
contracts/contracts/utils/math/Math.sol

We import it so that the compiler has access to the code
```

```
pragma solidity ^0.8.0;
import "https://github.com/OpenZeppelin/openzeppelin-contracts
/contracts/utils/math/Math.sol";

contract Test {
    using Math for uint256;


     function bigger(uint256 _a, uint256 _b) public pure returns(uint256){
         uint256 big = _a.max(_b);
         return(big);
    }

}
```

The keyword *using* associates a datatype with our library, we can then use a variable of that datatype with the dot notation to call a library function

```
uint256 big = _a.max(_b);
```

You can reference already deployed libraries, at deploy time a linking process takes place which gives your contract the address of the library.
The the library has external or public functions these need to be linked to your contract at deploy time.
If the library functions are internal, they will be inlined into your contract at compile time.

Importing from Github in Remix

See Documentation
You can import directly from github or npm

```
import "https://github.com/OpenZeppelin/openzeppelin-contracts
/contracts/access/Ownable.sol";
```

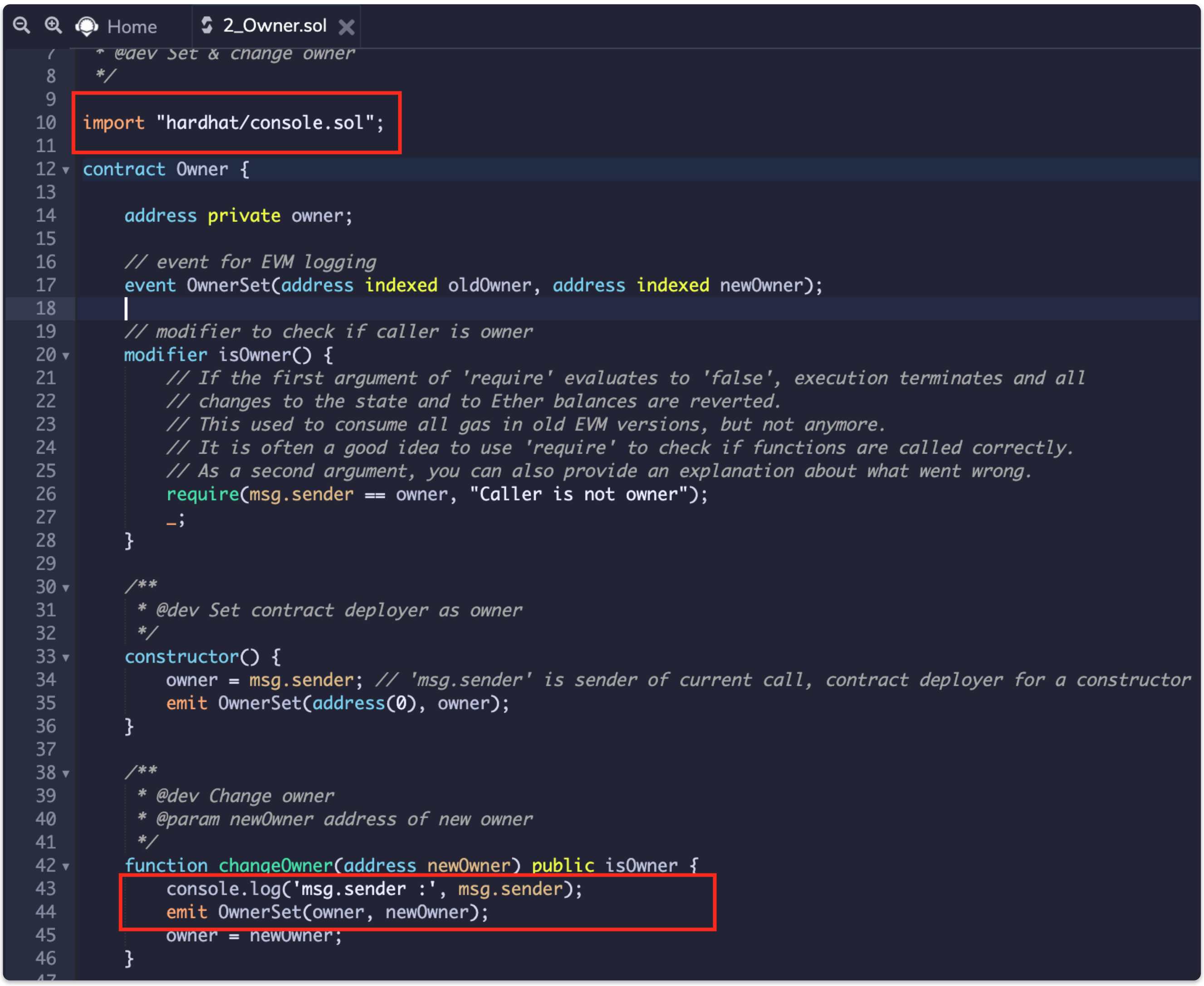
or

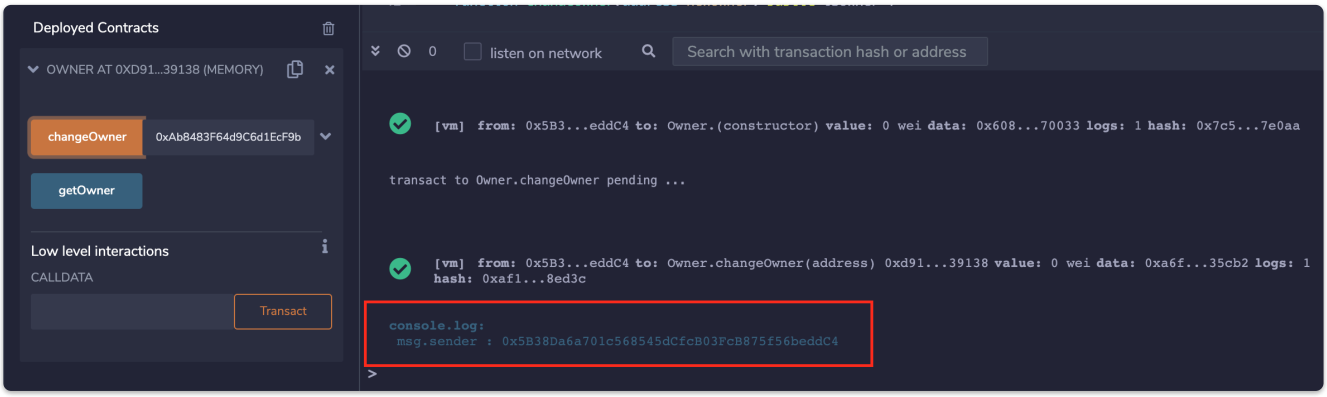
```
import "@openzeppelin/contracts@4.2.0/token/ERC20/ERC20.sol";
```

Logging in Remix

Recently announced
https://remix-ide.readthedocs.io/en/latest/hardhat_console.html

Remix IDE supports hardhat console library while using `JavaScript VM`. It can be used while making a transaction or running unit tests.
To try it out, you need to put an import statement and use `console.log` to print the value as shown in image.





Development Tools

The main tools used are

Remix (Browser or Desktop)

See Documentation

Hardhat

Documentation
Useful boilerplate code

Truffle

Documentation
Truffle Boxes

Forking from Mainnet

Forking from mainnet

The easiest way to try this feature is to start a node from the command line:

