



BLOCKCHAIN TECHNOLOGY LAB

(20CP406P)

LAB ASSIGNMENT - 6



**B.Tech in Computer Science and Engineering Dept.,
Pandit Deendayal Energy University,
Gandhinagar**



Name: Vrushank Ariwala

Roll No.: 19BCP141

Branch: CSE

❖ **Aim:**

Learn Syntactical details of Solidity through simple Smart Contracts

❖ **Introduction:**

Smart Contract:

A smart contract is a small program consist of collections of code(functions) and data(states) that runs on an Ethereum blockchain.

Once the smart contract is deployed on the Ethereum blockchain, it cannot be changed.

To deploy the smart contract to Ethereum, you must pay the ether (ETH) cost.

Understand it as a digital agreement that builds trust and allows both parties to agree on a particular set of conditions that cannot be tampered with.

Solidity:

Solidity Contracts are like a class in any other object-oriented programming language. They firmly contain data as state variables and functions which can modify these variables. When a function is called on a different instance (contract), the EVM function call happens and the context is switched in such a way that the state variables are inaccessible. A contract or its function needs to be called for anything to happen. Some basic properties of contracts are as follows:

- **Constructor:** Special method created using the constructor keyword, which is invoked only once when the contract is created.
- **State Variables:** These are the variables that are used to store the state of the contract.
- **Functions:** Functions are used to manipulate the state of the contracts by modifying the state variables.

Writing a Smart Contract:

Creating contracts programmatically is generally done by using JavaScript API web3.js, which has a built-in function web3.eth.Contract to create the contracts. When a contract is created its constructor is executed, a constructor is an

optional special method defined by using the constructor keyword which executes one per contract. Once the constructor is called the final code of the contract is added to the blockchain.

- Syntax:

```
contract <contract_name>{  
    constructor() <visibility>{  
        .....  
    }  
    // rest code  
}
```

- Simple Smart Contract:

```
// SPDX-License-Identifier: UNLICENSED  
pragma solidity >=0.4.17 <0.9.0;  
  
contract Storage {  
    uint data;  
  
    function set(uint newData) public {  
        data = newData;  
    }  
  
    function get() public view returns (uint) {  
        return data;  
    }  
}
```

Because smart contracts' source code is often readily available to read, it's a good idea to specify the license of your code in the first line after SPDX-License-Identifier:

Following that, the pragma directive tells the compiler which version of Solidity to use. The versions start with 0. to indicate that breaking changes are to be expected in minor, regular updates. Our smart contract can be compiled against version 0.4 or higher, but not version 0.9.

- Creating a Constructor:

A Constructor is defined using a constructor keyword without any function name followed by an access modifier. It's an optional function which initializes state variables of the contract. A constructor can be either internal or public, an internal constructor marks contract as abstract.

Syntax:

```
constructor() <Access Modifier> {  
}
```

- Creating a Variable:

In Solidity declaration of variables is a little bit different, to declare a variable the user has to specify the data type first followed by access modifier.

Syntax:

```
<type> <access modifier> <variable name> ;
```

Example:

```
int public int_var;
```

Types:

Solidity is a statically typed language i.e. each declared variable always has a default value based on its data type, which means there is no concept of 'null' or 'undefined'. Solidity supports three types of variables:

1. State Variables: Values of these variables are permanently stored in the contract storage. Each function has its own scope, and state variables should always be defined outside of that scope.

Example:

```
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;
}
}

```

```

decoded output      {
                      "0": "uint8: 16"
                      }

```

2. Local Variable: Values of these variables are present till the function executes and it cannot be accessed outside that function. This type of variable is usually used to store temporary values.

Example:

```

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;
}
}

```

```

decoded output      {
                    "0": "uint256: 3"
                    }

```

- Creating a Function

A function is basically a group of code that can be reused anywhere in the program, which generally saves the excessive use of memory and decreases the runtime of the program. Creating a function reduces the need of writing the same code over and over again. With the help of functions, a program can be divided into many small pieces of code for better understanding and managing.

In Solidity a function is generally defined by using the function keyword, followed by the name of the function which is unique and does not match with any of the reserved keywords. A function can also have a list of parameters containing the name and data type of the parameter. The return value of a function is optional but in solidity, the return type of the function is defined at the time of declaration.

