Blockchain

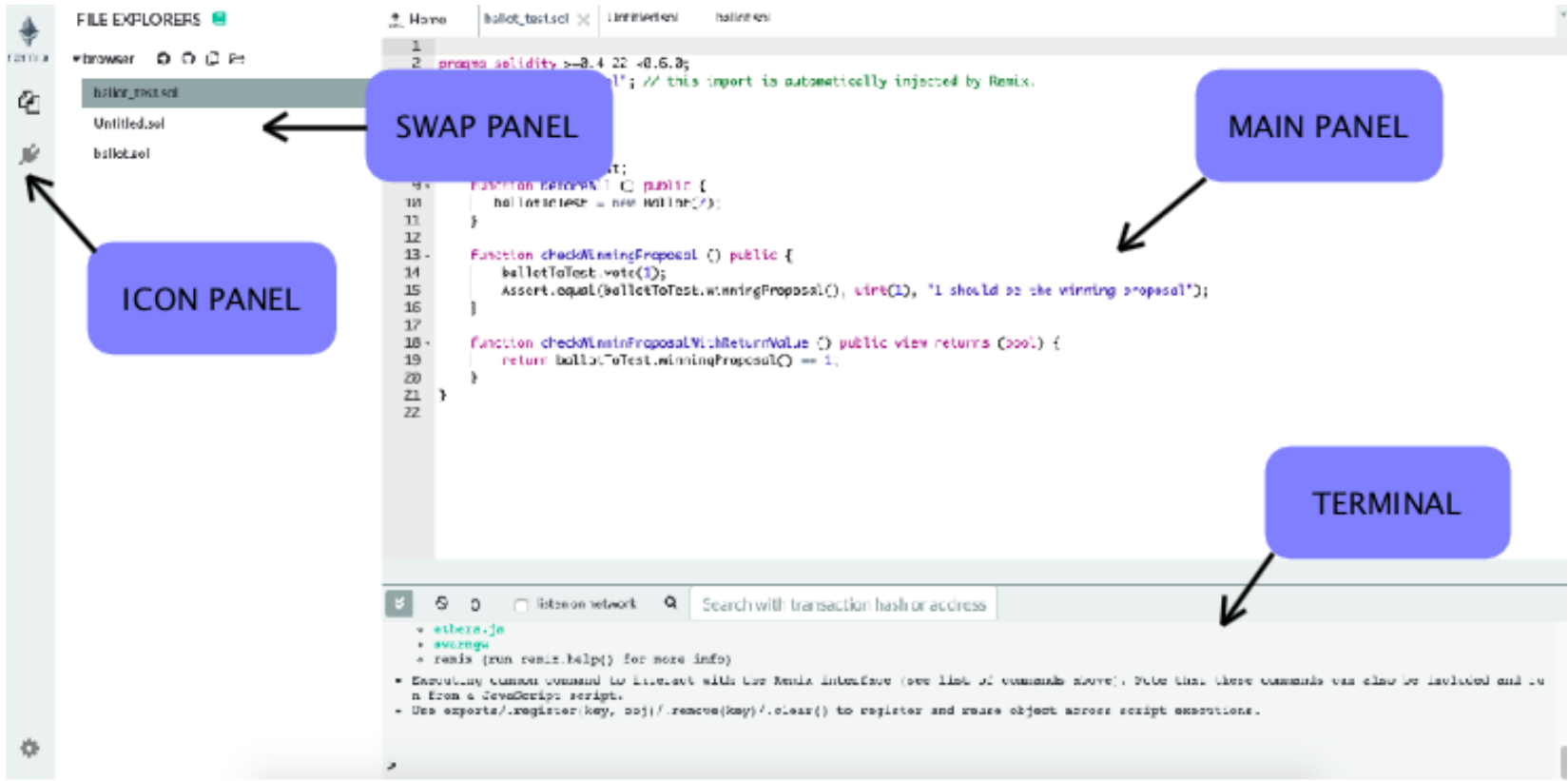
Programming with Solidity

Dr. Sarwan Singh