```
npx hardhat node --fork https://eth-mainnet.alchemyapi.io/v2/<key>
```

You can also configure Hardhat Network to always do this:

```js
networks: {
  hardhat: {
    forking: {
      url: "https://eth-mainnet.alchemyapi.io/v2/<key>",
    }
  }
}
```

A useful template you might like to try is
Solidity Template

COVERAGE REPORTS

all files contracts/

100% Statements 5/5 **100%** Branches 0/0 **75%** Functions 3/4 **83.33%** Lines 5/6

File		Statements		Branches		Functions		Lines	
Greeter.sol		100%	5/5	100%	0/0	75%	3/4	83.33%	5/6

all files / contracts/ Greeter.sol

100% Statements 5/5 **100%** Branches 0/0 **75%** Functions 3/4 **83.33%** Lines 5/6

```solidity
// SPDX-License-Identifier: MIT
pragma solidity >=0.8.4;

import "hardhat/console.sol";

error GreeterError();

contract Greeter {
    string public greeting;

    constructor(string memory _greeting) {
        console.log("Deploying a Greeter with greeting:", _greeting);
        greeting = _greeting;
    }

    function greet() public view returns (string memory) {
        return greeting;
    }

    function setGreeting(string memory _greeting) public {
        console.log("Changing greeting from '%s' to '%s'", greeting, _greeting);
        greeting = _greeting;
    }

    function throwError() external pure {
        revert GreeterError();
    }
}
```

GAS REPORT

```
    ✓ should return the new greeting once it's changed

·----------------------------|---------------------------|-------------|-----------------------------·
|    Solc version: 0.8.9     ·  Optimizer enabled: true  ·  Runs: 800  ·  Block limit: 30000000 gas  │
·····························|···························|·············|······························
|  Methods                                                                                           │
·············|···············|·············|·············|·············|···············|··············
|  Contract  ·  Method       ·  Min        ·  Max        ·  Avg        ·  # calls      ·  usd (avg)  │
·············|···············|·············|·············|·············|···············|··············
|  Greeter   ·  setGreeting  ·          -  ·          -  ·      34489  ·            1  ·          -  │
·············|···············|·············|·············|·············|···············|··············
|  Deployments               ·                                         ·  % of limit   ·             │
·····························|·············|·············|·············|···············|··············
|  Greeter                   ·          -  ·          -  ·     428178  ·        1.4 %  ·          -  │
·----------------------------|-------------|-------------|-------------|---------------|-------------·

  1 passing (2s)
```

References

Solidity Documentation

Libraries