**Solidity is an object oriented ,high level language for implementing smart contract(Dapps).Smart contracts are programs which govern the behaviour of accounts within the Ethereum state.**

**Solidity was influenced by c++ ,Python and javascript and is designed to target the Ethereum Virtual Machine(EVM)**

**Solidity is static typed supports inheritance,libraries and complex user-defined types among other features.With solidity you can create contracts for uses such as voting ,crow funding ,blind auctions and multi-signature wallets.**

**Practical 3A**

**AIM: WRITE A SOLIDITY PROGRAM FOR VARIABLES, OPERATORS, LOOPS, DECISION MAKING AND STRING.**

**A)Variables:**

supports three types of variables.

**State Variables** − Variables whose values are permanently stored in a contract storage.

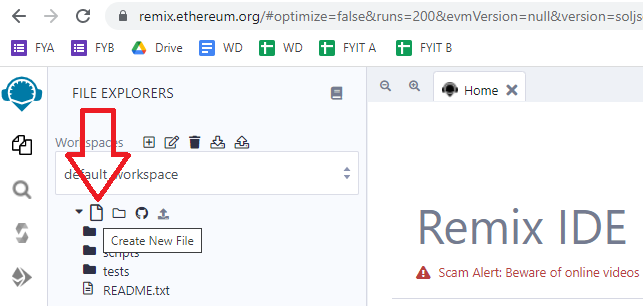
**Local Variables** − Variables whose values are present till function is executing.

**Global Variables** − Special variables exists in the global namespace used to get information about the blockchain.i.e. blockhash(uint blockNumber) returns (bytes32), block.coinbase (address payable), block.difficulty (uint)…..and many more

Step 1: Open this website

<https://remix.ethereum.org/>

Step 2: Create new file – practical.sol



Step 3: Write this program in the new file

///////////////

pragma solidity ^0.5.0;

contract SolidityTest {

uint storedData; // State variable

constructor() public {

storedData = 10;

}

function getResult() public view returns(uint){

uint a = 1; // local variable

uint b = 2;

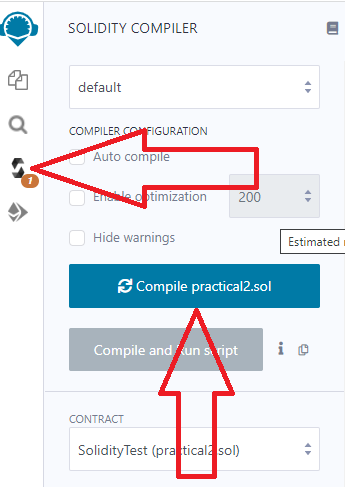
uint result = a + b;

return result; //access the state variable

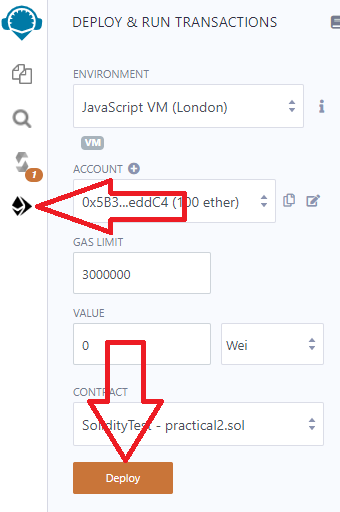
}

}

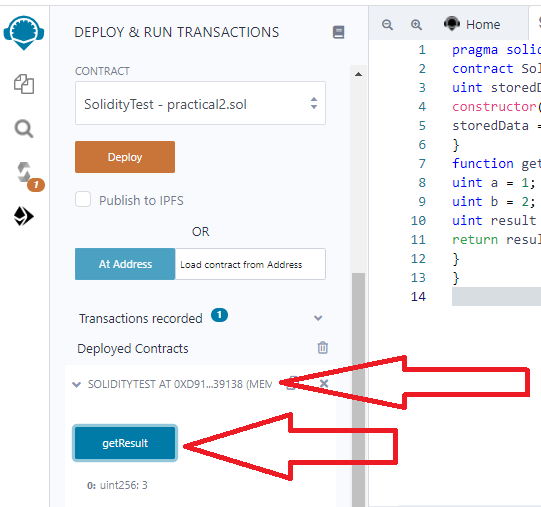
Step 4: Compile contract



Step 5: Deploy contract



Step 6: Select the contract and click button



**1.State Variable:**

// Solidity program to

// demonstrate state

// variables

pragma solidity ^0.5.0;

