Blockchain Programming with Solidity

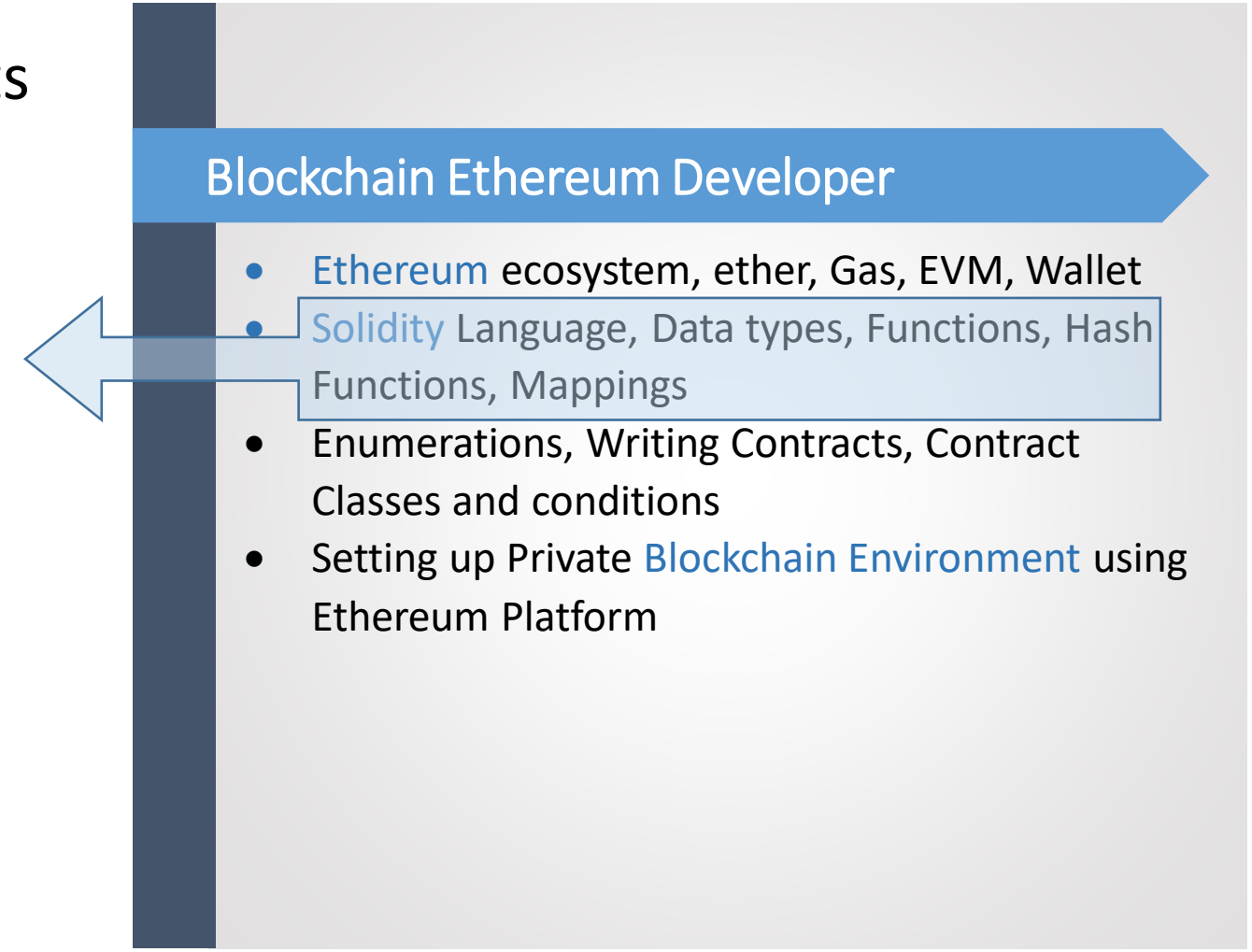
Ethereum – an world computer

Dr. Sarwan Singh

NIELIT Chandigarh

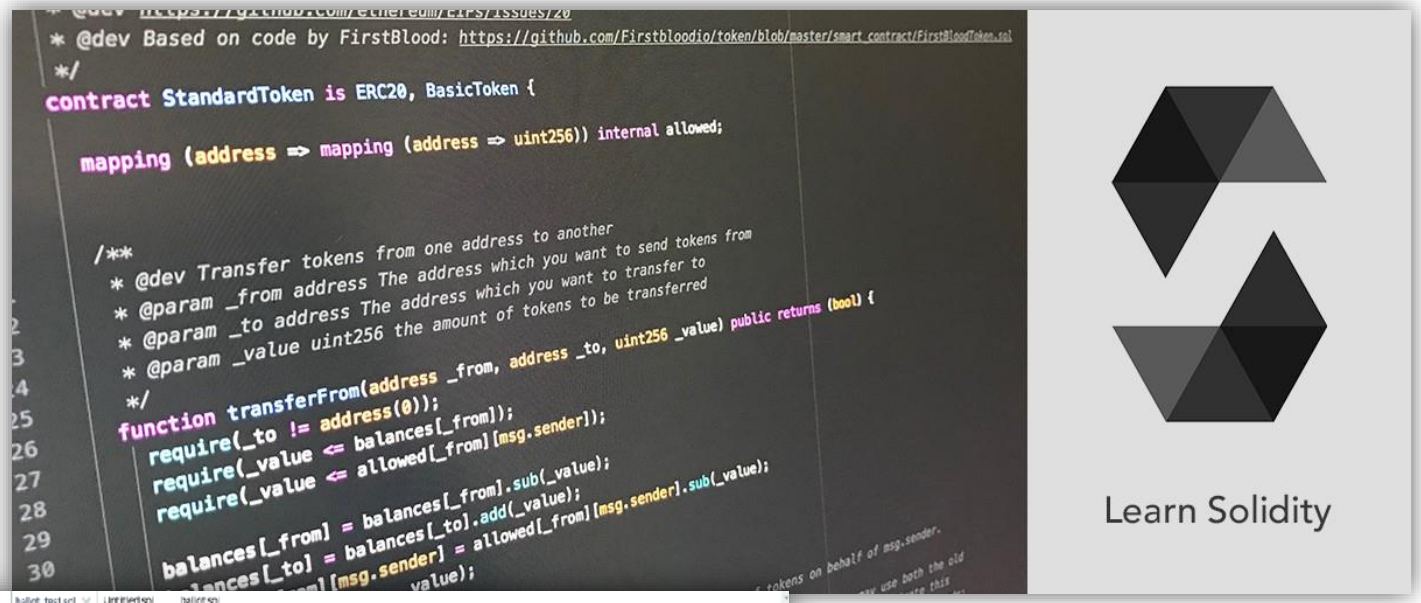
Agenda

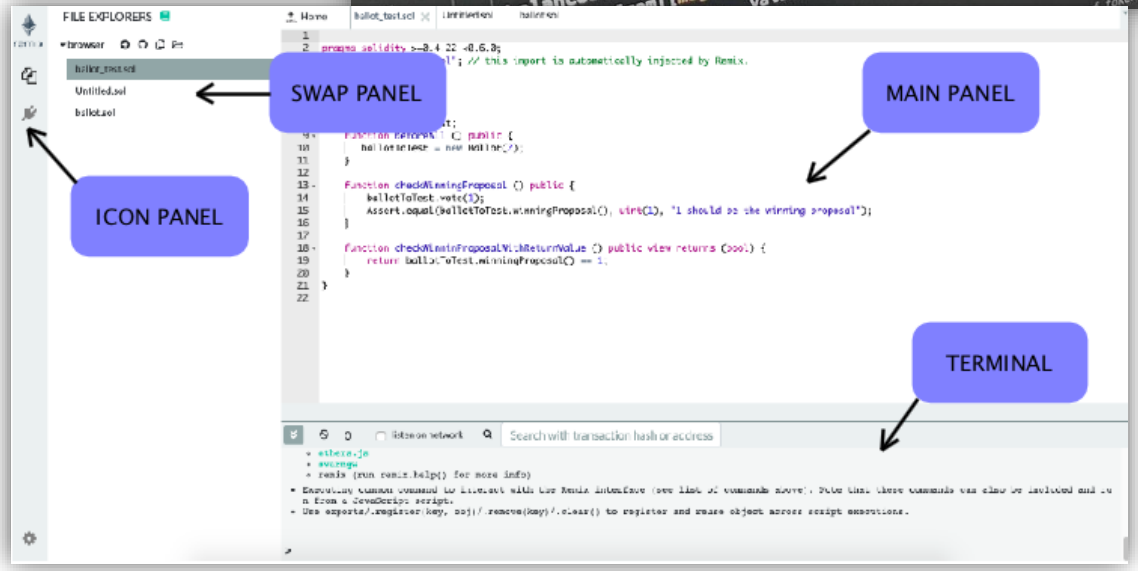
- Solidity programming constructs
- Remix IDE
  - Compile, deploy…
- pragma directive
- Datatype
- Keywords
- Operators



References

- Medium.com – Blockchain
- solidity.readthedocs.io
- tutorialspoint.com
- Dappuniversity.com
- Remix.readthedocs.io





Solidity – an Introduction

- Solidity is an object-oriented, high-level language for implementing smart contracts. Smart contracts are programs which govern the behavior 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 statically typed, supports inheritance, libraries and complex user-defined types among other features.
- With Solidity you can create contracts for uses such as voting, crowdfunding, blind auctions, and multi-signature wallets.

Source : solidity.readthedocs.io

Solidity

- A Solidity source files can contain an any number of contract definitions, import directives and pragma directives.

```
pragma solidity >=0.4.0  <0.6.0;
contract SimpleStorage {
    uint  storedData;
    function set(uint x) public {
        storedData = x;
    }
    function get() public view returns (uint) {
        return storedData;
    }
}
```

Compile-Deploy… *first application*

- https://remix.ethereum.org/
- Step 1 – type/Copy the (given) code in Remix IDE Code Section.
- Step 2 – Under Compile Tab, click Start to Compile button.
- Step 3 – Under Run Tab, click Deploy button.
