Solidity and Remix Guide

❖ SOLIDITY:

Solidity is a contract-oriented, high-level programming language for implementing smart contracts. It is highly influenced by C++, Python, and JavaScript and is designed to target the Ethereum Virtual Machine (EVM). It is an object-oriented language that provides human-readable code. It is a static type language i.e., every data, and variable contains its type before storing it.

Since Solidity targets EVM, it is essential to learn about **Ethereum**. It is an open-source platform that uses blockchain technology for building and deploying decentralized applications.

A crypto token, **Ether**, fuels the Ethereum network. The application developers rely on ethers for making payments for transactions services and fees on the Ethereum network.

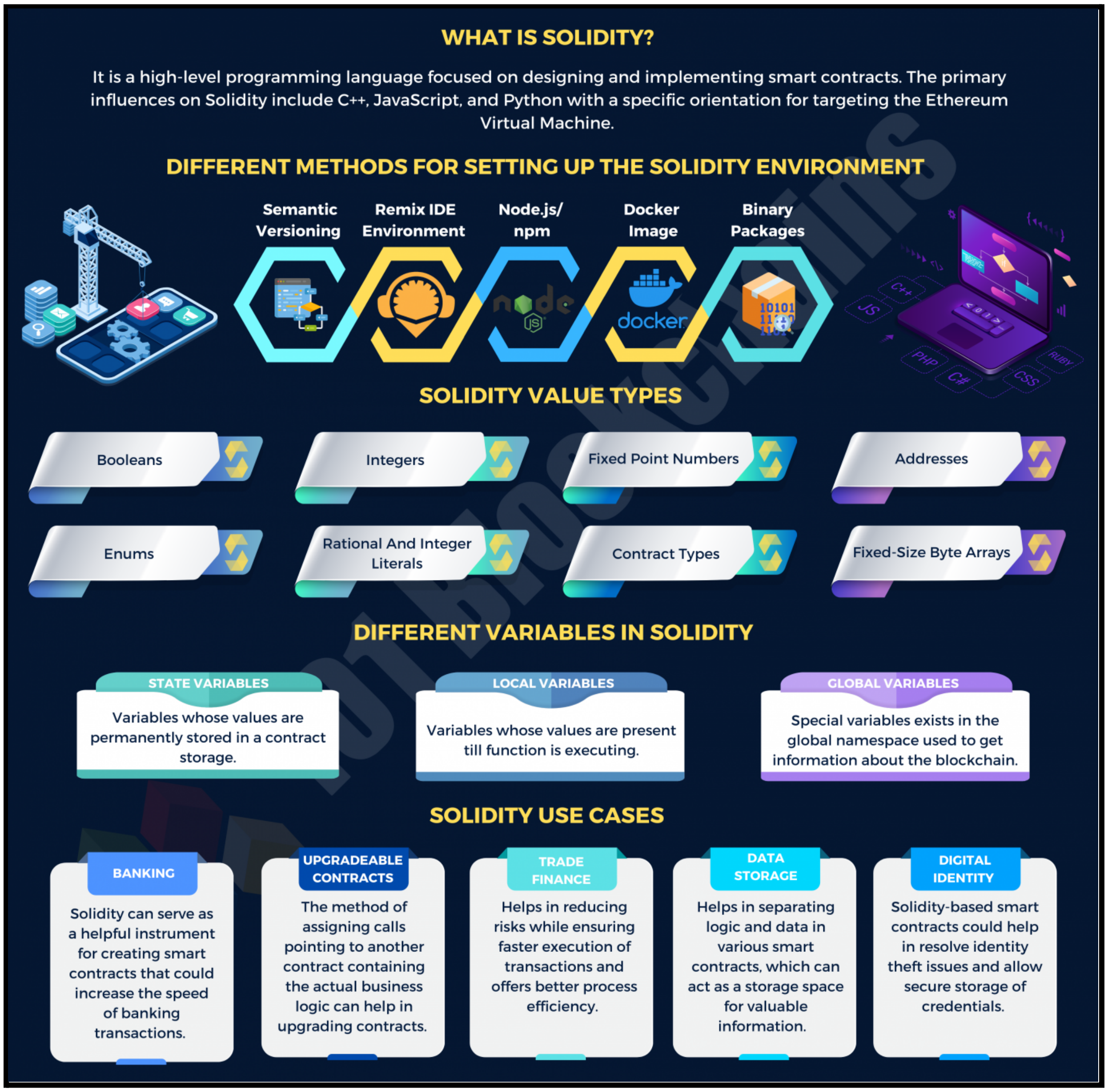
Ethereum **Gas** refers to the fee required to successfully conduct a transaction or execute the contract on the Ethereum blockchain platform. It is used to allocate resources of EVM so that decentralized applications like smart contracts can self execute in a secure way.

Ethereum Virtual Machine(EVM) facilitates a runtime environment for smart contracts execution. It offers security and a facility for executing untrusted code through an international network consisting of public nodes. It prevents Denial-of-service attacks and ensures that a specific program does not have access to the states of each other.

So we discussed that Solidity is used to implement smart contracts. What are smart contracts?

Smart contracts are computer protocols that intend to digitally facilitate, verify, and enforce the negotiation or performance of the contract. They basically act as a vending machine that doesn’t require a third party for the transaction. These transactions are trackable and irreversible. Smart contracts use pragmas as common instructions to help compile and in understanding the ideal approaches for treating the source code.

Attached below is an overview diagram from 101blockchains.com



WHAT IS SOLIDITY?

It is a high-level programming language focused on designing and implementing smart contracts. The primary influences on Solidity include C++, JavaScript, and Python with a specific orientation for targeting the Ethereum Virtual Machine.

DIFFERENT METHODS FOR SETTING UP THE SOLIDITY ENVIRONMENT

- Semantic Versioning
- Remix IDE Environment
- Node.js/ npm
- Docker Image
- Binary Packages

SOLIDITY VALUE TYPES

- Booleans
- Integers
- Fixed Point Numbers
- Addresses
- Enums
- Rational And Integer Literals
- Contract Types
- Fixed-Size Byte Arrays

DIFFERENT VARIABLES IN SOLIDITY

STATE VARIABLES

Variables whose values are permanently stored in a contract storage.

LOCAL VARIABLES

Variables whose values are present till function is executing.

GLOBAL VARIABLES

Special variables exists in the global namespace used to get information about the blockchain.

SOLIDITY USE CASES

BANKING

Solidity can serve as a helpful instrument for creating smart contracts that could increase the speed of banking transactions.

UPGRADEABLE CONTRACTS

The method of assigning calls pointing to another contract containing the actual business logic can help in upgrading contracts.

TRADE FINANCE

Helps in reducing risks while ensuring faster execution of transactions and offers better process efficiency.

DATA STORAGE

Helps in separating logic and data in various smart contracts, which can act as a storage space for valuable information.

DIGITAL IDENTITY

Solidity-based smart contracts could help in resolve identity theft issues and allow secure storage of credentials.

❖ REMIX IDE:

To set up a solidity environment we use remix IDE. It provides an integrated environment to code, run and deploy the smart contracts in a user-friendly way. This platform doesn't require any additional installations. It fosters a fast development cycle and has a rich set of plugins with intuitive GUIs. It is written in JavaScript.

The home page of the remix allows us to select all the featured plugins: **solidity, starknet,** and **learneth** along with some important links for the documentation.

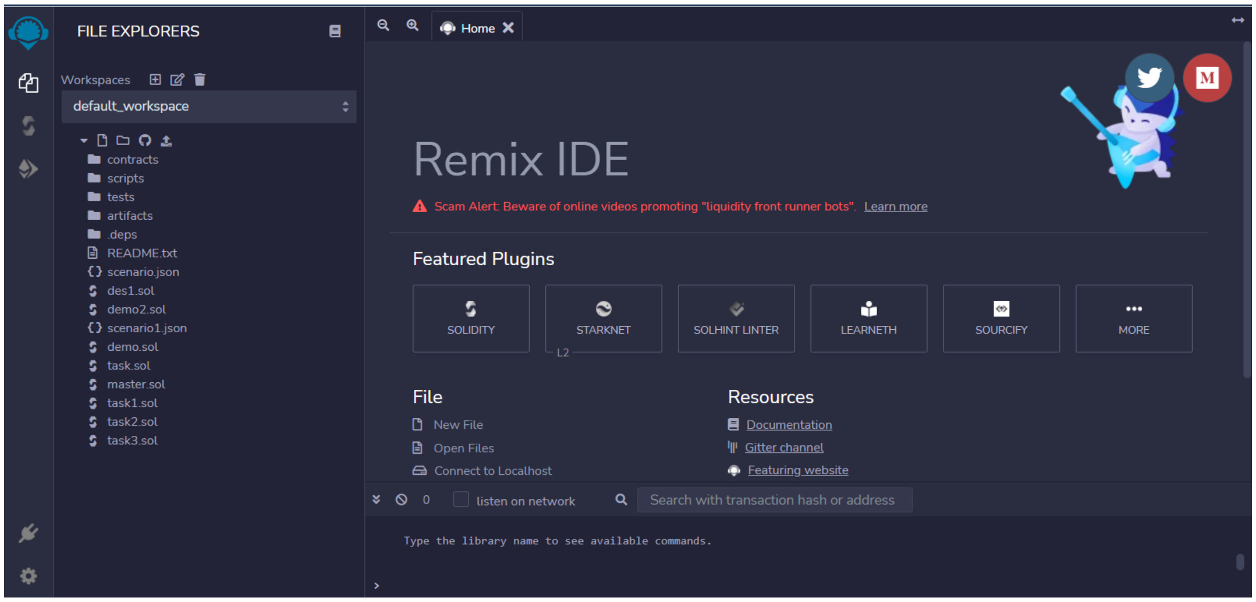
The features of Remix IDE include:

1. **File storage:** The workspace files of the IDE are stored in the browser's local history. To store it in the PC's storage, the Remix desktop version can be used.
2. **Plugin Manager:** The plugin manager allows controlling the active and inactive plugins. The permissions button allows us to check permissions given to different plugins.
3. **Github access token:** This access token is used to publish Gist and retrieve Github contents.
4. **Solidity editor:** The remix editor auto compiles the code when a file is modified. It also provides syntax highlighting mapped to solidity keywords. All the compilation warnings and errors are displayed in the gutter
5. **Terminal:** It integrates a web3 object and a JavaScript interpreter. It displays important actions made while interacting with Remix IDE (i.e. sending a new transaction). It displays the transactions that are mined in the current context. We can also run scripts by inputting them at the bottom.
6. **Compiler:** In the compiler, all the scripts written in the editor are compiled against a specific Ethereum hard fork. Compilation Errors and Warning are displayed below the contract section. At each compilation, the static analysis tab builds a report.
7. **Deploy and Run:** After the compilation, one can run the contracts. It involves various specifications-
    a. **Account:** It is the list of accounts associated with the current environment (and their associated balances).
    b. **Gas limit:** This sets the maximum amount of gas that will be allowed for all the transactions created in Remix.
    c. **Value:** This sets the amount of ETH, WEI, GWEI, etc that is sent to a contract or a payable function.
    d. **Deploy:** This sends a transaction that deploys the selected contract. When the transaction is mined the newly created instance will be added. If the contract's

constructor function has parameters then you need to specify them. There is an input box, where based on the data type of the variable one can input the parameters of the function.

e. **At address:** It is used to access the contract that has been already deployed and it accesses the contract without any gas. To use AtAddress, you need to have the source code or ABI of the deployed contract in the active tab of the editor.

f. **Recorder:** It records the transactions in the JSON file.



Scam Alert: Beware of online videos promoting "liquidity front runner bots". Learn more

```
Type the library name to see available commands.
```

❖ UNDERSTANDING SOLIDITY LANGUAGE:

To get a clear understanding of the solidity of language syntax, let's understand a few major concepts.

1. Basic syntax

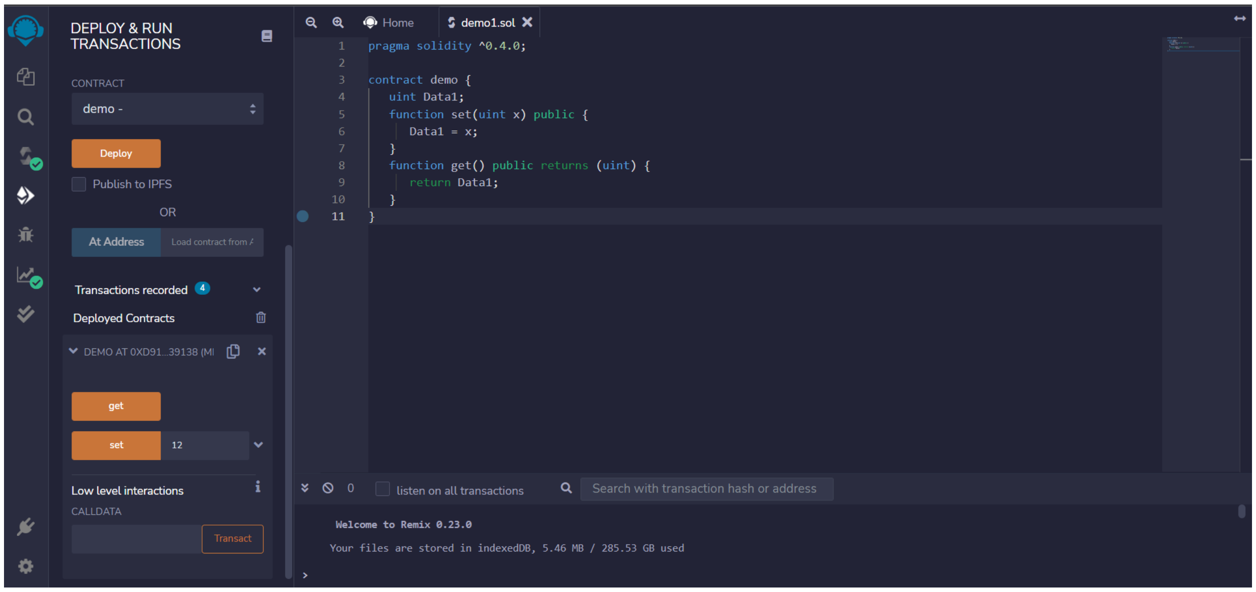
Here, we will learn about the basic syntax used in all the solidity programs.

The first one is the declaration of the version of the solidity that we want to use. This is done in the first line of the code itself as shown below:

pragma solidity >=0.4.0 <0.6.0;
pragma solidity 0.4.0;
pragma solidity ^0.4.0;

As shown above, we can declare the version as a range as well as a single version also. If we use a carrot "^" before the version declaration, it will compile for all the versions 0.4.x (where x>0) but nothing other than that. The pragma directive tells that the source code is written in which version. It is always local to the source file.

Contracts are similar to the classes of object-oriented programming languages. They are a collection of functions and data that resides at a specific address on the Ethereum blockchain.



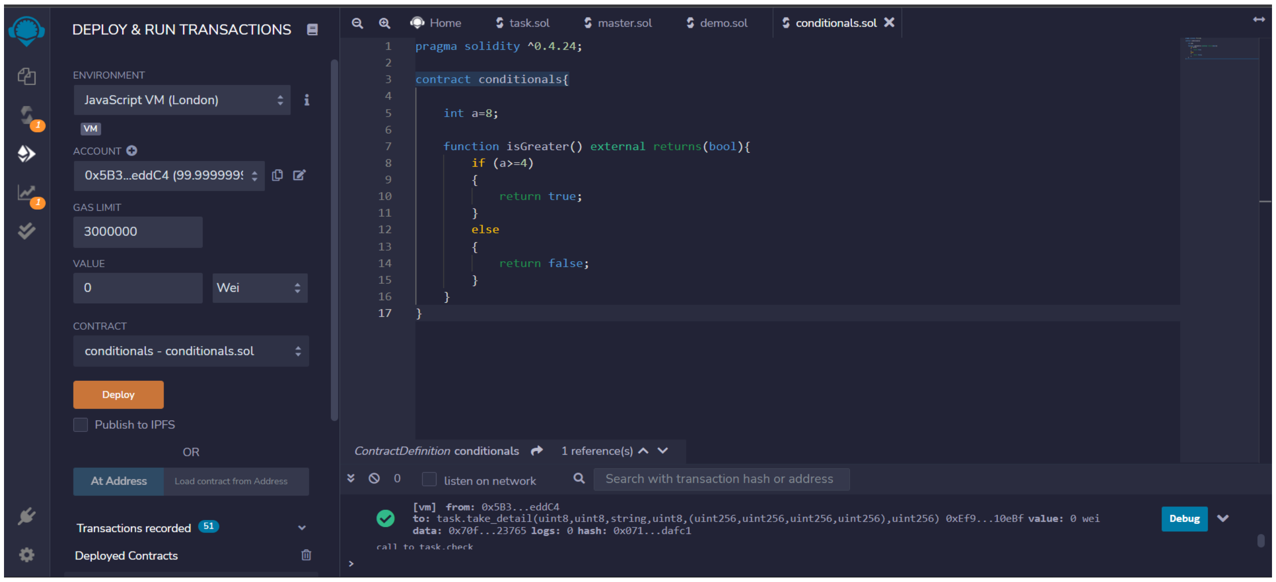
```
pragma solidity ^0.4.0;

contract demo {
    uint Data1;
    function set(uint x) public {
        Data1 = x;
    }
    function get() public returns (uint) {
        return Data1;
    }
}
```

```
Welcome to Remix 0.23.0
Your files are stored in indexedDB, 5.46 MB / 285.53 GB used
```

2. Conditional statements and loops

Here, we will learn about the basic control structures used in Solidity.

a. **If**: The if statement is the fundamental control statement that allows Solidity to make decisions and execute statements conditionally. **Else** is used with if to execute the statement in a more controlled way.



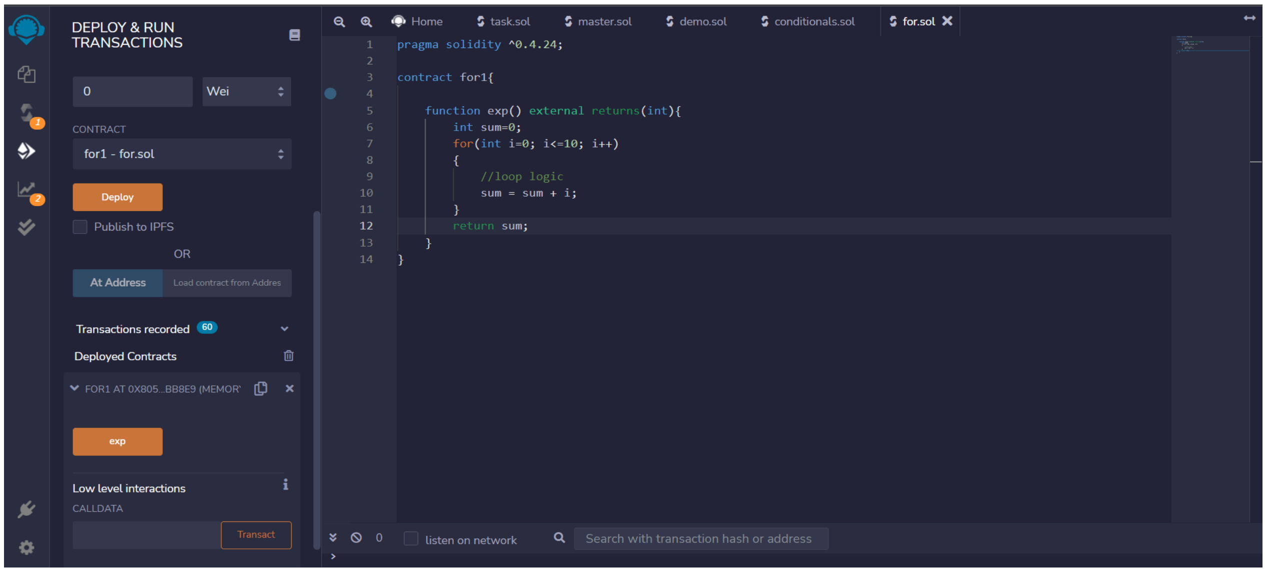
ENVIRONMENT: JavaScript VM (London)
ACCOUNT: 0x5B3...eddC4 (99.999999!
GAS LIMIT: 3000000
VALUE: 0 Wei
CONTRACT: conditionals - conditionals.sol

```
pragma solidity ^0.4.24;

contract conditionals{

    int a=8;

    function isGreater() external returns(bool){
        if (a>=4)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
}
```