// Creating a contract

contract Solidity\_var\_Test {

// Declaring a state variable

uint8 public state\_var;

// Defining a constructor

constructor() public {

state\_var = 16;

}

}



**2.Local Variable:**

// Solidity program to demonstrate

// local variables

pragma solidity ^0.5.0;

// Creating a contract

contract Solidity\_var\_Test {

// Defining function to show the declaration and

// scope of local variables

function getResult() public view returns(uint){

// Initializing local variables

uint local\_var1 = 1;

uint local\_var2 = 2;

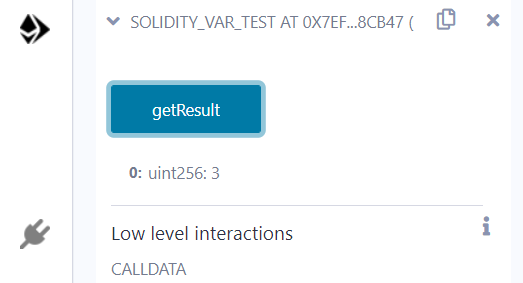
uint result = local\_var1 + local\_var2;

// Access the local variable

return result;

}

}



**3.Global variable:**

// Solidity program to

// show Global variables

pragma solidity ^0.5.0;

// Creating a contract

contract Test {

// Defining a variable

address public admin;

// Creating a constructor to

// use Global variable

constructor() public {

admin = msg.sender;

}

}



Scope of local variables is limited to function in which they are defined but State variables can have three types of scopes.

**Public** − Public state variables can be accessed internally as well as via messages. For a public state variable, an automatic getter function is generated.

**Internal** − Internal state variables can be accessed only internally from the current contract or contract deriving from it without using this.

**Private** − Private state variables can be accessed only internally from the current contract they are defined not in the derived contract from it.

**B)Operators**

Solidity supports the following types of operators.

Arithmetic Operators

Comparison Operators

Logical (or Relational) Operators

Assignment Operators

Conditional (or ternary) Operators

**1. Arithematic Operator**

// Solidity contract to demonstrate

// Arithematic Operator

pragma solidity ^0.5.0;

// Creating a contract

contract SolidityTest {

// Initializing variables

uint16 public a = 20;

uint16 public b = 10;

// Initializing a variable

// with sum

uint public sum = a + b;

// Initializing a variable

// with the difference

uint public diff = a - b;

// Initializing a variable

// with product

uint public mul = a \* b;

// Initializing a variable

// with quotient

uint public div = a / b;

// Initializing a variable

// with modulus

uint public mod = a % b;

// Initializing a variable

// decrement value

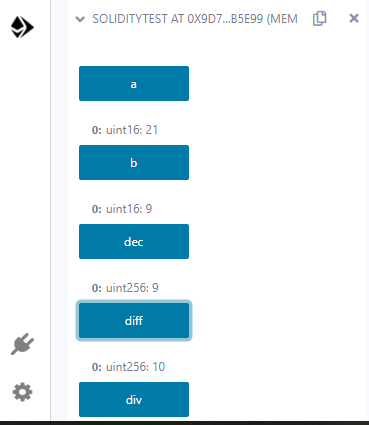
uint public dec = --b;

// Initializing a variable

// with increment value

uint public inc = ++a;

}



**2.Relational Operator**

// Solidity program to demonstrate

// Relational Operator

pragma solidity ^0.5.0;

// Creating a contract

contract SolidityTest {

// Declaring variables

uint16 public a = 20;

uint16 public b = 10;

// Initializing a variable

// with bool equal result

bool public eq = a == b;

// Initializing a variable

// with bool not equal result

bool public noteq = a != b;

// Initializing a variable

// with bool greater than result

bool public gtr = a > b;

// Initializing a variable

// with bool less than result

bool public les = a < b;

// Initializing a variable

// with bool greater than equal to result

bool public gtreq = a >= b;

// Initializing a variable

// bool less than equal to result

bool public leseq = a <= b;

}

**3.Logical Operators**

// Solidity program to demonstrate

// Logical Operators

pragma solidity ^0.5.0;

// Creating a contract

contract logicalOperator{

// Defining function to demonstrate

// Logical operator

function Logic(

bool a, bool b) public view returns(

bool, bool, bool){

// Logical AND operator

bool and = a&&b;

// Logical OR operator

bool or = a||b;

// Logical NOT operator

bool not = !a;

return (and, or, not);

}

}

**4.Bitwise Operators**

// Solidity program to demonstrate

// Bitwise Operator

pragma solidity ^0.5.0;