- Step 4 – Under Run Tab, Select Solidity Test at 0x... in drop-down.
- Step 5 – Click get *Button* to display the result.

Pragma

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

- The first line is a pragma directive which tells that the source code is written for Solidity version 0.4.0 or anything newer that does not break functionality up to, but not including, version 0.6.0.
- A pragma directive is always local to a source file and if you import another file, the pragma from that file will not automatically apply to the importing file.

```
pragma solidity ^0.4.0
```

- pragma for a file which will not compile earlier than version 0.4.0 and it will also not work on a compiler starting from version 0.5.0

Contract

- A Solidity contract is a collection of code (its functions) and data (its state) that resides at a specific address on the Ethereum blockchain.
- The line uint storedData declares a state variable called storedData of type uint and the functions set and get can be used to modify or retrieve the value of the variable.

```
pragma solidity >=0.4.0  <0.6.0;
contract SimpleStorage {
    uint  storedData;
    function set(uint x) public {
        storedData = x;
    }
    function get() public view returns
(uint) {
        return storedData;
    }
}
```

Comments

Solidity supports both C-style and C++-style comments, Thus –

- Any text between a // and the end of a line is treated as a comment and is ignored by Solidity Compiler.
- Any text between the characters /* and */ is treated as a comment. This may span multiple lines.

