Blockchain

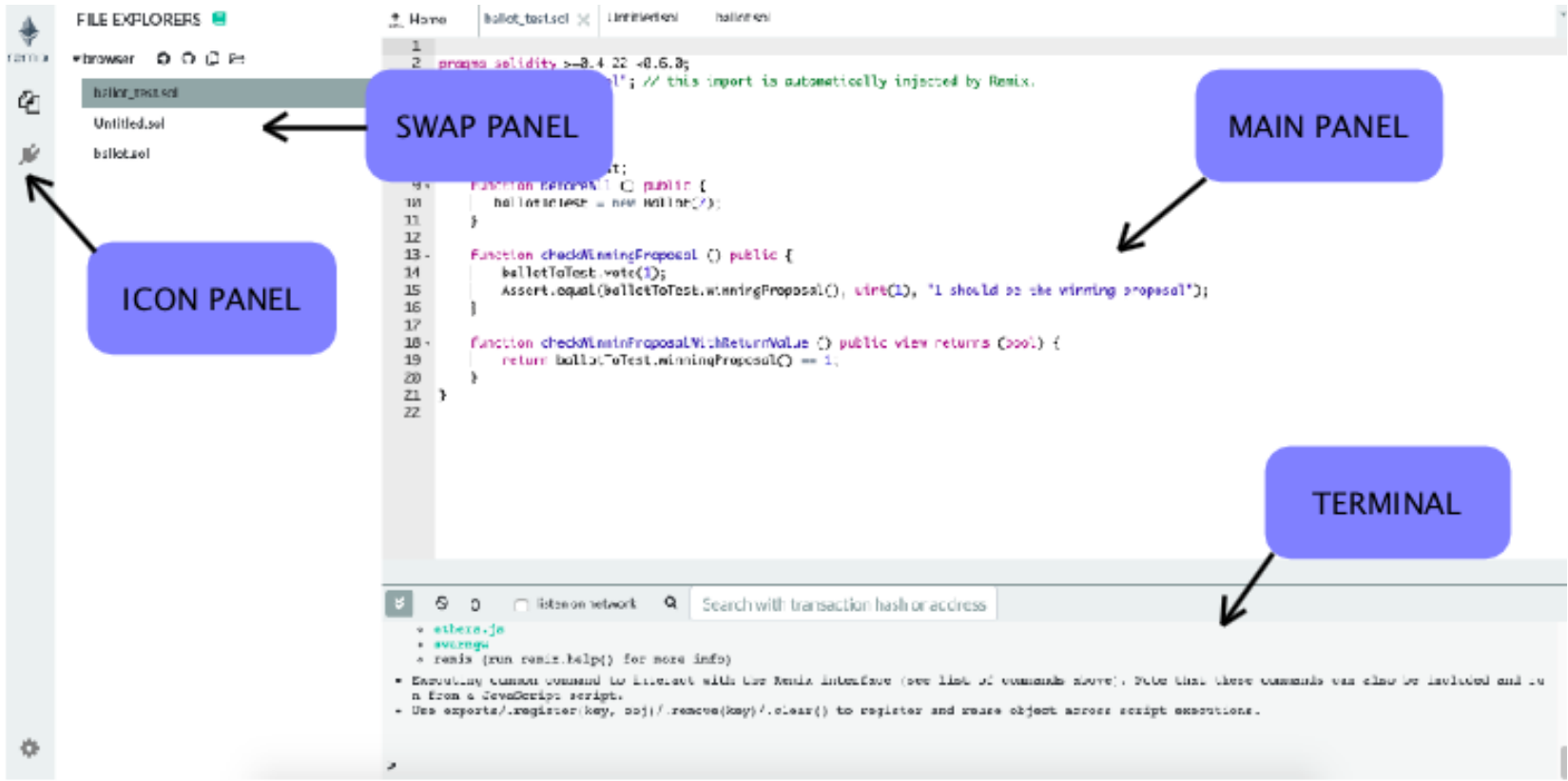
Programming with Solidity

Dr. Sarwan Singh

NIELIT Chandigarh

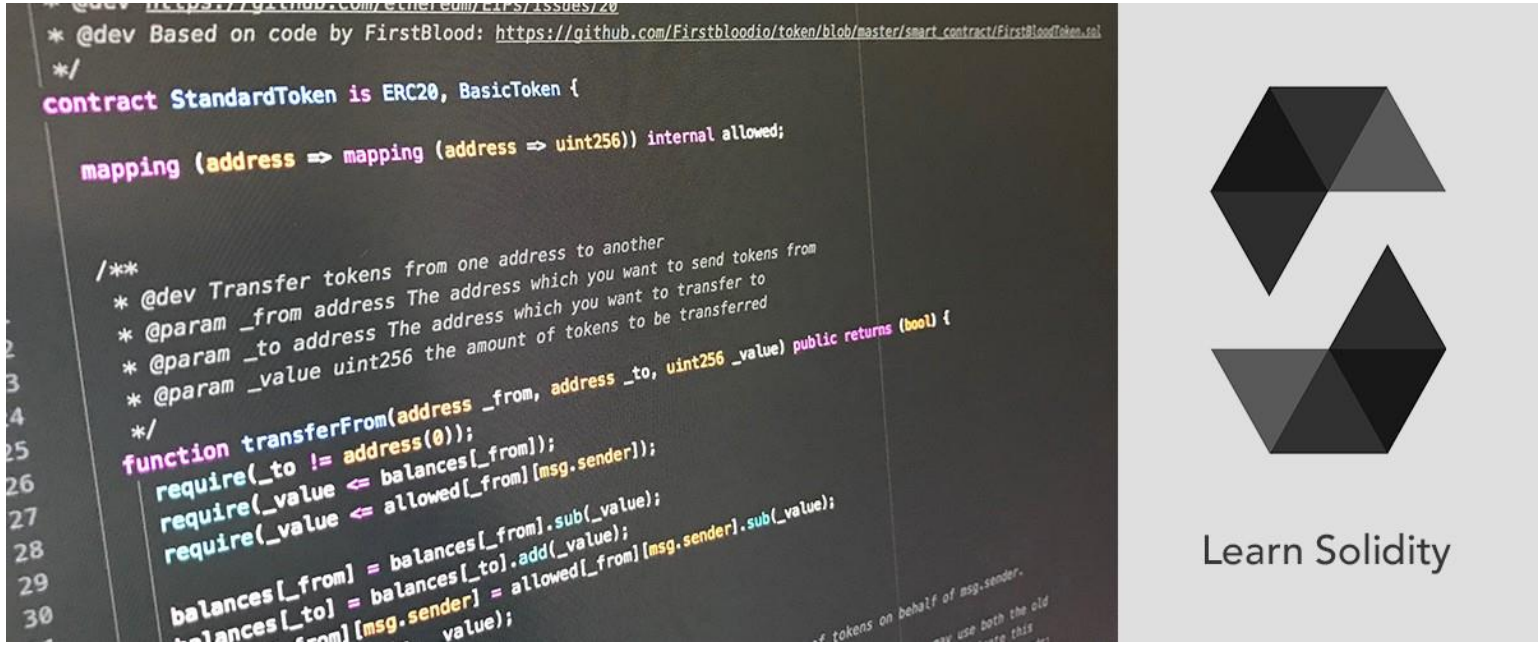
Agenda

- 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



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.
- **pop** - removes an element at the end of the dynamic storage arrays and bytes (not string).

```
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.

Enum

- Enums are one way to create a user-defined type in Solidity.
- restrict a variable to have one of only a few predefined values. The values in this enumerated list are called enums.
- With the use of enums it is possible to reduce the number of bugs in your code.

```
enum ActionChoices { GoLeft, GoRight, GoStraight, SitStill }
ActionChoices choice;
ActionChoices constant defaultChoice = ActionChoices.GoStraight;
```

```
contract test {
    enum ActionChoices { GoLeft, GoRight, GoStraight, SitStill }
    ActionChoices choice;
    ActionChoices constant defaultChoice = ActionChoices.GoStraight;

    function setGoStraight() public {
        choice = ActionChoices.GoStraight;
    }
    function getChoice() public view returns (ActionChoices) {
        return choice;
    }
    function getDefaultChoice() public pure returns (uint) {
        return uint(defaultChoice);
    }
}
```

Structs

```
contract test {
    struct Book {
        string title;
        string author;
        uint book_id;
    }
    Book book;

    function setBook() public {
        book = Book('Learn Java', 'TP', 1);
    }
    function getBookId() public view returns (uint) {
        return book.book_id;
    }
}
```

- Solidity provides a way to define new types in the form of structs
- can basically model any kind of data you want with arbitrary attributes of varying data types

Mapping

- Solidity provides a data structure called a mapping which allows us to store key-value pairs.
- This structure acts much like an associative array or a hash table in other functions.

Mapping

- Mapping is a reference type as arrays and structs.

syntax to declare a mapping type.