ContractDefinition conditionals 1 reference(s)

```
[vm] from: 0x5B3...eddC4
to: task.take_detail(uint8,uint8,string,uint8,(uint256,uint256,uint256,uint256),uint256) 0xEf9...10eBf value: 0 wei
data: 0x70f...23765 logs: 0 hash: 0x071...dafc1
call to task.check
```

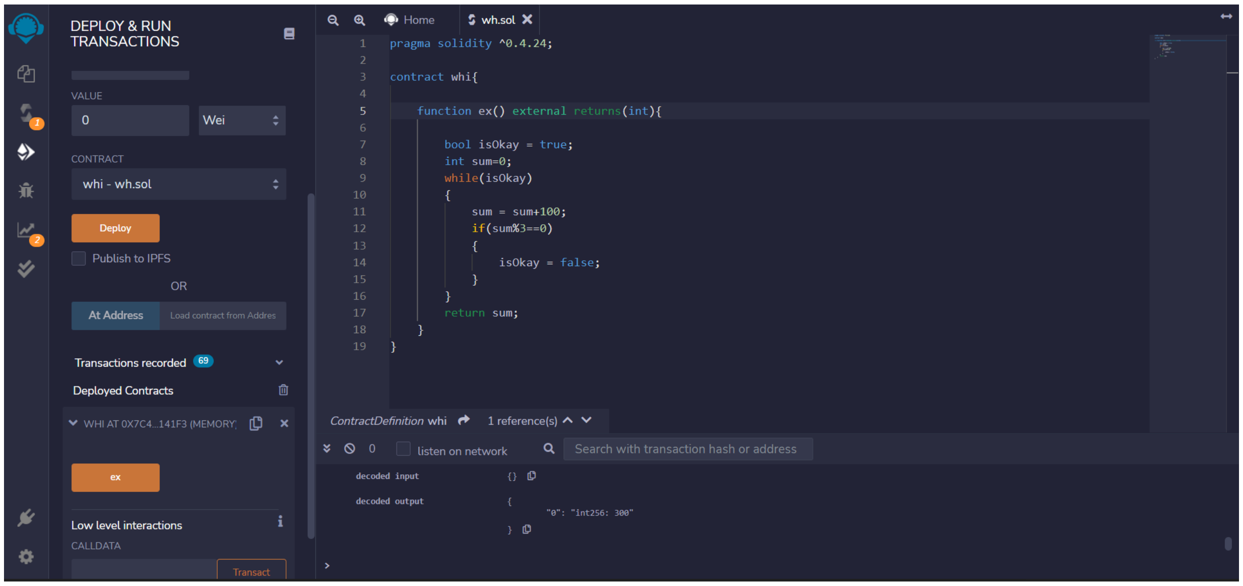
b. **For**: It is the most compact form of looping in Solidity that includes loop initialization, test statement and iteration statement separated by a semicolon.



VALUE: 0 Wei
CONTRACT: for1 - for.sol

```
pragma solidity ^0.4.24;

contract for1{

    function exp() external returns(int){
        int sum=0;
        for(int i=0; i<=10; i++)
        {
            //loop logic
            sum = sum + i;
        }
        return sum;
    }
}
```

c. **While**: It is the most basic loop. Its purpose is to repeatedly execute a statement or code block as long as an expression is true. Once the expression becomes false, the loop terminates.



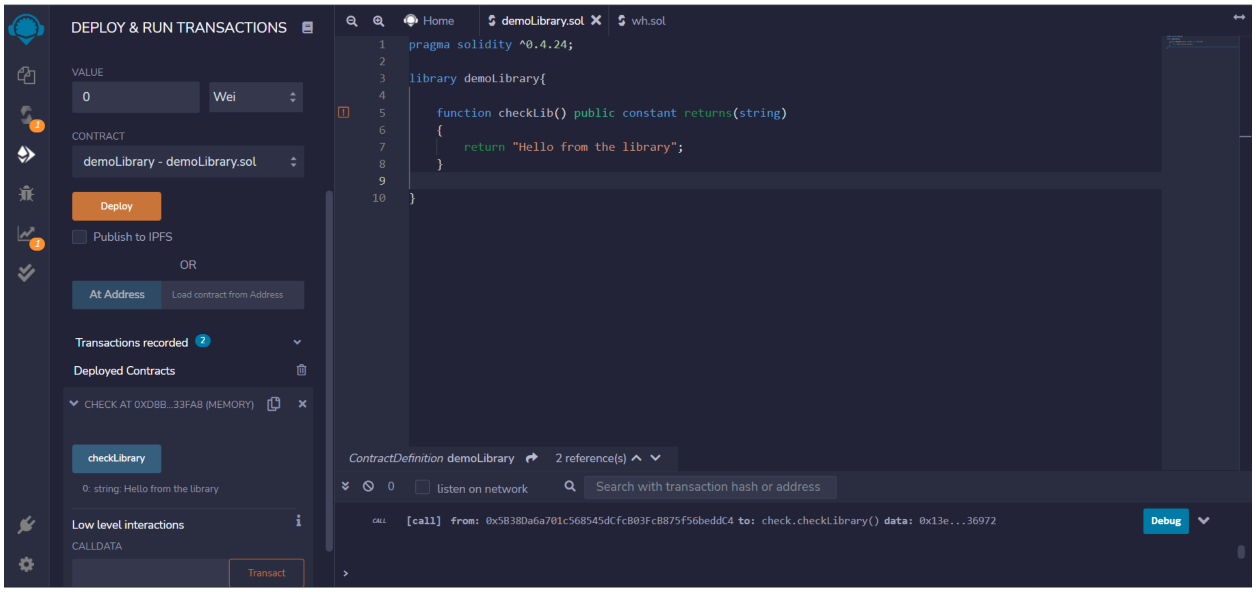
```
pragma solidity ^0.4.24;

contract whi{

    function ex() external returns(int){

        bool isOkay = true;
        int sum=0;
        while(isOkay)
        {
            sum = sum+100;
            if(sum%3==0)
            {
                isOkay = false;
            }
        }
        return sum;
    }
}
```

```
decoded input       {}
decoded output      {
                        "0": "int256: 300"
                    }
```

Note: We can use a **break** statement to end the loop and the **continue** statement to jump to the next loop iteration.

3. Imports and libraries

To import a user-defined library, we first need to create a library along with its functions that we want to use in other solidity files, as shown below in the first image.
To call the library, we will use the import statement as shown in the second image below.
To access the library functions, we will use (library name).(function name) syntax.



VALUE: 0 Wei

CONTRACT: demoLibrary - demoLibrary.sol

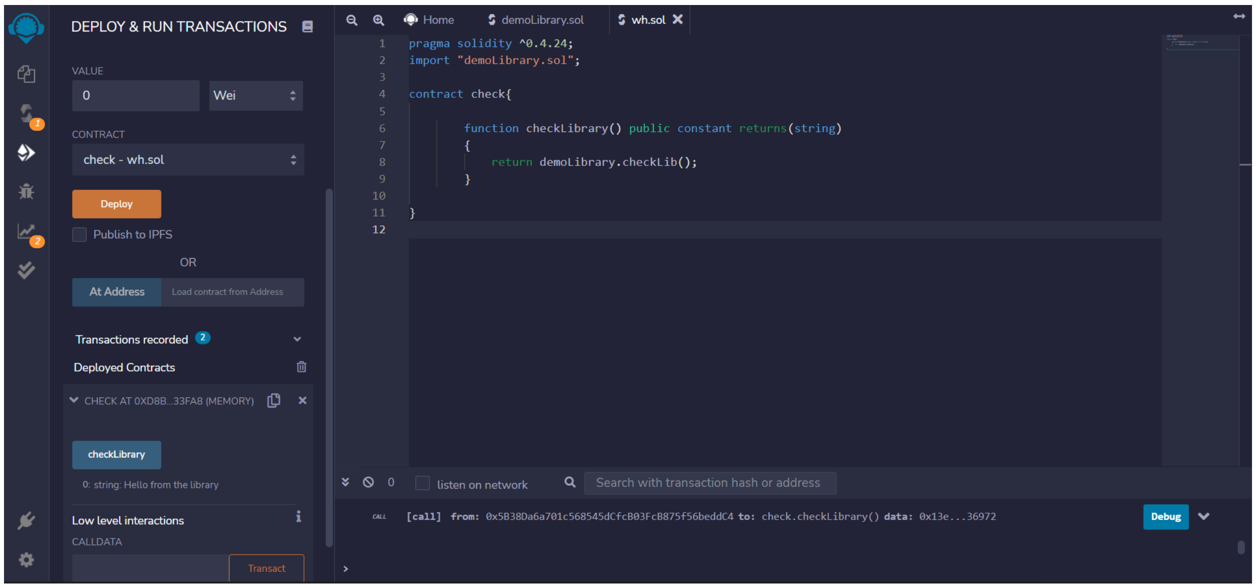
CHECK AT 0XD8B...33FA8 (MEMORY)

0: string: Hello from the library

```
pragma solidity ^0.4.24;

library demoLibrary{

    function checkLib() public constant returns(string)
    {
        return "Hello from the library";
    }

}
```

ContractDefinition demoLibrary 2 reference(s)

```
[call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: check.checkLibrary() data: 0x13e...36972
```



VALUE: 0 Wei

CONTRACT: check - wh.sol

CHECK AT 0XD8B...33FA8 (MEMORY)