Import files

- Solidity supports import statements that are very similar to those available in JavaScript.
- The following statement imports all global symbols from "filename“.

```
import "filename";
```

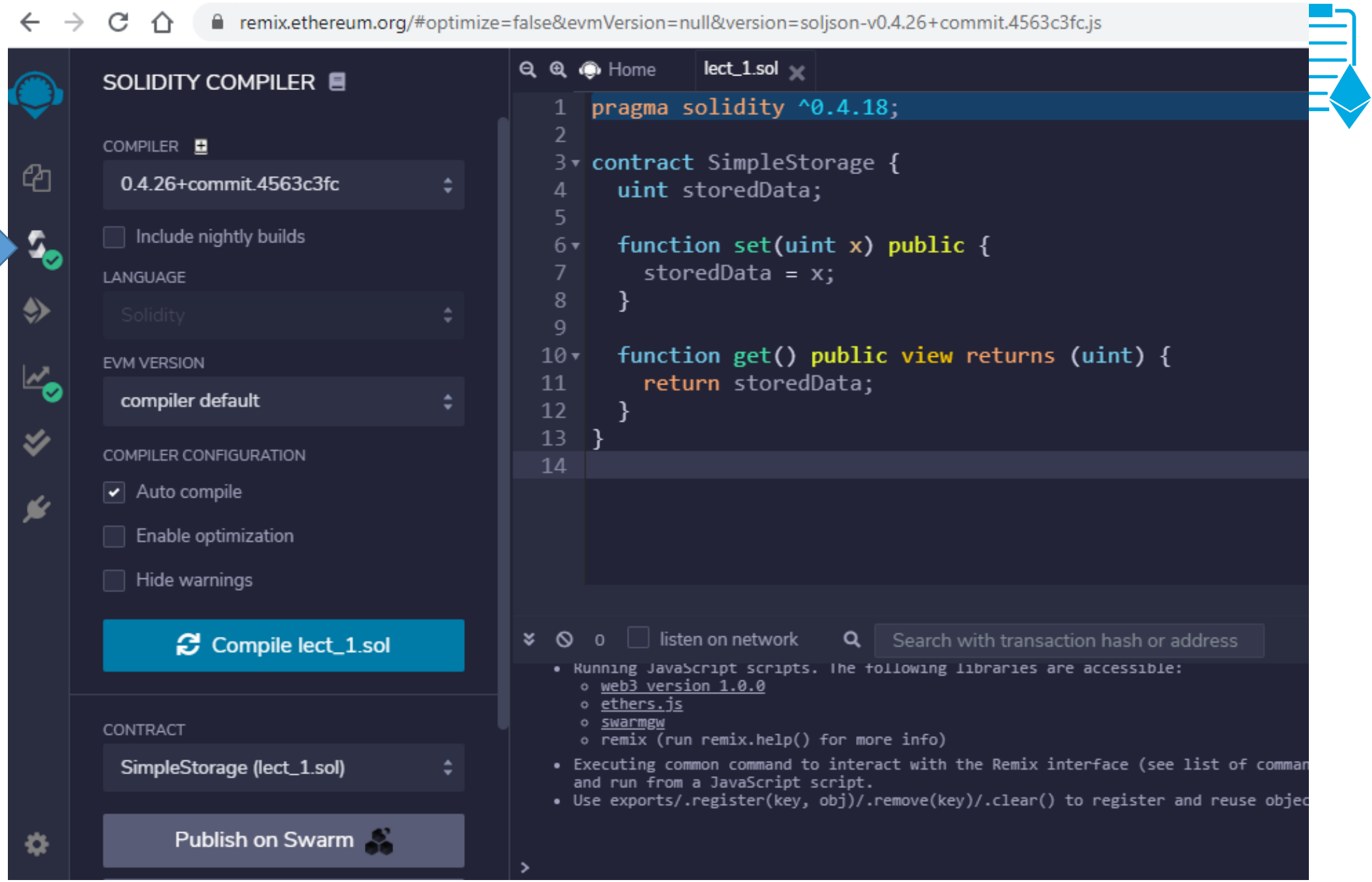
- creates a new global symbol symbolName whose members are all the global symbols from "filename".

```
import * as symbolName from "filename";
```