// Creating a contract

contract SolidityTest {

// Declaring variables

uint16 public a = 20;

uint16 public b = 10;

// Initializing a variable

// to '&' value

uint16 public and = a & b;

// Initializing a variable

// to '|' value

uint16 public or = a | b;

// Initializing a variable

// to '^' value

uint16 public xor = a ^ b;

// Initializing a variable

// to '<<' value

uint16 public leftshift = a << b;

// Initializing a variable

// to '>>' value

uint16 public rightshift = a >> b;

// Initializing a variable

// to '~' value

uint16 public not = ~a ;

}

**5.Assignment Operator**

// Solidity program to demonstrate

// Assignment Operator

pragma solidity ^0.5.0;

// Creating a contract

contract SolidityTest {

// Declaring variables

uint16 public assignment = 20;

uint public assignment\_add = 50;

uint public assign\_sub = 50;

uint public assign\_mul = 10;

uint public assign\_div = 50;

uint public assign\_mod = 32;

// Defining function to

// demonstrate Assignment Operator

function getResult() public{

assignment\_add += 10;

assign\_sub -= 20;

assign\_mul \*= 10;

assign\_div /= 10;

assign\_mod %= 20;

return ;

}

}

6**.Conditional Operators**

// Solidity program to demonstrate

// Conditional Operator

pragma solidity ^0.5.0;

// Creating a contract

contract SolidityTest{

// Defining function to demonstrate

// conditional operator

function sub(

uint a, uint b) public view returns(

uint){

uint result = (a > b? a-b : b-a);

return result;

}

}

**C)Loops:**

1.While loop: The most basic loop in Solidity is the **while** loop which would be discussed in this chapter. The purpose of a **while** loop is to execute a statement or code block repeatedly as long as an **expression** is true. Once the expression becomes **false,** the loop terminates.

2.do-while loop: The **do...while** loop is similar to the **while** loop except that the condition check happens at the end of the loop. This means that the loop will always be executed at least once, even if the condition is **false**.

3.for loop: The **for** loop is the most compact form of looping. It includes the following three important parts −

The **loop initialization** where we initialize our counter to a starting value. The initialization statement is executed before the loop begins.

The **test statement** which will test if a given condition is true or not. If the condition is true, then the code given inside the loop will be executed, otherwise the control will come out of the loop.

The **iteration statement** where you can increase or decrease your counter.

4.loop control: Solidity provides full control to handle loops and switch statements. There may be a situation when you need to come out of a loop without reaching its bottom. There may also be a situation when you want to skip a part of your code block and start the next iteration of the loop.To handle all such situations, Solidity provides **break** and **continue** statements. These statements are used to immediately come out of any loop or to start the next iteration of any loop respectively.

**1.While Loop**

pragma solidity ^0.5.0;

contract SolidityTest {

uint storedData;

constructor() public{

storedData = 10;

}

function getResult() public view returns(string memory){

uint a = 10;

uint b = 2;

uint result = a + b;

return integerToString(result);

}

function integerToString(uint \_i) internal pure

returns (string memory) {

if (\_i == 0) {

return "0";

}

uint j = \_i;

uint len;

while (j != 0) {

len++;

j /= 10;

}

bytes memory bstr = new bytes(len);

uint k = len - 1;

while (\_i != 0) { // while loop

bstr[k--] = byte(uint8(48 + \_i % 10));

\_i /= 10;

}

return string(bstr);

}

}

**2.Do-while loop:**

pragma solidity ^0.5.0;

contract SolidityTest {

uint storedData;

constructor() public{

storedData = 10;

}

function getResult() public view returns(string memory){

uint a = 10;

uint b = 2;

uint result = a + b;

return integerToString(result);

}

function integerToString(uint \_i) internal pure

returns (string memory) {

if (\_i == 0) {

return "0";

}

uint j = \_i;

uint len;

while (j != 0) {

len++;

j /= 10;

}

bytes memory bstr = new bytes(len);

uint k = len - 1;

do {                   // do while loop

bstr[k--] = byte(uint8(48 + \_i % 10));

\_i /= 10;

}

while (\_i != 0);

return string(bstr);

}

}

**3.For Loop:**

pragma solidity ^0.5.0;

contract SolidityTest {

uint storedData;

constructor() public{

storedData = 10;

}

function getResult() public view returns(string memory){

uint a = 10;

uint b = 2;

uint result = a + b;

return integerToString(result);

}

function integerToString(uint \_i) internal pure

returns (string memory) {

if (\_i == 0) {

return "0";