```
mapping(_KeyType => _ValueType)
```

  - **_KeyType** – can be any built-in types plus bytes and string. No reference type or complex objects are allowed.
  - **_ValueType** – can be any type.
- Mapping can only have type of **storage** and are generally used for state variables.
- Mapping can be marked public. Solidity automatically create getter for it.

- variables of mapping type are declared using the syntax

```
mapping(_KeyType => _ValueType) _VariableName
```

- The _KeyType can be any built-in value type, bytes, string, or any contract or enum type.
- _ValueType can be any type, including mappings, arrays and structs.

```
pragma solidity 0.5.1;
contract StudentMgt {
    uint256 studentCount = 0;
    struct Student {
        uint _rollno;
        string _name;
        string _courseName;
        uint256 _coureStartDate;
    }
    mapping(uint => Student) public enrolledStudents;
    function enrollStudent(uint _rollno, string memory _name, string memory _courseName,
        uint256   _coureStartDate) public {
        studentCount +=1 ;
        enrolledStudents[studentCount] = Student(_rollno,_name, _courseName, _coureStartDate);
    }
}
```

Student Management contract

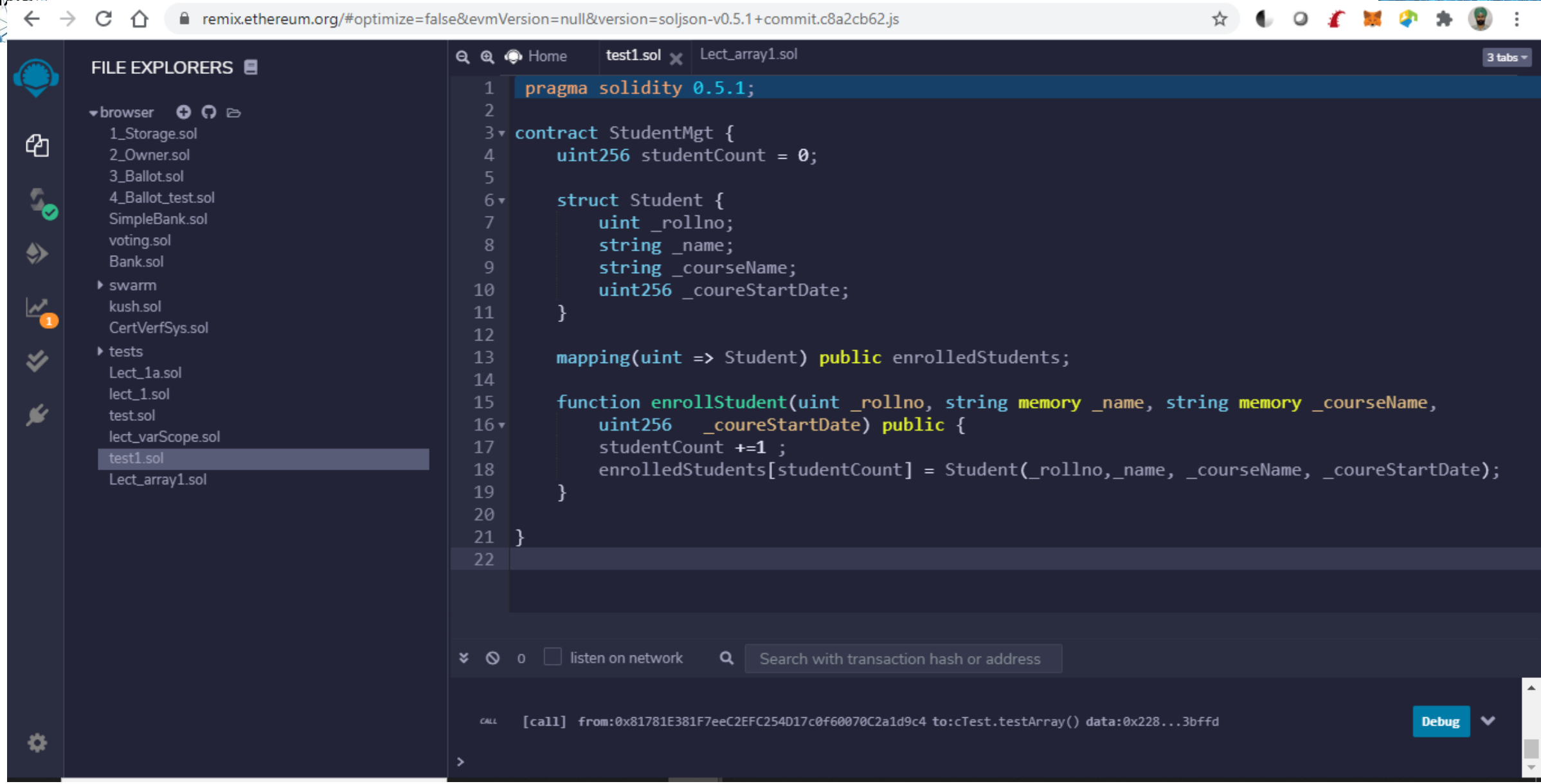
*By Calling the enrollStudent we are saving the data of new student

enrollStudent (1, 'Amit' , 'Python with machine learning', '13-07-2020')

enrollStudent (2, 'Abida' , 'Python with machine learning', '13-07-2020‘)

View the saved data

- Function to show all the data saved – *function showallStudents()*
- Function to see particular student data - *function showStudent(uint rollno)*
- *Function removeStudent(uint rollno)*
- *Function updateStudent(uint rollno, String name, String coursename, coursestartdate)*
- *Funtion showStudentinCourse (String CourseName)*


remix.ethereum.org/#optimize=false&evmVersion=null&version=soljson-v0.5.1+commit.c8a2cb62.js
FILE EXPLORERS
browser
1_Storage.sol
2_Owner.sol
3_Ballot.sol
4_Ballot_test.sol
SimpleBank.sol
voting.sol
Bank.sol
swarm
kush.sol
CertVerfSys.sol
tests
Lect_1a.sol
lect_1.sol
test.sol
lect_varScope.sol
test1.sol
Lect_array1.sol
Home
test1.sol
Lect_array1.sol
3 tabs
1 pragma solidity 0.5.1;
2
3 contract StudentMgt {
4 uint256 studentCount = 0;
5
6 struct Student {
7 uint _rollno;
8 string _name;
9 string _courseName;
10 uint256 _coureStartDate;
11 }
12
13 mapping(uint => Student) public enrolledStudents;
14
15 function enrollStudent(uint _rollno, string memory _name, string memory _courseName,
16 uint256 _coureStartDate) public {
17 studentCount +=1 ;
18 enrolledStudents[studentCount] = Student(_rollno,_name, _courseName, _coureStartDate);
19 }
20
21 }
22
0
listen on network
Search with transaction hash or address
[call] from:0x81781E381F7eeC2EFC254D17c0f60070C2a1d9c4 to:cTest.testArray() data:0x228...3bffd
Debug