NIELIT Chandigarh

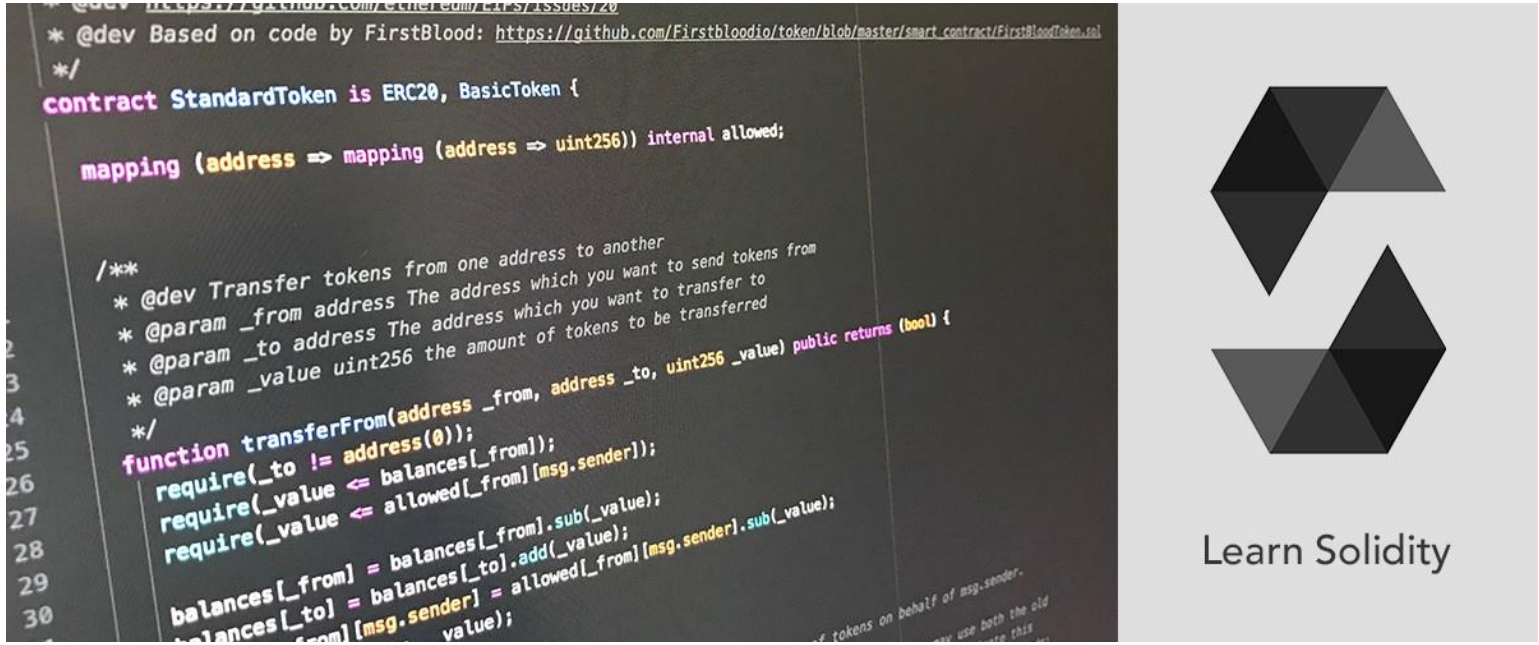
Agenda

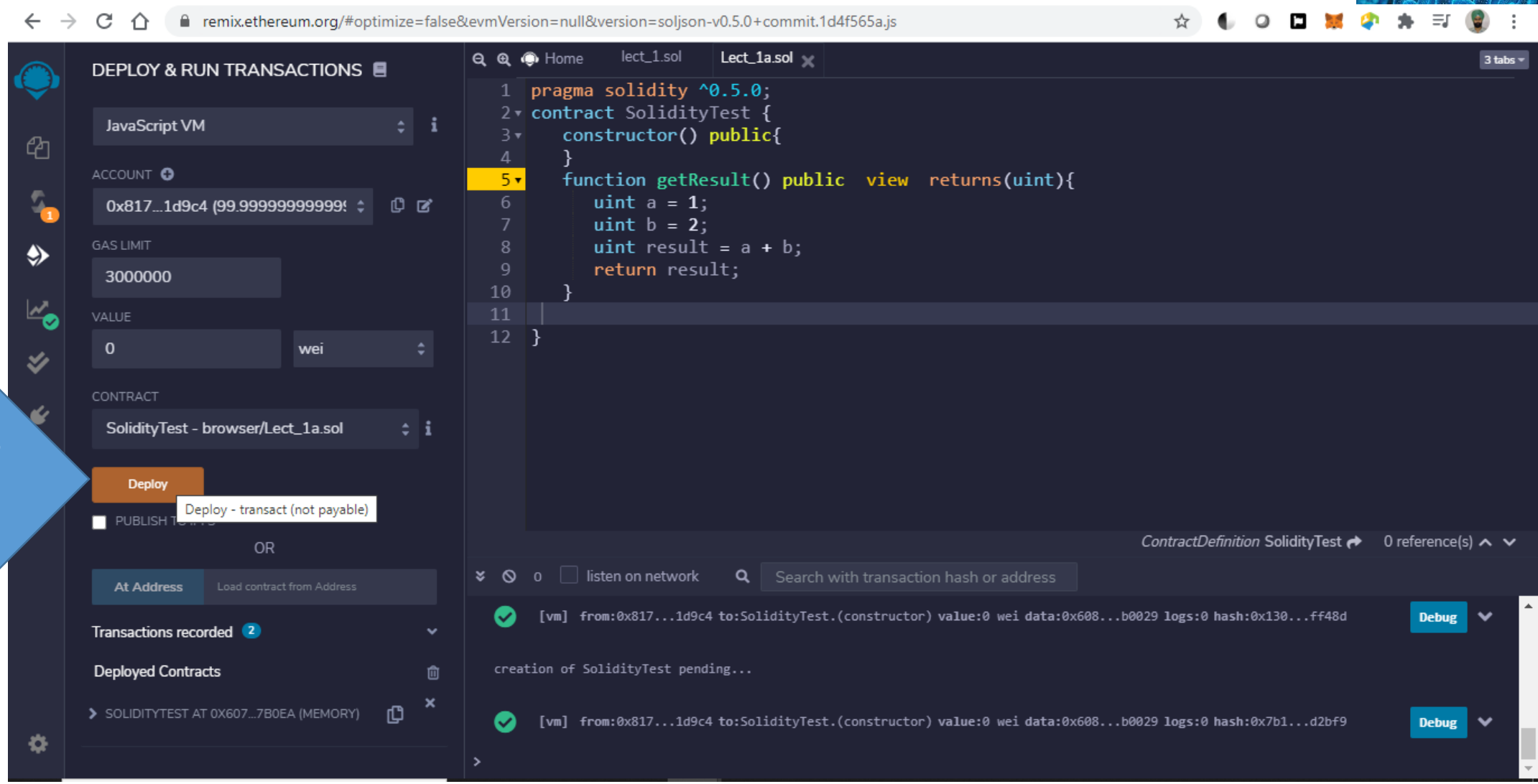