}

uint j=0;

uint len;

for (j = \_i; j != 0; j /= 10) {  //for loop example

len++;

}

bytes memory bstr = new bytes(len);

uint k = len - 1;

while (\_i != 0) {

bstr[k--] = byte(uint8(48 + \_i % 10));

\_i /= 10;

}

return string(bstr);//access local variable

}}

**4.loop Control: (Break statement)**

pragma solidity ^0.5.0;

contract SolidityTest {

uint storedData;

constructor() public{

storedData = 10;

}

function getResult() public view returns(string memory){

uint a = 1;

uint b = 2;

uint result = a + b;

return integerToString(result);

}

function integerToString(uint \_i) internal pure

returns (string memory) {

if (\_i == 0) {

return "0";

}

uint j = \_i;

uint len;

while (true) {

len++;

j /= 10;

if(j==0){

break;   //using break statement

}

}

bytes memory bstr = new bytes(len);

uint k = len - 1;

while (\_i != 0) {

bstr[k--] = byte(uint8(48 + \_i % 10));

\_i /= 10;

}

return string(bstr);

}

}

**(continue statement)**

pragma solidity ^0.5.0;

contract SolidityTest {

uint storedData;

constructor() public{

storedData = 10;

}

function getResult() public view returns(string memory){

uint n = 1;

uint sum = 0;

while( n < 10){

n++;

if(n == 5){

continue; // skip n in sum when it is 5.

}

sum = sum + n;

}

return integerToString(sum);

}

function integerToString(uint \_i) internal pure

returns (string memory) {

if (\_i == 0) {

return "0";

}

uint j = \_i;

uint len;

while (true) {

len++;

j /= 10;

if(j==0){

break;   //using break statement

}

}

bytes memory bstr = new bytes(len);

uint k = len - 1;

while (\_i != 0) {

bstr[k--] = byte(uint8(48 + \_i % 10));

\_i /= 10;

}

return string(bstr);

}

}

**D) Decision Making:**

While writing a program, there may be a situation when you need to adopt one out of a given set of paths. In such cases, you need to use conditional statements that allow your program to make correct decisions and perform right actions.Solidity supports conditional statements which are used to perform different actions based on different conditions. Here we will explain the **if..else** statement.

1.if statement: The **if** statement is the fundamental control statement that allows Solidity to make decisions and execute statements conditionally.

pragma solidity ^0.5.0;

contract SolidityTest {

uint storedData;

constructor() public {

storedData = 10;

}

function getResult() public view returns(string memory){

uint a = 1;

uint b = 2;

uint result = a + b;

return integerToString(result);

}

function integerToString(uint \_i) internal pure

returns (string memory) {

if (\_i == 0) {   // if statement

return "0";

}

uint j = \_i;

uint len;

while (j != 0) {

len++;

j /= 10;

}

bytes memory bstr = new bytes(len);

uint k = len - 1;

while (\_i != 0) {

bstr[k--] = byte(uint8(48 + \_i % 10));

\_i /= 10;

}

return string(bstr);//access local variable

}}

**2.if-else statement:** The **'if...else'** statement is the next form of control statement that allows Solidity to execute statements in a more controlled way.

pragma solidity ^0.5.0;

// Creating a contract

contract Types {

// Declaring state variables

uint i = 10;

bool even;

// Defining function to

// demonstrate the use of

// 'if...else statement'

function decision\_making(

) public payable returns(bool){

if (i%2 == 0){

even = true;

}

else{

even = false;

}

return even;

}

}

**3.if-else..if statement**: The **if...else if...** statement is an advanced form of **if...else** that allows Solidity to make a correct decision out of several conditions.

pragma solidity ^0.5.0;

// Creating a contract

contract Types {

// Declaring state variables

uint i = 12;

string result;

// Defining function to

// demonstrate the use

// of 'if...else if...else

// statement'

function decision\_making (

) public returns(string memory){

if(i<10){

result = "less than 10";

}

else if(i == 10){

result = "equal to 10";

}

else{

result = "greater than 10";

}

return result;

}

}

**String:**

// Solidity program to demonstrate

// how to create a contract

pragma solidity ^0.4.23;

// Creating a contract

contract Test {

// Declaring variable

string  str;

// Defining a constructor

constructor(string str\_in){

str = str\_in;

}

// Defining a function to

// return value of variable 'str'

function str\_out() public view returns(string memory){

return str;

}

}

Note: after deploy it asked u to enter string then enter string over there and then see the output after clicking on str\_out button