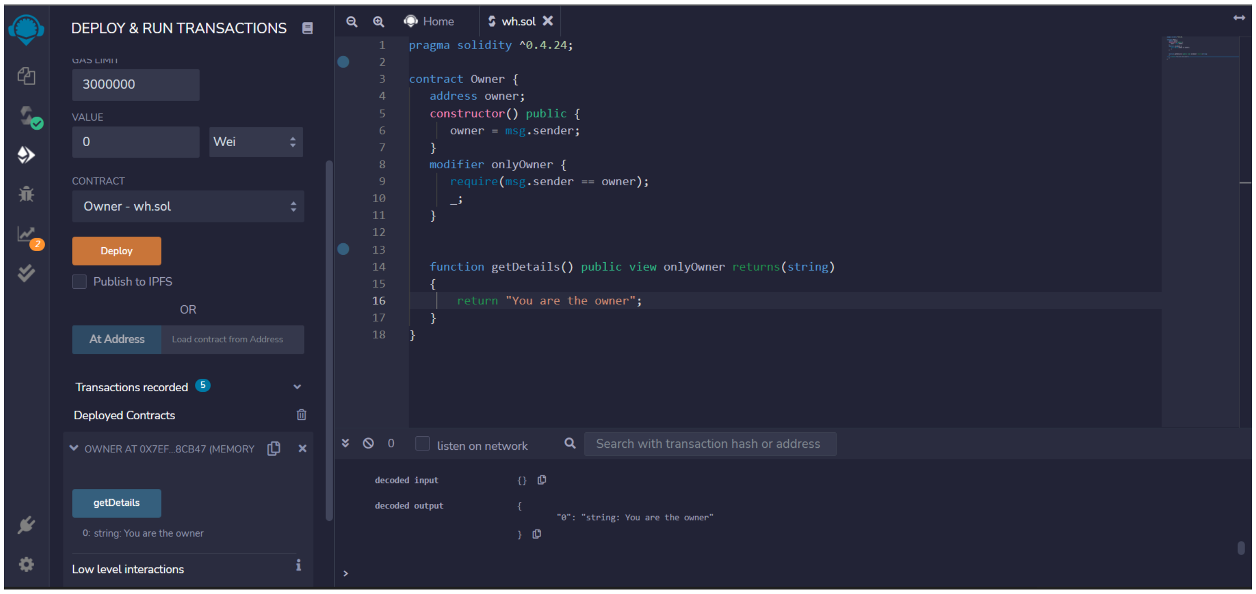
0: string: Hello from the library

```
pragma solidity ^0.4.24;
import "demoLibrary.sol";

contract check{

    function checkLibrary() public constant returns(string)
    {
        return demoLibrary.checkLib();
    }

}
```

```
[call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: check.checkLibrary() data: 0x13e...36972
```

4. Modifiers

Function Modifiers are used to modify the behavior of a function. The function body is inserted where the special symbol "_;" appears in the definition of a modifier. So, if the condition of the modifier is satisfied while calling this function, the function is executed, and otherwise, an exception is thrown.
So, in the below given example, if the owner does the access, only then the function getDetails() will return the mentioned string output.



```
GAS LIMIT  3000000
VALUE      0   Wei
CONTRACT   Owner - wh.sol
0: string: You are the owner
```

```
pragma solidity ^0.4.24;

contract Owner {
    address owner;
    constructor() public {
        owner = msg.sender;
    }
    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }


    function getDetails() public view onlyOwner returns(string)
    {
        return "You are the owner";
    }
}
```

```
decoded input          {} 
decoded output         {
                            "0": "string: You are the owner"
                       }
```

Some of the function modifier keywords are discussed below:

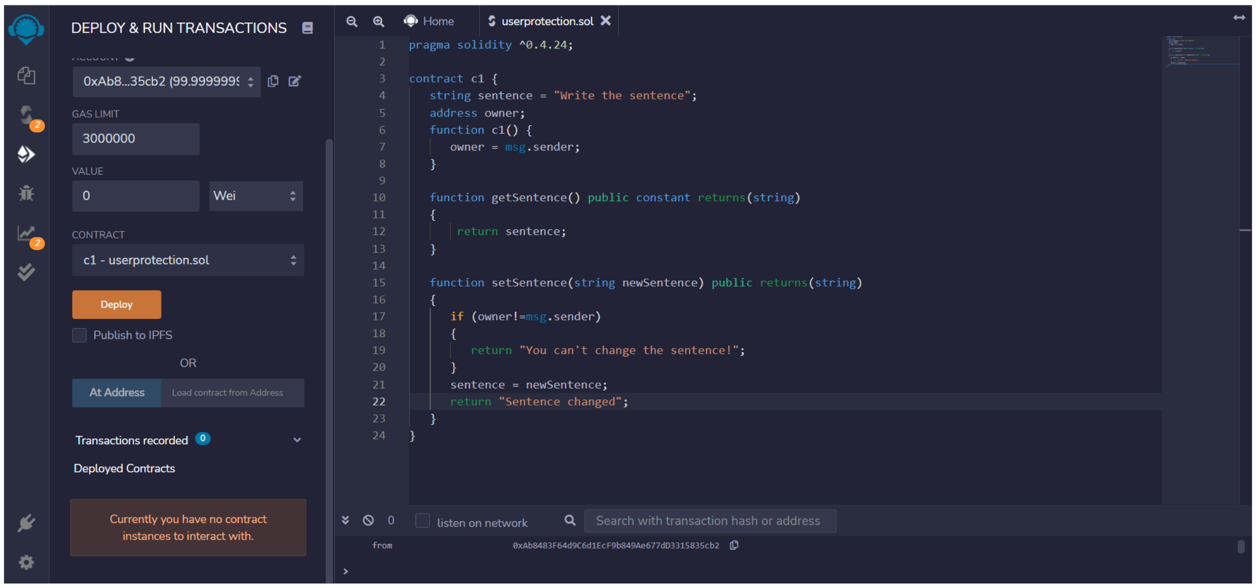
a. **View**(or “constant” in older version): This makes the function read-only. This means that one cannot change the function with the view modifier.
b. **Pure**: It is also a read-only function but instead of returning a value, it does a computation.

5. Function visibility

a. **Private:** This function can be called from within the contract only. It is the most restricted type of visibility.
b. **External:** Only call the function from outside the smart contract.
c. **Public:** Call the function from outside and inside of the smart contract.
d. **Internal:** Similar to protected in object-oriented programming. Internal functions or variables can be used by the same contract or derived contracts.

6. Creating user protection

To create user protection, we first declare the owner variable and store the owner's address in that using the function with the same name as the contract. The same-named function will be executed exactly once, and so the address will be stored in the variable owner.



```
pragma solidity ^0.4.24;

contract c1 {
    string sentence = "Write the sentence";
    address owner;
    function c1() {
        owner = msg.sender;
    }

    function getSentence() public constant returns(string)
    {
        return sentence;
    }

    function setSentence(string newSentence) public returns(string)
    {
        if (owner!=msg.sender)
        {
            return "You can't change the sentence!";
        }
        sentence = newSentence;
        return "Sentence changed";
    }
}
```

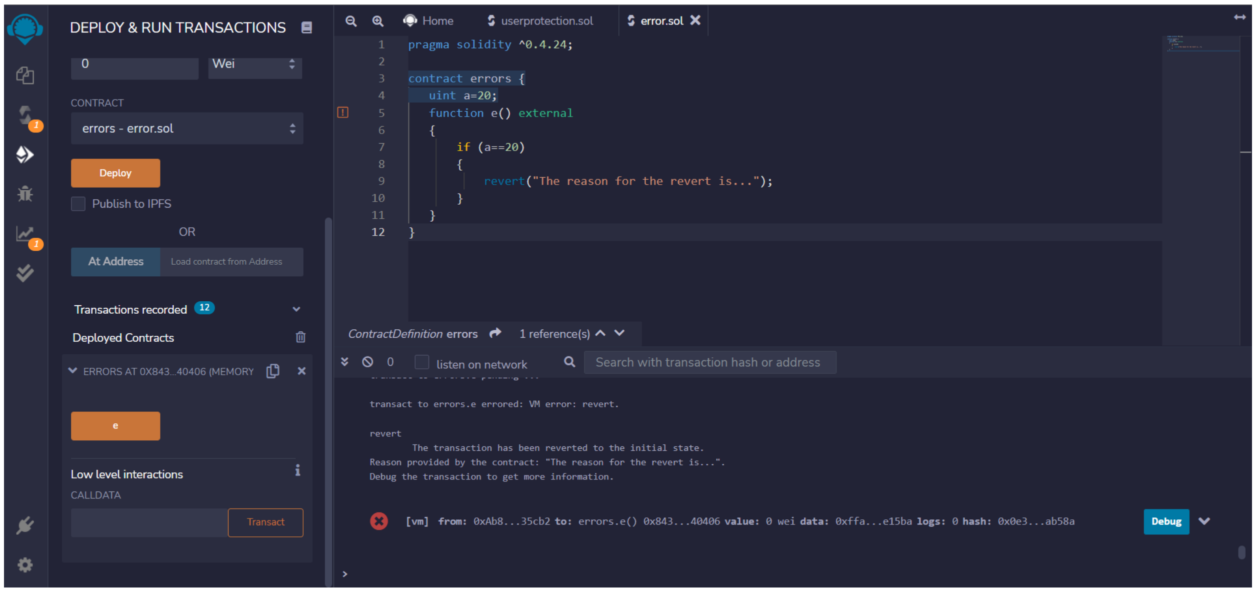
```
from    0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2
```

7. Error Handling

After the error, the execution of the function will stop.

a. **Throw:** Used in older versions of Solidity for error handling; discontinued from version 5.0.