- Solidity programming constructs
- Remix IDE
  - Compile, deploy…
- pragma directive
- Datatype
- Keywords
- Operators



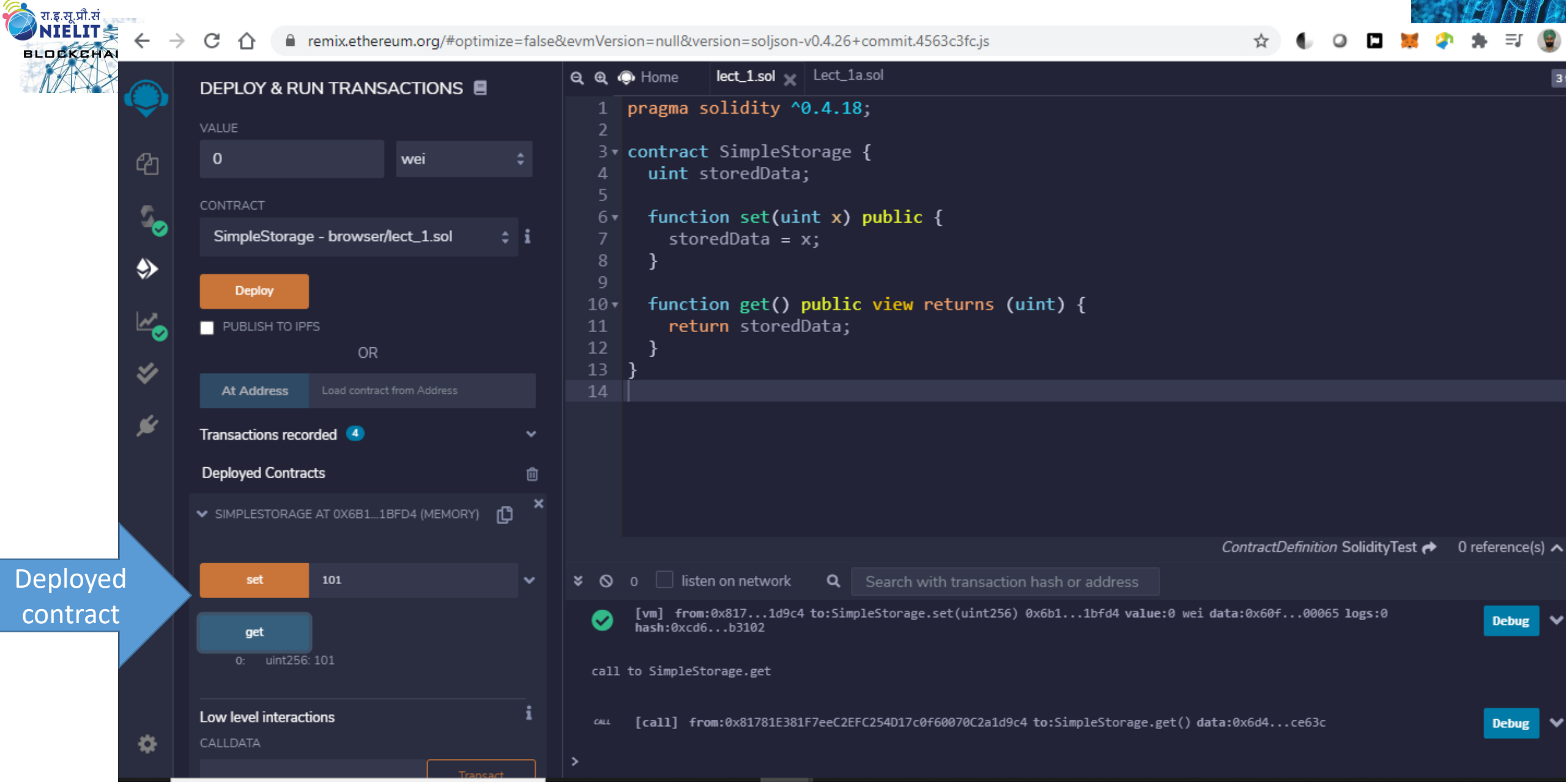
References

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