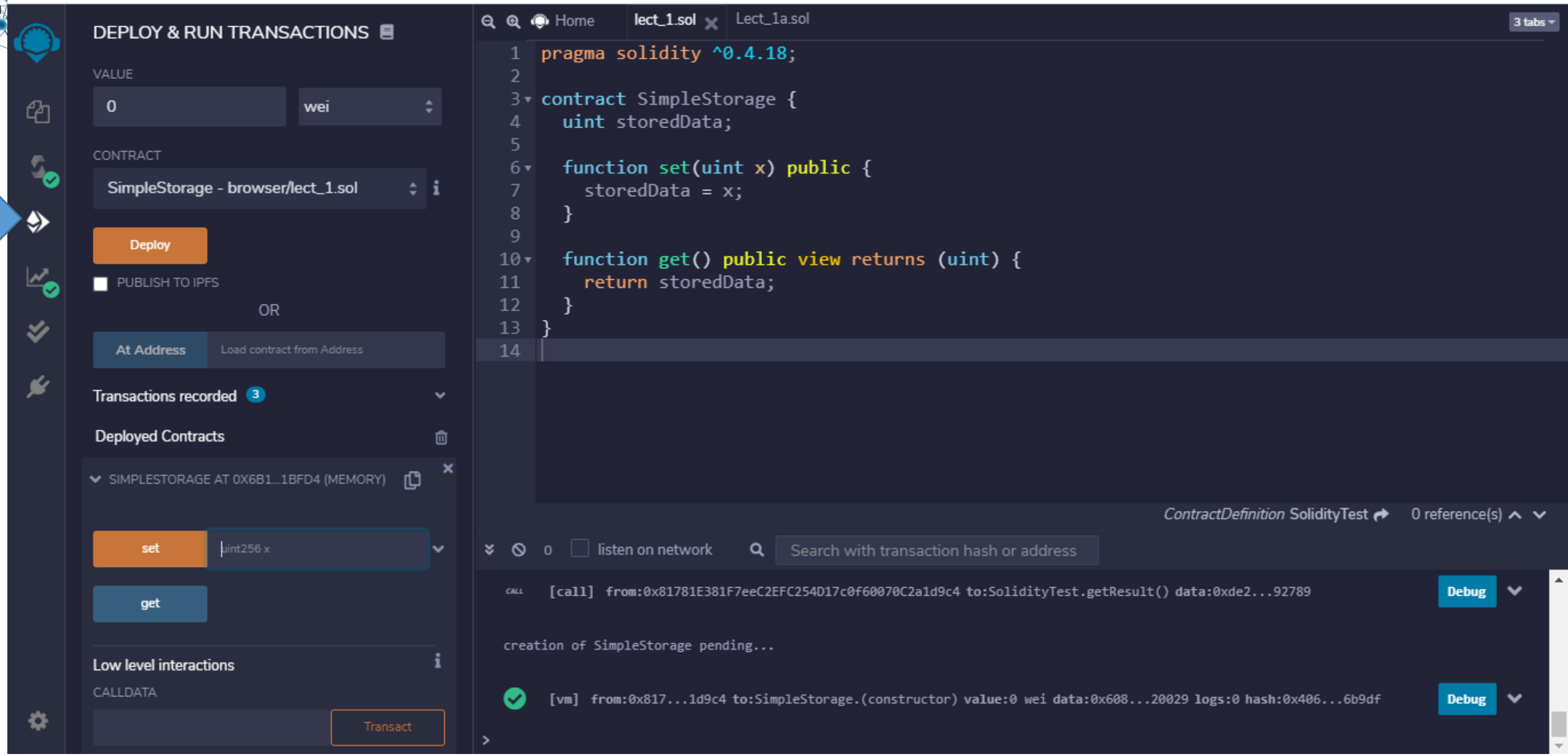
keywords

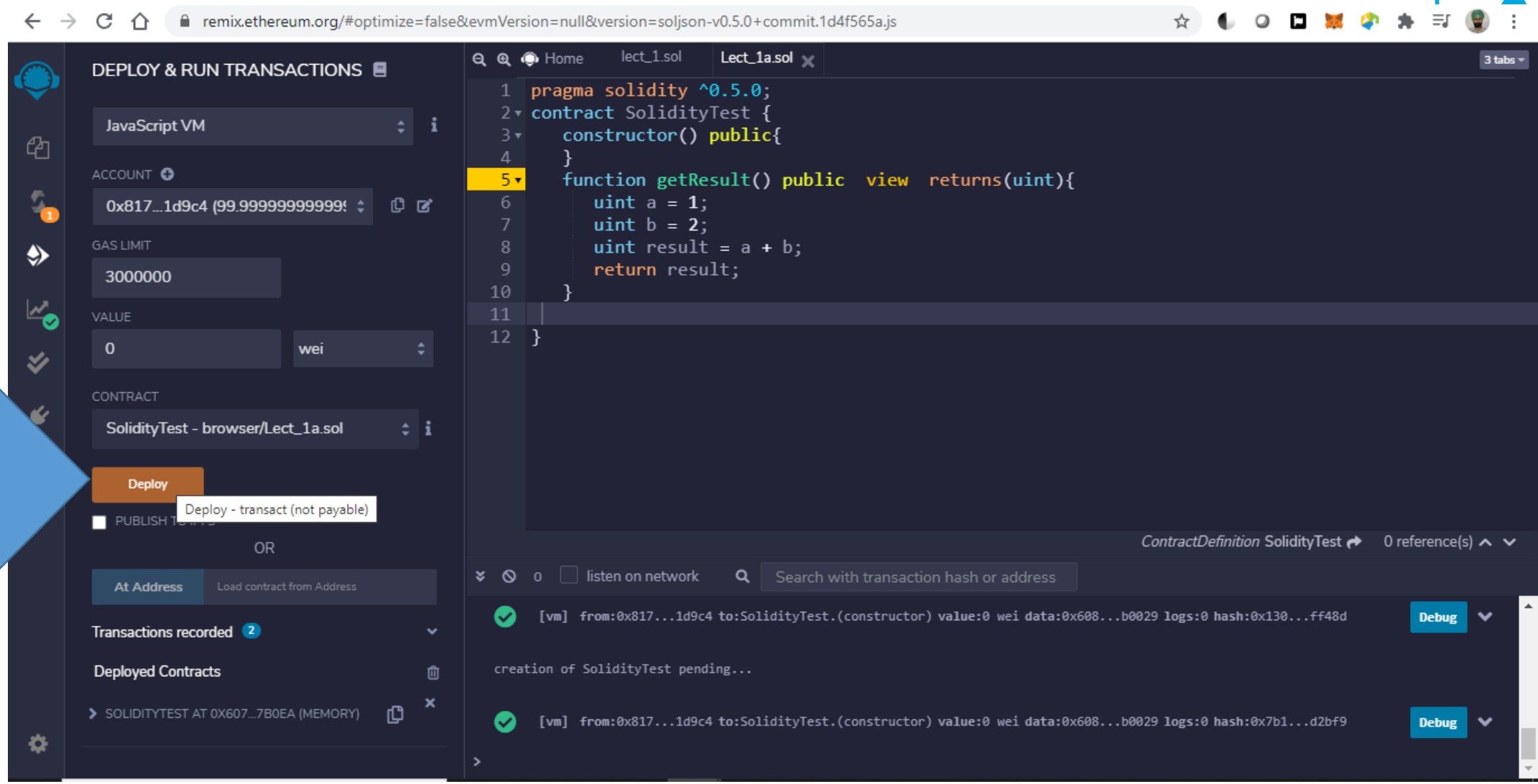
abstract	after	alias	apply
auto	case	catch	copyof
default	define	final	immutable
implements	in	inline	let
macro	match	mutable	null
of	override	partial	promise
reference	relocatable	sealed	sizeof
static	supports	switch	try
typedef	typeof	unchecked	