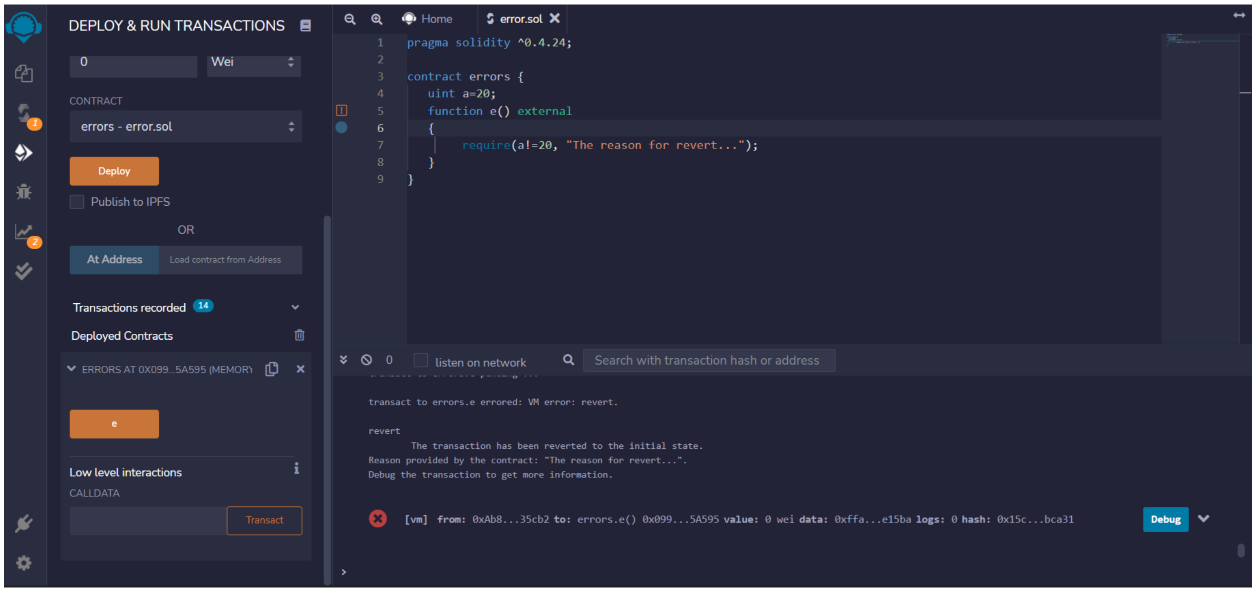
b. **revert():** This method aborts the execution and reverts any changes done to the state.



```
pragma solidity ^0.4.24;

contract errors {
    uint a=20;
    function e() external
    {
        if (a==20)
        {
            revert("The reason for the revert is...");
        }
    }
}
```

```
transact to errors.e errored: VM error: revert.

revert
        The transaction has been reverted to the initial state.
Reason provided by the contract: "The reason for the revert is...".
Debug the transaction to get more information.

[vm] from: 0xAb8...35cb2 to: errors.e() 0x843...40406 value: 0 wei data: 0xffa...e15ba logs: 0 hash: 0x0e3...ab58a
```

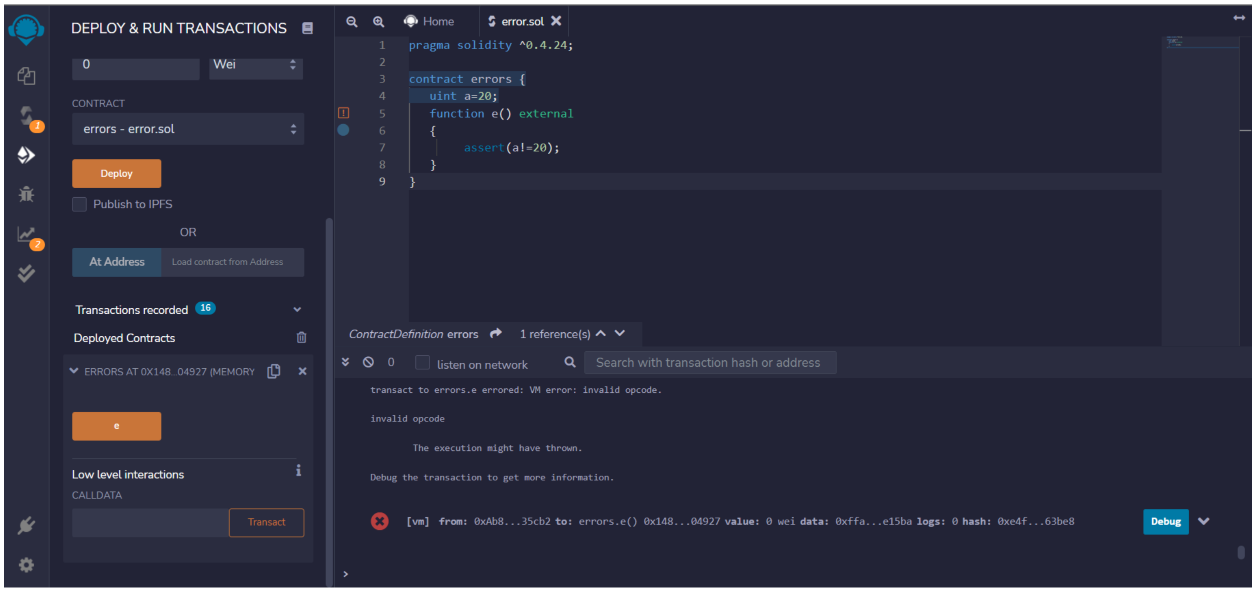
c. **require():** In case the condition is not met, this method call reverts to its original state. - This method is used for errors in inputs or external components. It provides an option to provide a custom message.



```
pragma solidity ^0.4.24;

contract errors {
    uint a=20;
    function e() external
    {
        require(a!=20, "The reason for revert...");
    }
}
```

```
transact to errors.e errored: VM error: revert.

revert
        The transaction has been reverted to the initial state.
Reason provided by the contract: "The reason for revert...".
Debug the transaction to get more information.

[vm] from: 0xAb8...35cb2 to: errors.e() 0x099...5A595 value: 0 wei data: 0xffa...e15ba logs: 0 hash: 0x15c...bca31
```

d. **Assert():** If the condition is not met, this method call causes an invalid opcode and any changes to the state get reverted. This method is to be used for internal errors.



```
pragma solidity ^0.4.24;

contract errors {
  uint a=20;
  function e() external
  {
       assert(a!=20);
  }
}
```

```
transact to errors.e errored: VM error: invalid opcode.

invalid opcode

        The execution might have thrown.

Debug the transaction to get more information.

[vm] from: 0xAb8...35cb2 to: errors.e() 0x148...04927 value: 0 wei data: 0xffa...e15ba logs: 0 hash: 0xe4f...63be8
```

8. Mappings

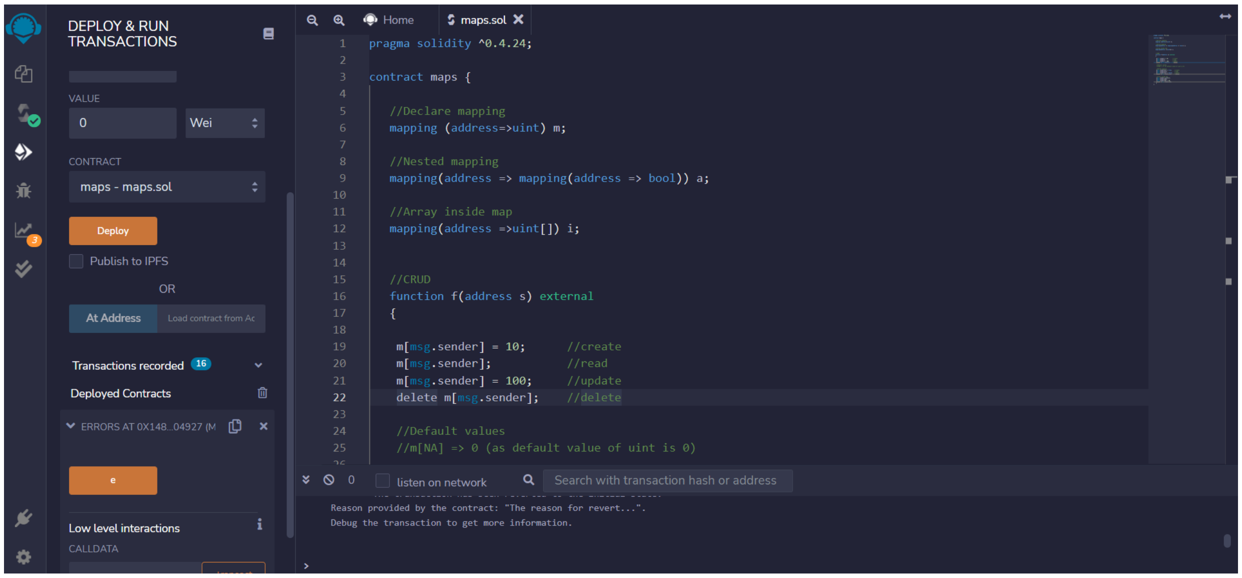
Mapping is a reference type as arrays and structs. Below is the syntax to declare a mapping type.

mapping(_KeyType => _ValueType)

Where,

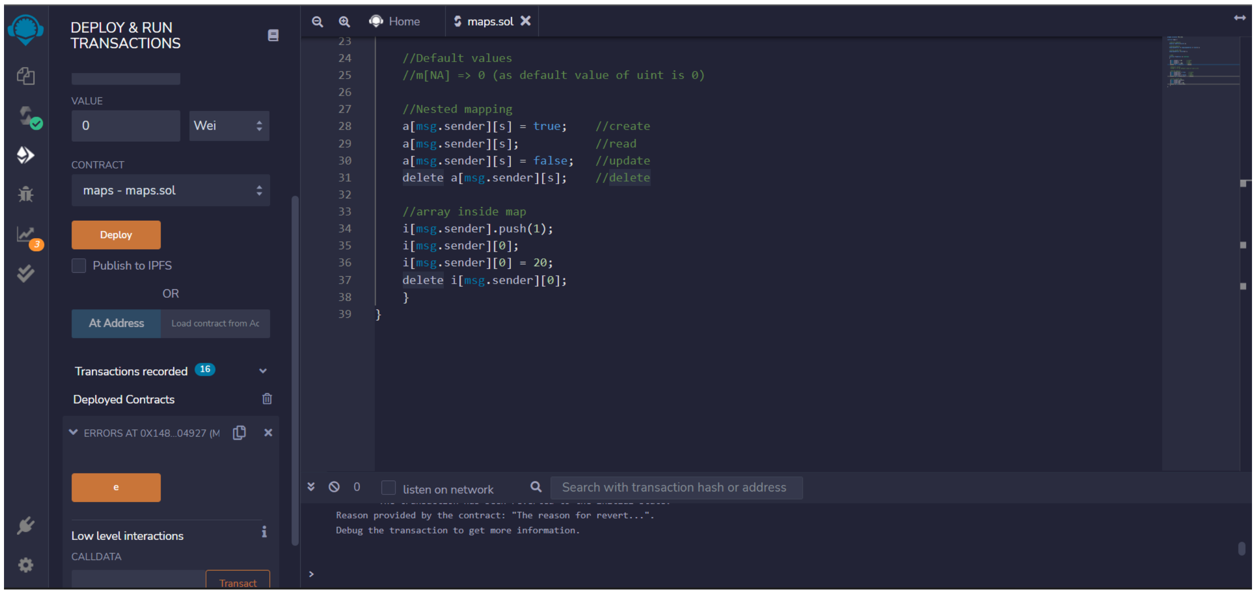
_KeyType – can be any built-in type plus bytes and string. No reference type or complex objects are allowed.