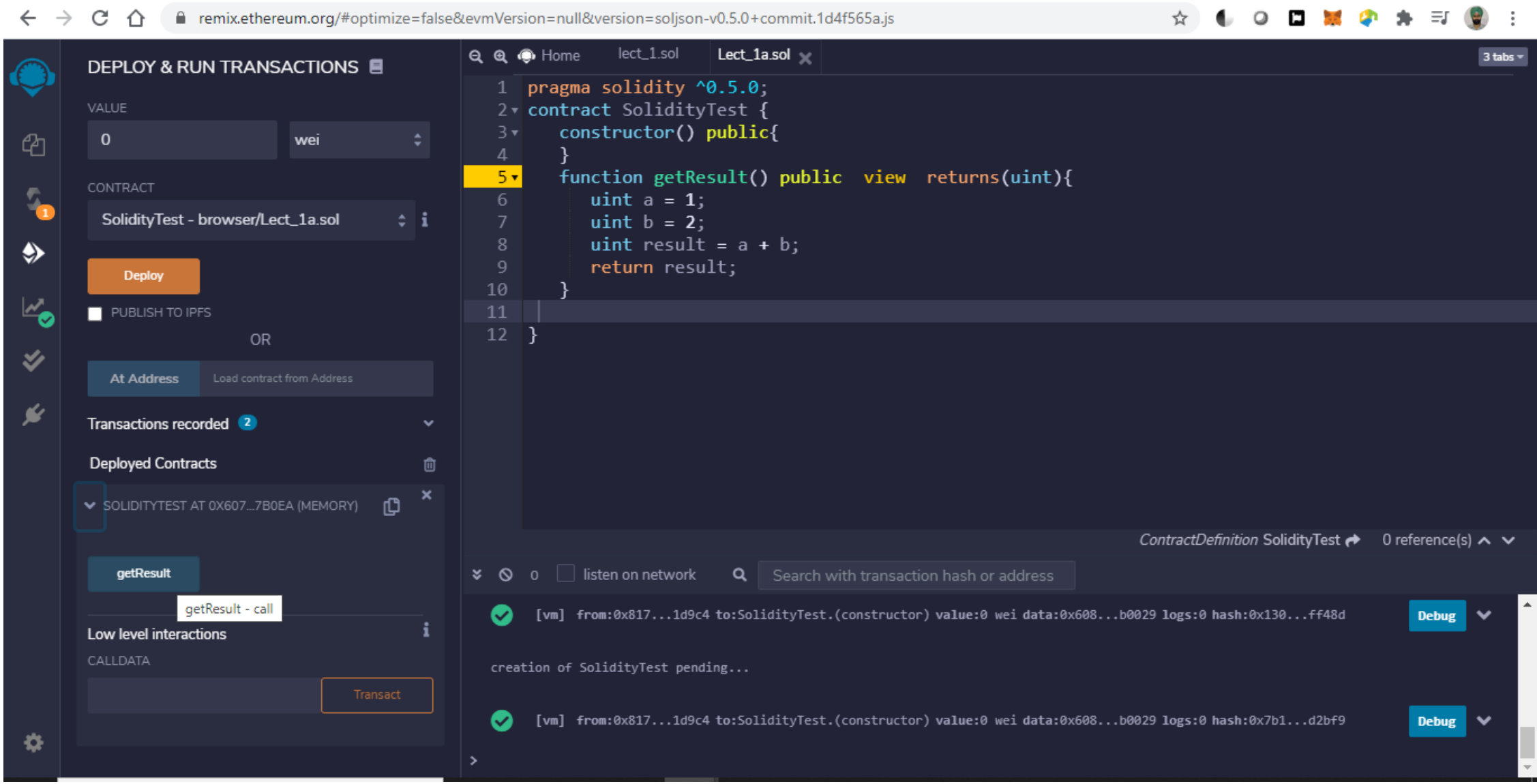


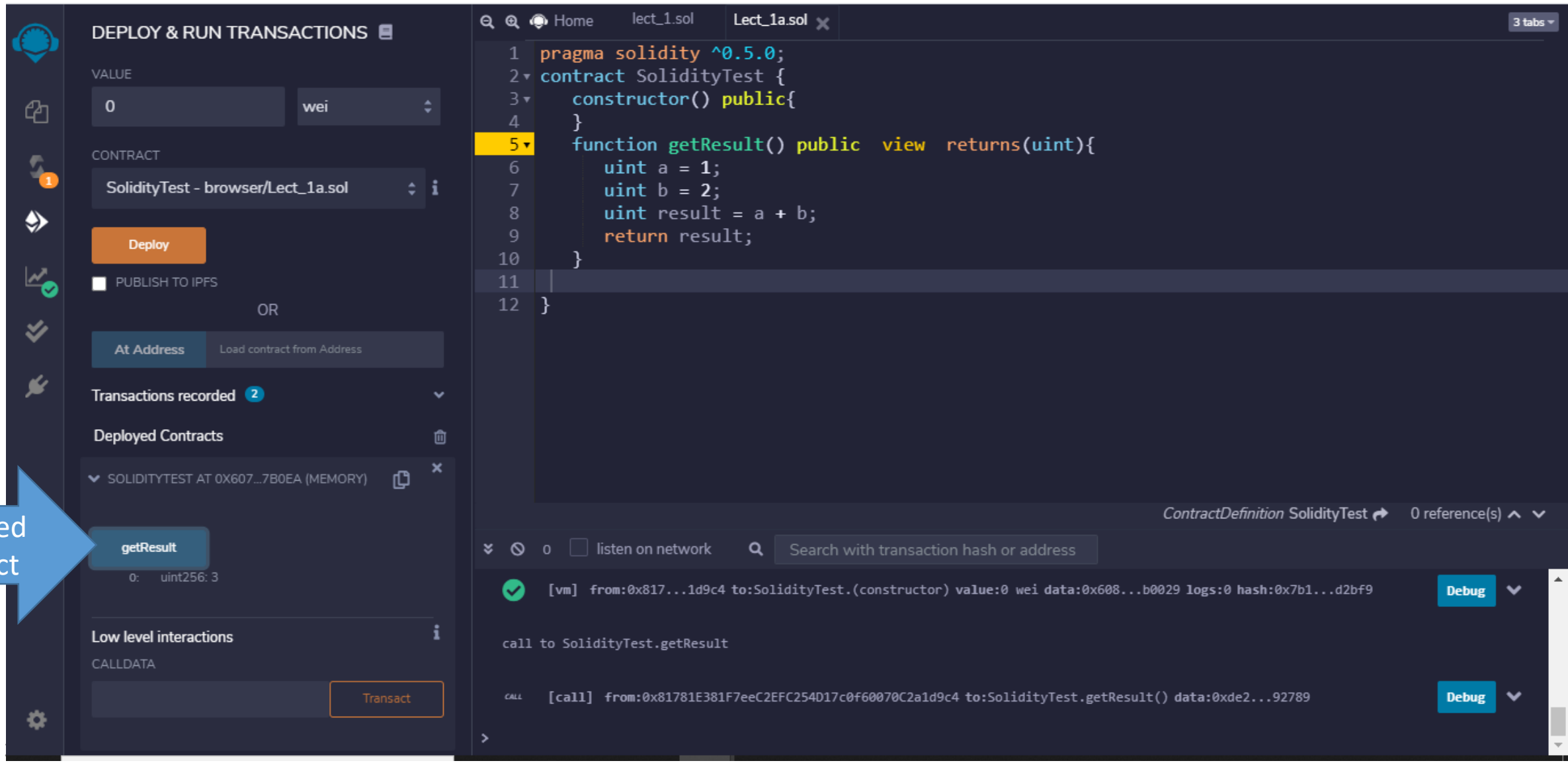
Click Deploy button, to deploy the contract



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.