compile

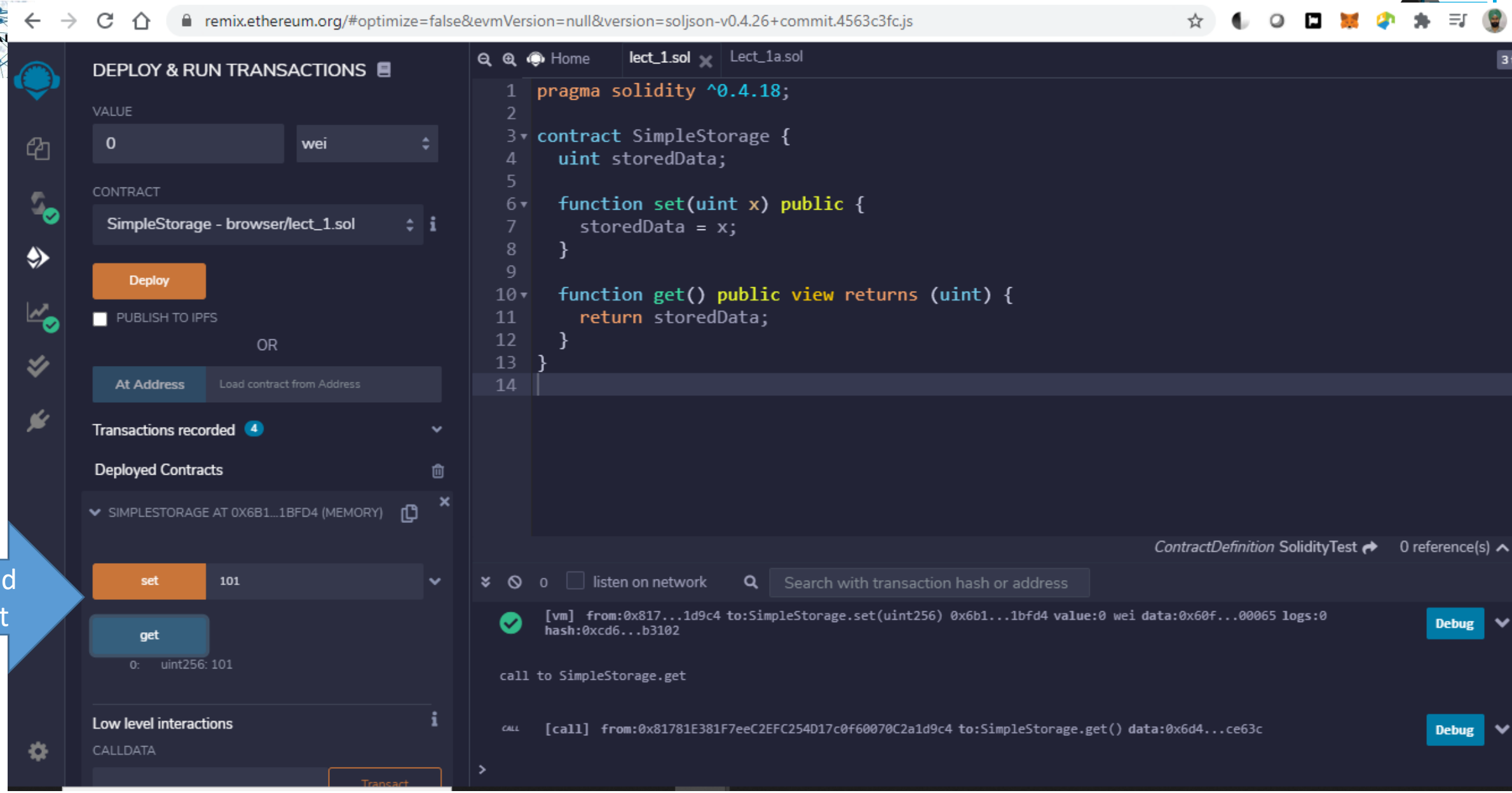

remix.ethereum.org/#optimize=false&evmVersion=null&version=soljson-v0.4.26+commit.4563c3fc.js
SOLIDITY COMPILER
COMPILER
0.4.26+commit.4563c3fc
Include nightly builds
LANGUAGE
Solidity
EVM VERSION
compiler default
COMPILER CONFIGURATION
Auto compile
Enable optimization
Hide warnings
Compile lect_1.sol
CONTRACT
SimpleStorage (lect_1.sol)
Publish on Swarm
Home
lect_1.sol
pragma solidity ^0.4.18;
contract SimpleStorage {
uint storedData;
function set(uint x) public {
storedData = x;
}
function get() public view returns (uint) {
return storedData;
}
}
listen on network
Search with transaction hash or address
Running JavaScript scripts. The following libraries are accessible:
web3 version 1.0.0
ethers.js
swarmgw
remix (run remix.help() for more info)
Executing common command to interact with the Remix interface (see list of comman
and run from a JavaScript script.
Use exports/.register(key, obj)/.remove(key)/.clear() to register and reuse objec