_ValueType – can be any type.



```
pragma solidity ^0.4.24;

contract maps {

  //Declare mapping
  mapping (address=>uint) m;

  //Nested mapping
  mapping(address => mapping(address => bool)) a;

  //Array inside map
  mapping(address =>uint[]) i;


  //CRUD
  function f(address s) external
  {

   m[msg.sender] = 10;      //create
   m[msg.sender];           //read
   m[msg.sender] = 100;     //update
   delete m[msg.sender];    //delete

   //Default values
   //m[NA] => 0 (as default value of uint is 0)
```

```
Reason provided by the contract: "The reason for revert...".
Debug the transaction to get more information.
```



```

   //Default values
   //m[NA] => 0 (as default value of uint is 0)

   //Nested mapping
   a[msg.sender][s] = true;    //create
   a[msg.sender][s];           //read
   a[msg.sender][s] = false;   //update
   delete a[msg.sender][s];    //delete

   //array inside map
   i[msg.sender].push(1);
   i[msg.sender][0];
   i[msg.sender][0] = 20;
   delete i[msg.sender][0];
   }
}
```

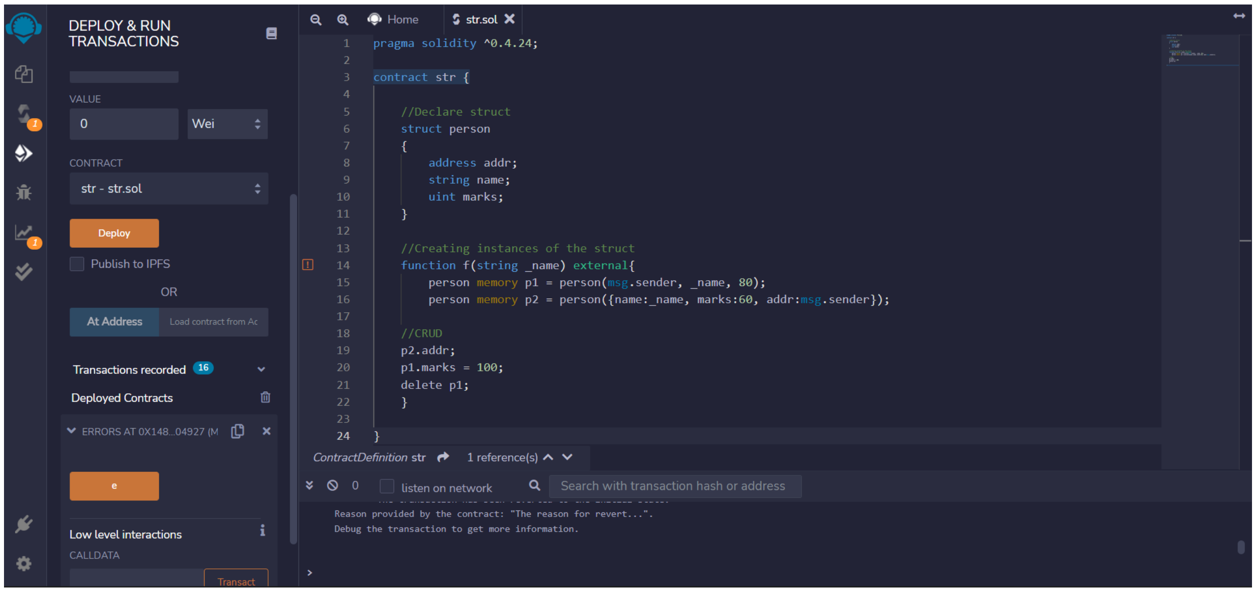
```
Reason provided by the contract: "The reason for revert...".
Debug the transaction to get more information.
```

9. Structs

Struct types are used to represent a record. To define a Struct, you must use the struct keyword. The struct keyword defines a new data type with more than one member. The format of the struct statement is as follows –

struct struct_name {
type1 type_name_1;
type2 type_name_2;
type3 type_name_3;
}

To access any member of a structure, we use the member access operator (.). The member access operator is coded as a period between the structure variable name and the structure member we wish to access.

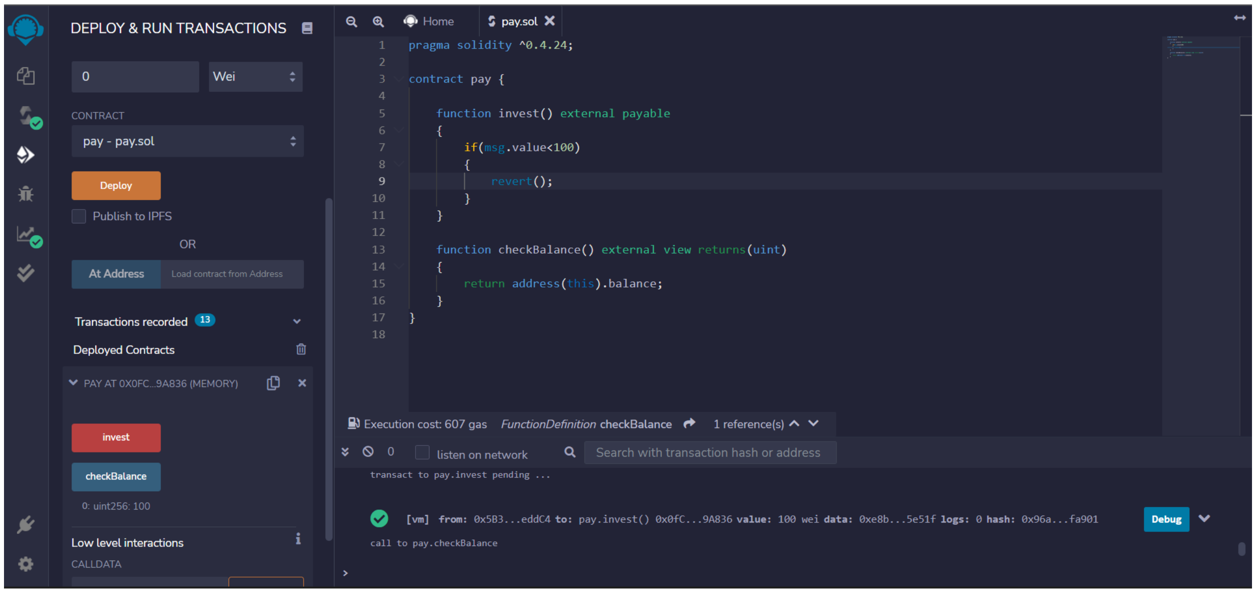


```
pragma solidity ^0.4.24;

contract str {

    //Declare struct
    struct person
    {
        address addr;
        string name;
        uint marks;
    }

    //Creating instances of the struct
    function f(string _name) external{
        person memory p1 = person(msg.sender, _name, 80);
        person memory p2 = person({name:_name, marks:60, addr:msg.sender});

    //CRUD
    p2.addr;
    p1.marks = 100;
    delete p1;
    }

}
```

```
Reason provided by the contract: "The reason for revert...".
Debug the transaction to get more information.
```

10. Sending ether to another address

Payable keyword: This makes sure that the function has access to ethers. Without this, it will reject the transaction.
In the below-given code, we have created two functions namely invest() and checkBalance(). The invest function will check if the added amount is sufficient enough and will revert if it is not. The checkBalance function will return the balance associated with the particular address.

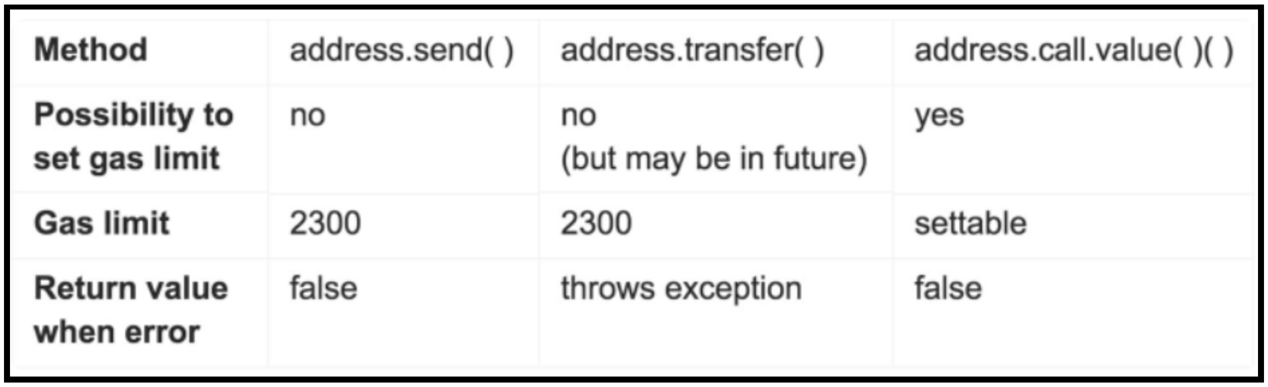


```
pragma solidity ^0.4.24;

contract pay {

    function invest() external payable
    {
        if(msg.value<100)
        {
            revert();
        }
    }

    function checkBalance() external view returns(uint)
    {
        return address(this).balance;
    }
}
```

```
transact to pay.invest pending ...

[vm] from: 0x5B3...eddC4 to: pay.invest() 0x0fC...9A836 value: 100 wei data: 0xe8b...5e51f logs: 0 hash: 0x96a...fa901
call to pay.checkBalance
```

Now, let's look at the three different methods of ether transfer:

a. **address.send(amount):** It has two details to be taken care of. The first is providing a 2300 gas limit of a fallback function of the contract receiving ether. The second one is handling unsuccessful executions. Hence, usage of send() should be inside of a require.
b. **address.transfer(amount):** This method has the same 2300 gas limit but it handles the errors at execution itself. So, one knows that the transaction is unsuccessful at the execution time only.
c. **address.call.value(amount)():** It is different from the other two as we can set the gas limit by using .gas(gasLimit).

