//BT 3 SOLDITY ARRAY STRUCTURE

Arrays in Solidity

The array is a special data structure used to create a list of similar type values. The array can be of fixed size and dynamic-sized. With the help of index elements can be accessed easily. below is a sample code to create, and access a fixed-sized array in solidity.

```
pragma solidity >= 0.5.0 < 0.9.0; contract
Array {
 uint [4] public arr = [10, 20, 30, 40];  function
setter(uint index, uint value) public {  arr[index] =
value;
 }

 function length() public view returns(uint) {  return
arr.length;
 }
}
```

You can compile and deploy the code to try changing the array elements with an index and printing the array length.

Creating Dynamic Array

A dynamic array is an array where we can insert any number of elements and delete the details easily using an index. So solidity has functions like push and pops like python, making it easy to create a dynamic array. Below is a code using which you can create a dynamic array. After writing code, compiles and deploy the code by visiting the deploy section in the left-side navigation bar. After that, try inserting and deleting some elements from an array.

```
pragma solidity >= 0.5.0 < 0.9.0; contract
Array { uint [] public arr; function
PushElement(uint item) public {
 arr.push(item);
 }
 function Length() public view returns(uint) {  return
arr.length;
 }
 function PopElement() public {
 arr.pop();
}
}
```

Structure in Solidity

The structure is a user-defined data type that stores more than one data member of different data types. As in array, we can only store elements of the same data type, but in structure, you can keep elements of different data types used to create multiple collections. The structure can be made outside and inside the contract storage, and the Structure keyword can be used to declare the form. The structure is storage

type, meaning we use it in-store only, and if we want to use it in function, then we need to use the memory keyword as we do in the case of a string.

```
pragma solidity >= 0.5.0 < 0.9.0;
```

```
struct Student {  uint
rollNo;
 string name; }

contract Demo {  Student
public s1;
 constructor(uint _rollNo, string memory _name) {  s1.rollNo =
_rollNo;
 s1.name = _name;
 }
 // to change the value we have to implement a setter function  function
changeValue(uint _rollNo, string memory _name) public {
 Student memory new_student = Student( {
 rollNo : _rollNo,
 name : _name
 });
 s1 = new_student;
 }
}
Fallback:
pragma solidity ^0.4.0; //
Creating a contract
contract fback
{
// Declaring the state variable uint
x;
// Mapping of addresses to their balances
mapping(address => uint) balance; //
Creating a constructor
constructor() public
{
// Set x to default
// value of 10 x=10;
}
// Creating a function
function SetX(uint _x) public returns(bool)
{
// Set x to the
// value sent x=_x;

return true; }

// This fallback function
// will keep all the Ether
function() public payable
{
balance[msg.sender] += msg.value;
```

```
}
}
// Creating the sender contract
contract Sender
{
function transfer() public payable
{
// Address of Fback contract address
_receiver =
0xbcD310867F1b74142c2f5776404b6bd97165FA56;
// Transfers 100 Eth to above contract
_receiver.transfer(100);
}
}
```

Create a Smart Contract with CRUD Functionality

We have excellent theoretical and hands-on practical knowledge about solidity, and now you can create a primary smart contract like hello world, getter, and setter contracts. So it’s a great time to try making some functional smart contracts, and the best way to try all the things in one code is to create one program that performs all CRUD operations. A sample smart contract code to create ERC20 tokens pragma solidity ^0.4.0; import "./ERC20.sol"; contract myToken is ERC20{ mapping(address =>uint256) public amount; uint256 totalAmount; string tokenName; string tokenSymbol; uint256 decimal;

```
constructor() public{ totalAmount =
10000 * 10**18;
amount[msg.sender]=totalAmount;
tokenName="Mytoken";
tokenSymbol="Mytoken";
decimal=18; }
function totalSupply() public view returns(uint256){ return
totalAmount;
}
function balanceOf(address to_who) public view
returns(uint256){ return amount[to_who];
}
function transfer(address to_a,uint256 _value) public
returns(bool){ require(_value&amp;lt;=amount[msg.sender]);
amount[msg.sender]=amount[msg.sender]-_value;
amount[to_a]=amount[to_a]+_value; return
true;
}
}
```