Deploy


remix.ethereum.org/#optimize=false&evmVersion=null&version=soljson-v0.4.26+commit.4563c3fc.js
DEPLOY & RUN TRANSACTIONS
VALUE
0
wei
CONTRACT
SimpleStorage - browser/lect_1.sol
Deploy
PUBLISH TO IPFS
OR
At Address
Load contract from Address
Transactions recorded 3
Deployed Contracts
SIMPLESTORAGE AT 0X6B1...1BFD4 (MEMORY)
set
uint256 x
get
Low level interactions
CALLDATA
Transact
Home
lect_1.sol
Lect_1a.sol
3 tabs
1 pragma solidity ^0.4.18;
2
3 contract SimpleStorage {
4 uint storedData;
5
6 function set(uint x) public {
7 storedData = x;
8 }
9
10 function get() public view returns (uint) {
11 return storedData;
12 }
13 }
14
ContractDefinition SolidityTest
0 reference(s)
listen on network
Search with transaction hash or address
[call] from:0x81781E381F7eeC2EFC254D17c0f60070C2a1d9c4 to:SolidityTest.getResult() data:0xde2...92789
Debug
creation of SimpleStorage pending...
[vm] from:0x817...1d9c4 to:SimpleStorage.(constructor) value:0 wei data:0x608...20029 logs:0 hash:0x406...6b9df
Debug