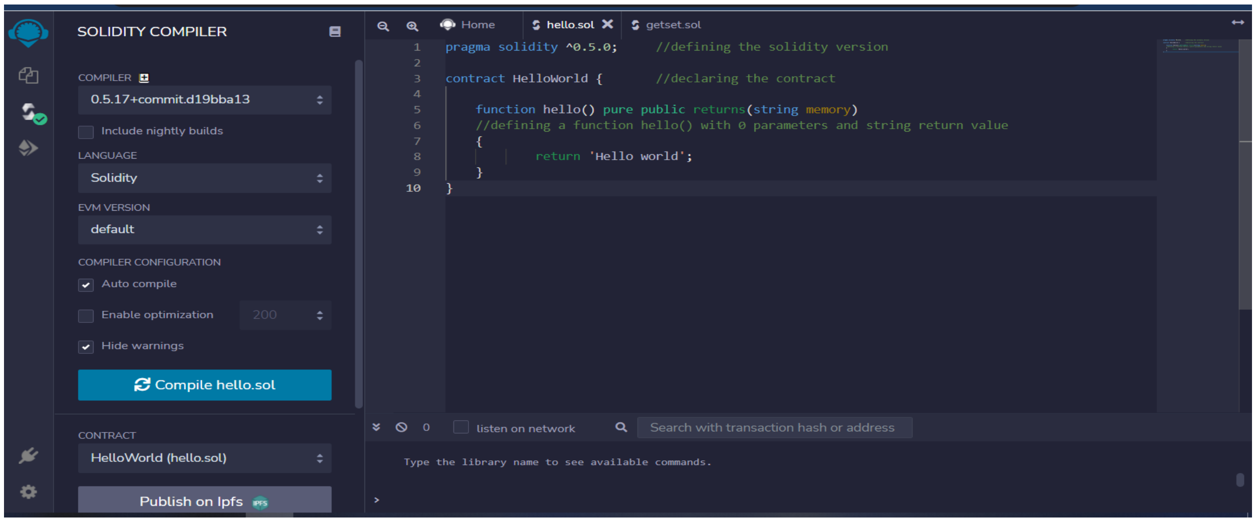


Method	address.send( )	address.transfer( )	address.call.value( )( )
Possibility to set gas limit	no	no (but may be in future)	yes
Gas limit	2300	2300	settable
Return value when error	false	throws exception	false

❖ SAMPLE SMART CONTRACTS:

To get a clear idea about the solidity language and remix IDE, let's go through certain sample smart contracts:

1. Hello world contract



SOLIDITY COMPILER

COMPILER: 0.5.17+commit.d19bba13

Include nightly builds

LANGUAGE: Solidity

EVM VERSION: default

COMPILER CONFIGURATION

- [x] Auto compile
- [ ] Enable optimization 200
- [x] Hide warnings

Compile hello.sol

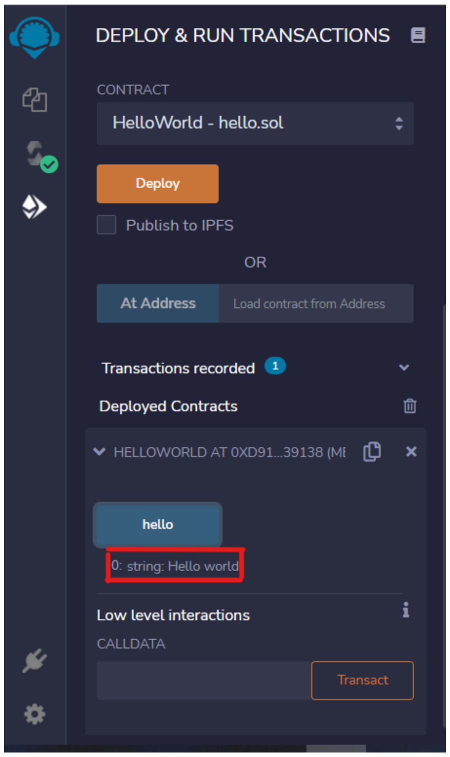
CONTRACT: HelloWorld (hello.sol)

Publish on Ipfs

```
pragma solidity ^0.5.0;     //defining the solidity version

contract HelloWorld {       //declaring the contract

    function hello() pure public returns(string memory)
    //defining a function hello() with 0 parameters and string return value
    {
            return 'Hello world';
    }
}
```

```
Type the library name to see available commands.
```

- Here, the solidity version is defined. The pragma keyword is used to enable certain compiler features or checks. The script is annotated with version pragma to reject compilation errors with future compiler versions.
- Inside the contract, the function hello() is defined using no parameters.
- It returns a string value, "Hello world" and it needs to be stored temporarily, so the keyword "memory" is used.
- The function is made public so that it can be accessed both internally and externally.
- Since this function doesn't read and modify the variables' states, hence it is defined to be "pure".
- After the contract definition, it is compiled and deployed.
- On deploying the contract at a particular address, we run the function "Hello" and it displays the output as "Hello world".



DEPLOY & RUN TRANSACTIONS

CONTRACT: HelloWorld - hello.sol

Transactions recorded 1

Deployed Contracts

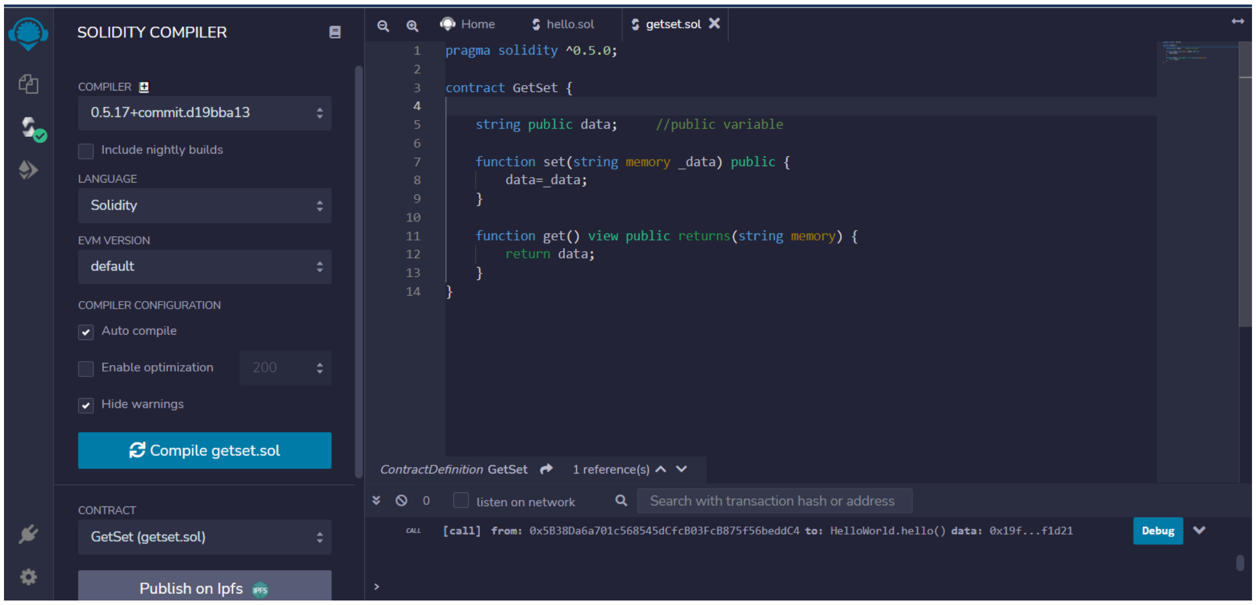
HELLOWORLD AT 0XD91...39138 (ME

```
0: string: Hello world
```

Low level interactions

CALLDATA

2. Get and Set data contract



COMPILER: 0.5.17+commit.d19bba13

Include nightly builds

LANGUAGE: Solidity

EVM VERSION: default

COMPILER CONFIGURATION

- ✓ Auto compile
- Enable optimization 200
- ✓ Hide warnings

CONTRACT: GetSet (getset.sol)

```
pragma solidity ^0.5.0;

contract GetSet {

    string public data;     //public variable

    function set(string memory _data) public {
        data=_data;
    }

    function get() view public returns(string memory) {
        return data;
    }
}
```

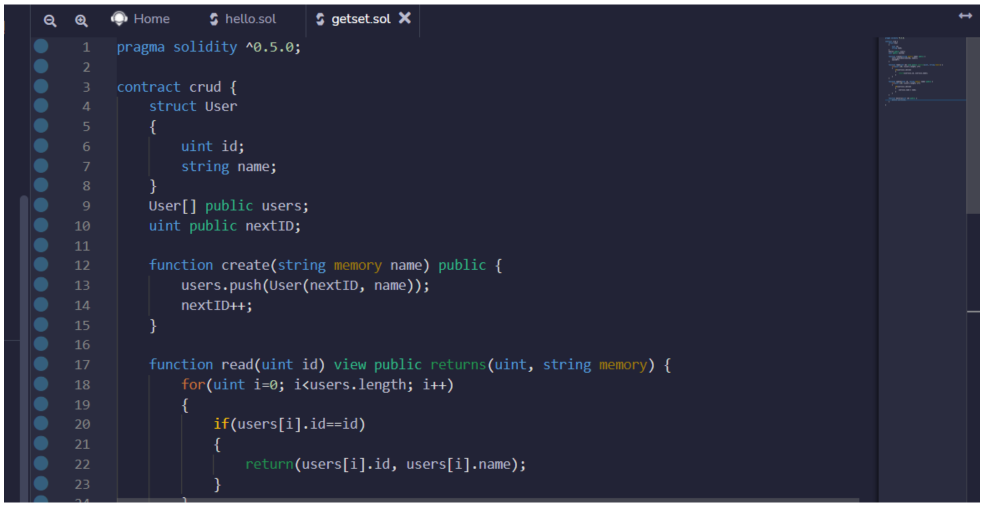
ContractDefinition GetSet   1 reference(s)