```
pragma solidity ^0.5.0;

contract LedgerBalance {
    mapping(address => uint) public balances;

    function updateBalance(uint newBalance) public {
        balances[msg.sender] = newBalance;
    }
}
contract Updater {
    function updateBalance() public returns (uint) {
        LedgerBalance ledgerBalance = new LedgerBalance();
        ledgerBalance.updateBalance(10);
        return ledgerBalance.balances(address(this));
    }
}
```

Contract : Simple storage

- declare a state variable, persisted on the blockchain
- declare a function that can modify data on the blockchain
- use string variables
- memory location

Types of memory locations

In Solidity, they are 4 memory locations:

- **Stack** - most simple memory location, for fixed-sized data (ex: uint) inside functions. It last only as long as the function it is contained in.
- **Memory** - is also short-lived, and is used for variable-length variable data.(spans across multiple function executions in the same smart contract.)
- **Storage** - is the only memory location than span across block, i.e it persists to the blockchain.
- **Calldata** - is only available in the argument of the outer function execution

```
pragma solidity ^0.5.0;

contract SimpleStorage {
  string public data;

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

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

Contract : Advanced storage

- Declare arrays
- Read array elements
- Create new elements in array

```
pragma solidity ^0.5.0;

contract AdvancedStorage {
  uint[] public ids;
  function add(uint id) public {
    ids.push(id);
  }
  function get(uint i) view public returns(uint) {
    return ids[i];
  }
  function getAll() view public returns(uint[] memory)
{
    return ids;
  }
  function length() view public returns(uint) {
    return ids.length;
  }
}
```

Contract : CRUD smart contract

- CRUD is an abbreviation that means “Create, Read, Update, Delete’, the 4 more common kind of operations on data.
- how to declare and use struct in Solidity (custom data)
- how to manage collections of structs in struct arrays

```
pragma solidity ^0.5.0;

contract Crud {
  struct User {
    uint id;
    string name;
  }
  User[] public users;
  uint public nextId = 1;

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

```
function read(uint id) view public
    returns(uint, string memory)
{
    uint j;
    for(uint i = 0; i < users.length; i++) {
      if(users[i].id == id) {
        j = i;
      }
    }
    if(j == 0) {
      revert('User does not exist!');
    }
    return(users[j].id, users[j].name);
}
```

```
function update(uint id, string memory name)
public
{
    uint i = _find(id);
    users[i].name = name;
}
function delete(uint id) public
{
    uint i = _find(id);
    delete users[i];
}
```