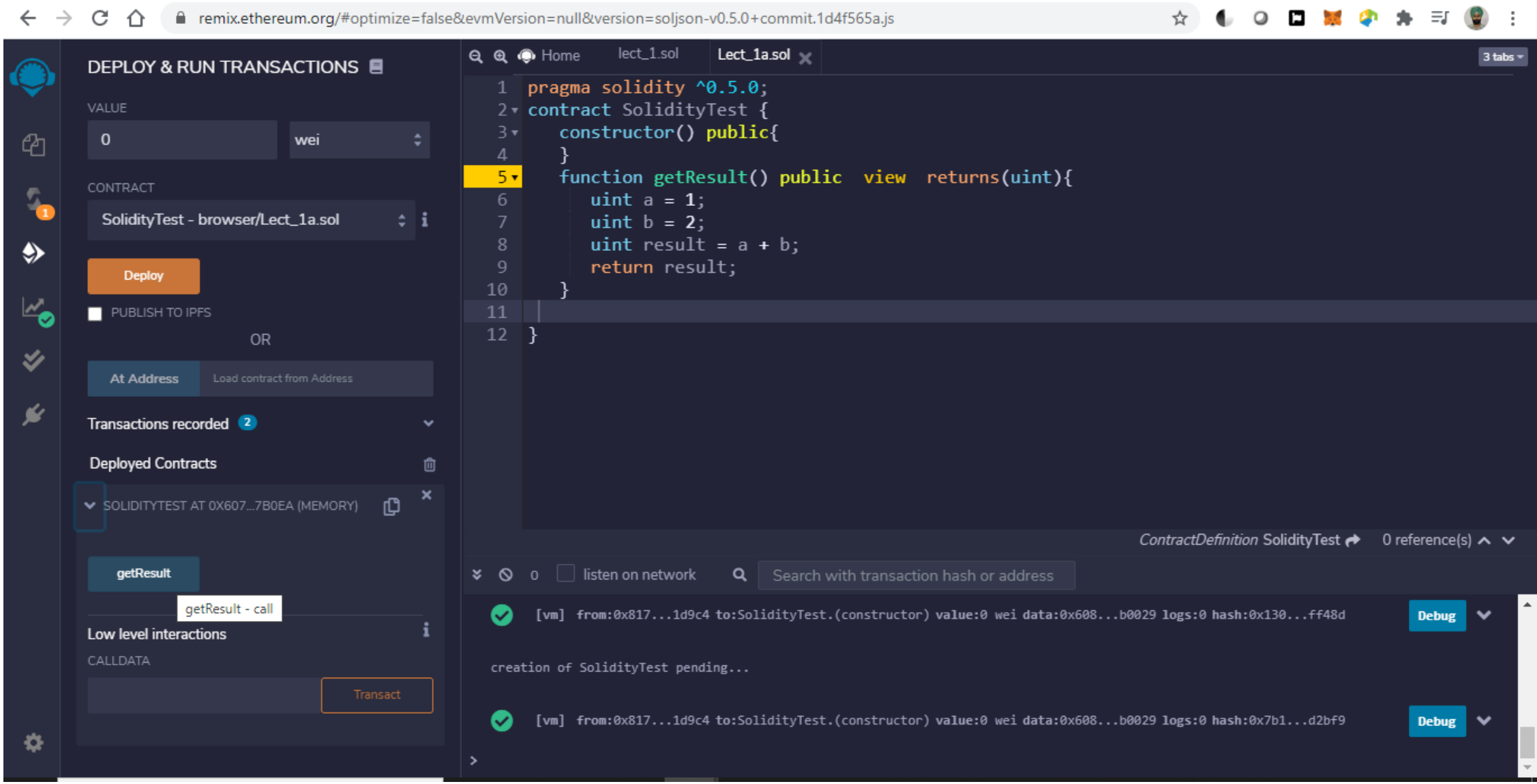


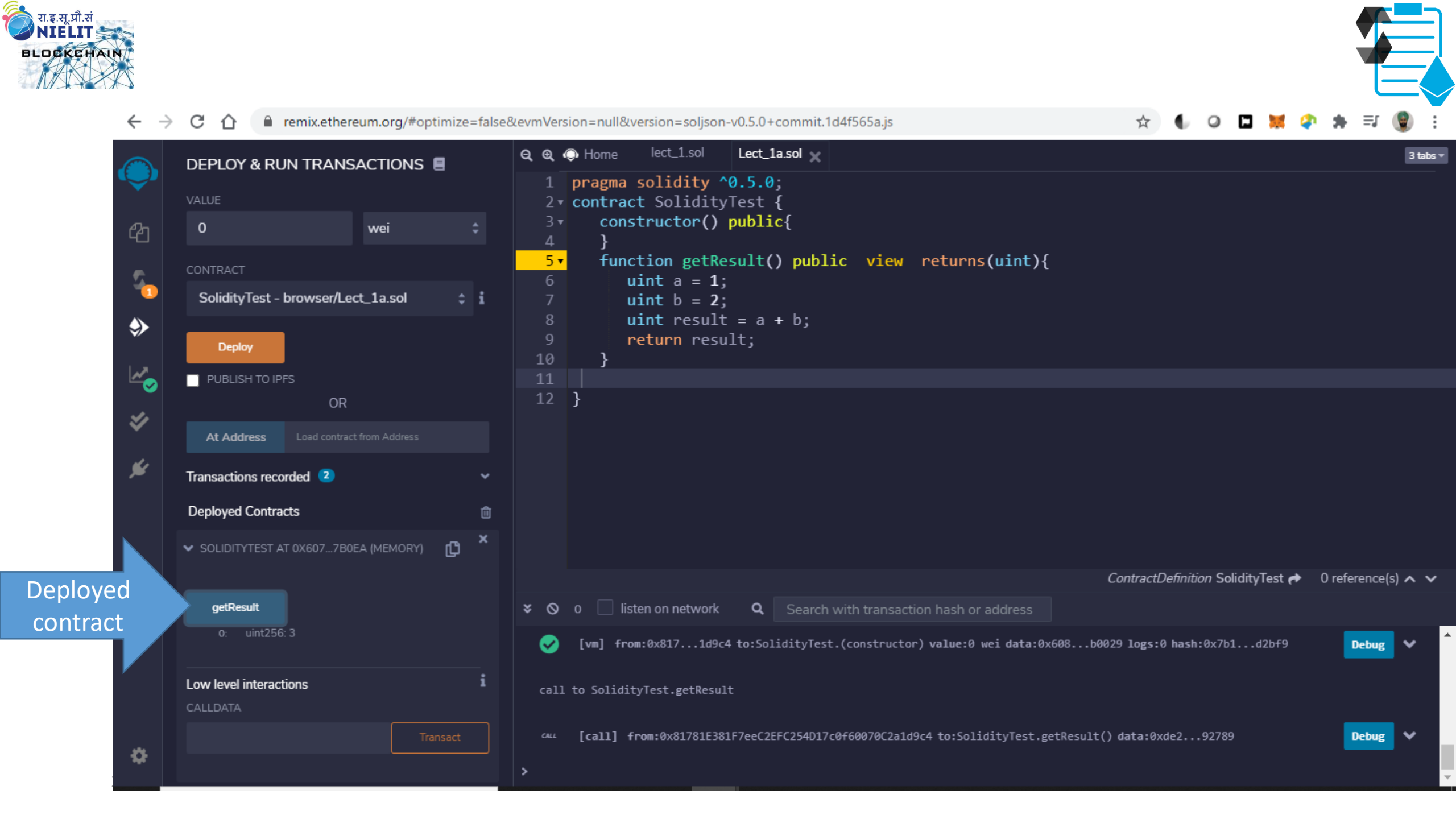
Click Deploy button, to deploy the contract



Deployed contract

Another Example





Deployed contract

Datatype

- Variables are nothing but reserved memory locations to store values.
- By creating a variable we reserve some space in memory.

Type	Keyword	Values
Boolean	bool	true/false
Integer	int/uint	Signed and unsigned integers of varying sizes.
Integer	int8 to int256	Signed int from 8 bits to 256 bits. int256 is same as int.
Integer	uint8 to uint256	Unsigned int from 8 bits to 256 bits. uint256 is same as uint.
Fixed Point Numbers	fixed/unfixed	Signed and unsigned fixed point numbers of varying sizes.
Fixed Point Numbers	fixed/unfixed	Signed and unsigned fixed point numbers of varying sizes.
Fixed Point Numbers	fixedMxN	Signed fixed point number where M represents number of bits taken by type and N represents the decimal points. M should be divisible by 8 and goes from 8 to 256. N can be from 0 to 80. fixed is same as fixed128x18.
Fixed Point Numbers	ufixedMxN	Unsigned fixed point number where M represents number of bits taken by type and N represents the decimal points. M should be divisible by 8 and goes from 8 to 256. N can be from 0 to 80. ufixed is same as ufixed128x18.

Type 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.

Solidity is a statically typed language, which means that the state or local variable type needs to be specified during declaration.

Each declared variable always have a default value based on its type. There is no concept of "undefined" or "null".