0   listen on network   Search with transaction hash or address

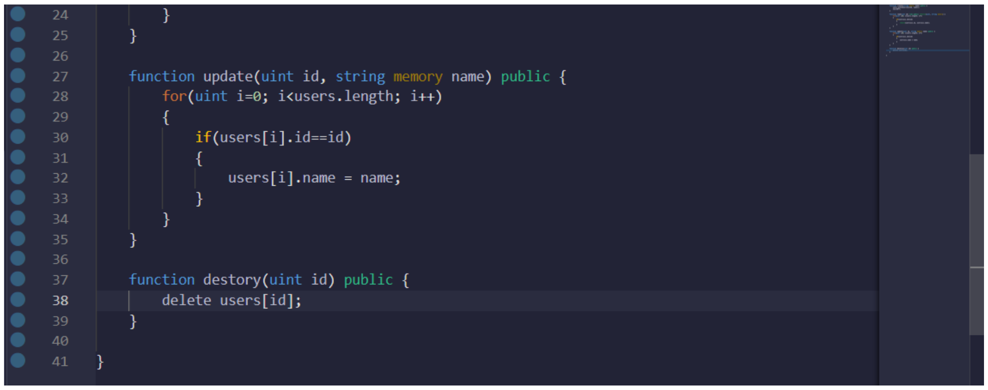
```
CALL  [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: HelloWorld.hello() data: 0x19f...f1d21
```

- Contract GetSet is defined that basically takes the data as the input and sets it. It can be then retrieved using the get function.
- Public variable data is defined to use both internally and externally.
- The function set() takes the input from the user and sets it into the “data” variable.
- The function get() retrieves the data stored.

3. Create, read, update and delete contract

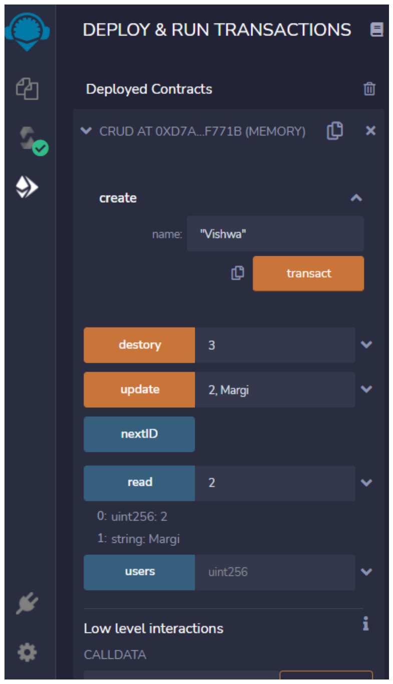


```
pragma solidity ^0.5.0;

contract crud {
    struct User
    {
        uint id;
        string name;
    }
    User[] public users;
    uint public nextID;

    function create(string memory name) public {
        users.push(User(nextID, name));
        nextID++;
    }

    function read(uint id) view public returns(uint, string memory) {
        for(uint i=0; i<users.length; i++)
        {
            if(users[i].id==id)
            {
                return(users[i].id, users[i].name);
            }
```



```
        }
    }

    function update(uint id, string memory name) public {
        for(uint i=0; i<users.length; i++)
        {
            if(users[i].id==id)
            {
                users[i].name = name;
            }
        }
    }

    function destory(uint id) public {
        delete users[id];
    }

}
```

- This contract defines create, read, update and delete functions.
- A struct user is defined, with members as ID and name.
- Create() function adds the user with inputted name and next ID (ID needs not to be entered).
- Read() function returns name of the ID entered.
- Update() function updates the name provided as one of the parameters of the provided function.
- Destroy() function deletes the record.
- These functions are the basic implementation of main operations.



```
name: "Vishwa"
destory 3
update 2, Margi
read 2
0: uint256: 2
1: string: Margi
users uint256
```

❖ PROPOSED MODEL:

The proposed solution aims to model a smart contract system to manage Agile software development projects. We use the sprint model to design smart contracts. At the Sprint planning stage, the set of tasks and their requisite conditions validated by the stakeholders are defined and written into the master smart contract. Along with it, additional smart contracts are created for specific tasks which include the acceptance test cases and particular deliverables and acceptance protocols.
So, we can consider a project with several developers and a Scrum master. The testers validate the deliverables from developers. Once the task is completed, its corresponding smart contract is executed to determine whether the task deliverables meet expectations like test results, code quality evaluation, etc. Once all tasks defined in the sprint are completed or the sprint as a whole is completed, the master smart contract is executed to verify the status of the tasks and count on the penalties if resources and time aren't used adequately. Upon the completion of master contract execution, the sprint software module is accepted and payment clearance is done from the client to the vendor team and the same process is continued for other sprints as well.
The solution mainly includes 3 types of contracts:

1. Task.sol:

- This smart contract defines all the acceptance criteria to state whether the particular task is accepted or not.
- The solidity version used is 0.6.0.
- The experimental ABIEncoderV2 is imported to support the structs and nested arrays.
- Inside the definition of the contract, two structs are defined.
- Struct **"Price"** is defined to gather the information about the total cost in $ to perform a certain development task. We have divided the price in terms of software modification, web development, integration and resources.
- The main struct defined is **"criteria"** which considers the various acceptance criteria like line coverage, branch coverage, key, cyclic dependency and price.
- The key is the encoded sha256 code of the JSON file that include test cases with their expected outputs and the price takes into account the total cost of accomplishing the task.
- Function **set_detail()** is to set the criteria by the project owner. He decides on the estimated values of different criteria and adds the key of the test case JSON file and approximate price. The details are already coded in the contract, so by executing the function, the criteria are set.
- Function **take_detail()** takes the details of the criteria required directly from the code. In this dummy representation, we input the criteria manually.
- The JSON file with the test cases and the output of the code is created by the developer and its sha256 code is generated. So, this code will be inputted by the tester.

- The important function of the task smart contract is **check()**. It verifies all the criteria based on the predefined criteria.
- The values calculated from the code should lie in the given range values of line coverage, branch coverage, cyclic dependency. The key of the predefined JSON file should also match to the key of the generated JSON file.
- Also, the IDs of the testers are checked and if they aren't matching, then the respective IDs won't be allowed to execute the check function.
- If the conditions don't pass, the contract will prompt a message upon the criteria that need to be improved. And if all the conditions match, then the task is accepted and its status will be moved to the master smart contract.
- Similar contracts are made for other tasks in the sprint as well.

2. Master.sol:

- This smart contract is designed by the product owner to check whether all the tasks are submitted correctly and on time. It also counts the penalty points of the developer who has executed the particular task contract.
- The solidity version used is 0.6.0.
- The experimental ABIEncoderV2 is imported to support the structs and nested arrays.
- All the task smart contracts are imported into the master smart contract to use its functions and data.
- **setAddressT()** function is defined to take in all the addresses of the deployed task contracts to get the values of the required variables.
- Exptime array stores the time required to complete a particular task.
- The Exp's arrays define the cost required for the particular task. The price is divided into other factors.
- All these time and cost factors are decided and inputted by the product owner.
- A mapping is created to store the developer's addresses and the penalty points associated.
- Function **callCheck()** is defined to calculate the penalty points of the developers based on the time taken by them to execute the task contract and the cost associated with it.
- This function only allows the product owner to execute and his/her address is already defined in the contract.
- An object is created for each task and it is called using the task contract's address.
- The check() function is also called to get the status of each task, time taken and total cost to complete the task.
- If the difference between the recorded start time and end time of the task is greater than the expected time duration provided by the project owner, the penalty points are added to the developer's address.

- If the price turns out to be higher than the expected price then also the penalty points get added and these points can be later used to decide upon the developers' pay.
- This is done for each task and its respective developer's penalty points are calculated.
- Then, the status of all the tasks is verified and if all the tasks are accepted then it further proceeds for the payment to the stakeholder's contract or else it is canceled and the product owner is informed about the same.

3. Client.sol:

- This contract is written by the stakeholder/client to check whether the sprint is completed on time and whether all tasks are completed correctly.
- The function **timestamp1()** takes the expected date from the stakeholder by which all the tasks in the particular sprint are completed and delivered.
- **addMasterAdd()** takes the deployed master contract's address to access the calculated values of the master contract.
- Function **payEther()** is used to transfer ethers from the address of the stakeholder to the stakeholder contract so that they can be directly transferred to the product owner's address.
- Function **getBalance()** to check the balance of the stakeholder contract.
- Function **sendEtherAccount()** is mainly used to transfer the ethers to the product owner.
- It first reads the product owner's address for the payment.
- The callCheck() function from the master smart contract is called to get the hands up regarding the payment and also to get the timestamp of the sprint completion.
- If the time exceeds the expected completion time then based on the days of delay the amount of ether transferred is reduced by a certain factor or else the full amount is transferred.
- This helps in maintaining transparency in payment and also the payment is done on time.

❖ REFERENCES:

1. Best Ethereum Solidity beginner level tutorials - YouTube
2. Solidity Tutorial - YouTube
3. Solidity / Ethereum Smart Contract BEGINNER Tutorial - Create 5 Smart Contracts
4. Solidity Tutorial - Telusko
5. Solidity Tutorial: Call function of other contract
6. Remix IDE: Documentation: remix-ide.pdf