Contract : Hello World

- write a read-only function in Solidity
- returns type of a Solidity functions
- pure and public function modifiers
- call a read-only function from outside the smart contract

```
pragma solidity ^0.5.0;

contract HelloWorld {
    function hello() pure public returns(string)
    {
        return `contract - Hello World';
    }
}
```

State Variable

- Variables whose values are permanently stored in a contract storage

```
pragma solidity ^0.5.0;
contract SolidityTest {
   uint storedData;         // State variable
   constructor() public {
      storedData = 10;    // Using State variable
   }
}
```

Local Variable

- Variables whose values are available only within a function where it is defined. Function parameters are always local to that function.

```
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 local variable
   }
}
```

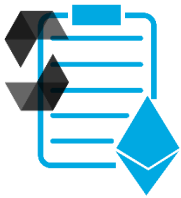
Solidity variable name

- Solidity reserved keywords should not be used as a variable name.
- Solidity variable names should not start with a numeral (0-9). They must begin with a letter or an underscore character. For example, 123test is an invalid variable name but _123test is a valid one.
- Solidity variable names are case-sensitive. For example, Name and name are two different variables.

Scope of variable

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.



```
pragma solidity ^0.5.0;

contract cBase {
    uint public pData = 50;
    uint internal iData = 70;

    function ifun () public returns (uint){
        pData = 10; // internal access
        return pData;
    }

}
contract call_cBase{
    cBase cb = new cBase();
    function show() public view returns(uint){
        return cb.pData();  //external access
    }
}
// Inheritance
contract derived is cBase{
    function dfun () public returns(uint){
        iData = 5; //internal access
        return iData;
    }
    function show()public pure returns(uint){
        uint a=10;
        uint b=20; // local access
        uint result = a+b;
        return result; //access the state variable
    }

}
```

function

- **View** can be used to with a function that does not modify the state but reads state variables.
- **Pure** should be used with functions that neither modify state nor read (access) state variables. They generally perform operations based on input params.
- **Public** to indicate that it can be read from outside the smart contract

```
pragma solidity ^0.4.24;
contract ViewVsPure
{
    uint public age = 18;
    function addToAge(uint _no)
    public view returns (uint)
    { return age + _no; }

    function add(uint _a, uint _b)
    public pure returns (uint)
    { return _a + _b; }
}
```

Operator

- Arithmetic Operators : + , - , * , / , % , ++ , -- ,**(exponent)
- Comparison Operators : == , != , > , < , >= , <=
- Logical (or Relational) Operators : && , || , !
- Bitwise operators : & , | , ^, ~ , << , >> , >>> (Right shift with Zero)
- Assignment Operators : =, +=, *= , -=, /=, %= , ^=
- Same logic applies to Bitwise operators like <<=, >>=, >>=, &=, |= , ^=
- Conditional (or ternary) Operators
  - ? : (Conditional)
  - If Condition is true? Then value X : Otherwise value Y

Decision Making

```
if (expression 1) {
   Statement(s) to be executed if expression 1 is true
} else if (expression 2) {
   Statement(s) to be executed if expression 2 is true
} else if (expression 3) {
   Statement(s) to be executed if expression 3 is true
} else {
   Statement(s) to be executed if no expression is true
}
```

Loops

```
while (expression) {
    Statement(s) to be executed if expression is true
}

do {
    Statement(s) to be executed;
} while (expression);

for (initialization; test condition; iteration statement)
{
    Statement(s) to be executed if test condition is true
}
```

- The **break** statement, which was briefly introduced with the *switch* statement, is used to exit a loop early, breaking out of the enclosing curly braces.
- The **continue** statement tells the interpreter to immediately start the next iteration of the loop and skip the remaining code block. When a **continue** statement is encountered, the program flow moves to the loop check expression immediately and if the condition remains true, then it starts the next iteration, otherwise the control comes out of the loop.

String vs byte32

- Supports both double quote (") and single quote (')

```
string str = “APJ Adbul Kalam”
```

- More preferred way is to use byte types instead of String as string operation requires more gas as compared to byte operation.

```
byte32 str = “APJ Adbul Kalam”
```

Array

- an array can be of compile-time fixed size or of dynamic size.
- Compile time fixed

```
Datatype arrayName [ arraySize ] ;
unit myArr [10];
```

Initializing Array

```
unit myArr [3] = [10,20,30];
unit myArr [ ] = [10,20,30];
myArr[2] = 50 ; // array assignment
```

Array

- Dynamic memory array.

```
uint size = 3;
uint balance[] = new uint[](size);
```

Members :

- **length** – length returns the size of the array. length can be used to change the size of dynamic array be setting it.
- **push** – push allows to append an element to a dynamic storage array at the end. It returns the new length of the array.

```
pragma solidity ^0.5.0;
 contract cTest {
   function testArray() public pure{
    uint len = 7;
    uint[] memory a = new uint[](7);   //dynamic array
    bytes memory b = new bytes(len); //bytes is same as byte[]
    assert(a.length == 7);
    assert(b.length == len);

    a[6] = 8; //access array variable
   assert(a[6] == 8); //test array variable
    uint[3] memory c = [uint(1) , 2, 3];       //static array
    assert(c.length == 3);
  }
}
```

assert and require

- assert (bool condition): abort execution and revert state changes if condition is false (use for internal error)
- require (bool condition): abort execution and revert state changes if condition is false (use for malformed input)

assert() is used to :

- check for overflow/underflow
- check invariants
- validate contract state after making changes
- avoid conditions which should never, ever be possible.
- Generally, you should use assert less often
- Generally, it will be use towards the end of your function.