Syntax:

```

function <function name>(<parameters type>) \[function type\] [returns
(<return type>)] {}

```

The commonly used function types are public, external, private, internal, view, pure, and payable.

Here's how the types that govern the visibility of the function work:

- public: anyone can call the function
- external: anyone but the contract can call the function. Using the external type instead of public can have a performance boost and potentially save a lot of gas
- private: only the contract holding the function can call it

- internal: the contract and its derivatives can call the function

These types govern access to the state:

- view: the function only reads the state
- pure: the function neither reads nor writes to the state

Note that payable is used when the function can accept payment when it's called.

Example:

```
pragma solidity ^0.5.0;

// Creating a contract
contract Test {

    // Defining function to calculate sum of 2 numbers
    function add() public view returns(uint){
        uint num1 = 10;
        uint num2 = 16;
        uint sum = num1 + num2;
        return sum;
    }
}
```

```
num1: 10 uint256
num2: 16 uint256
sum: 26 uint256
```

- Function Calling

A function is called when the user wants to execute that function. In Solidity the function is simply invoked by writing the name of the function where it has to

be called. Different parameters can be passed to function while calling, multiple parameters can be passed to a function by separating with a comma.

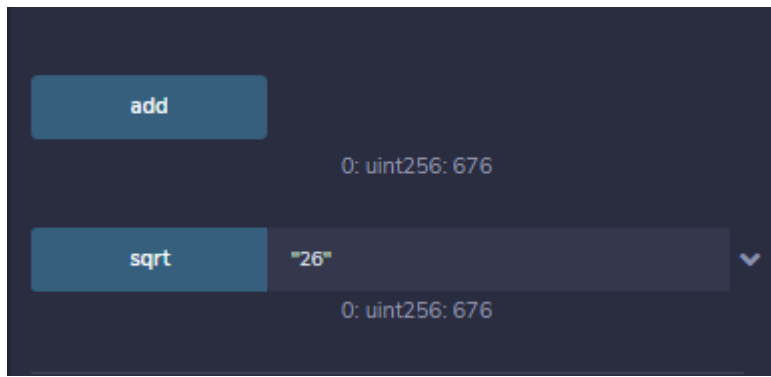
Example:

```
pragma solidity ^0.5.0;

// Creating a contract
contract Test {

    // Defining a public view function to demonstrate
    // calling of sqrt function
    function add() public view returns(uint){
        uint num1 = 10;
        uint num2 = 16;
        uint sum = num1 + num2;
        return sqrt(sum); // calling function
    }

    //Defining public view sqrt function
    function sqrt(uint num) public view returns(uint){
        num = num ** 2;
        return num;
    }
}
```

Solidity Datatypes:

Solidity is a statically typed language, which implies that the type of each of the variables should be specified. Data types allow the compiler to check the correct usage of the variables. The declared types have some default values called Zero-State, for example for bool the default value is False. Likewise other statically typed languages Solidity has Value types and Reference types which are defined below:

- **Value Types**

Value-type variables store their own data. These are the basic data types provided by solidity. These types of variables are always passed by value. The variables are copied wherever they are used in function arguments or assignments. Value type data types in solidity are listed below:

- **Boolean:** This data type accepts only two values True or False.
- **Integer:** This data type is used to store integer values, int and uint are used to declare signed and unsigned integers respectively.
- **Fixed Point Numbers:** These data types are not fully supported in solidity yet, as per the Solidity documentation. They can be declared as fixed and unfixed for signed and unsigned fixed-point numbers of varying sizes respectively.
- **Address:** Address hold a 20-byte value which represents the size of an Ethereum address. An address can be used to get balance or to transfer a balance by balance and transfer method respectively.
- **Bytes:** Although bytes are similar to strings, there are some differences between them. bytes used to store a fixed-sized character set while the string is used to store the character set equal to or more than a byte. The length of bytes is from 1 to 32, while the string has a dynamic length.

Byte has the advantage that it uses less gas, so better to use when we know the length of data.

- **Enums:** It is used to create user-defined data types, used to assign a name to an integral constant which makes the contract more readable, maintainable, and less prone to errors. Options of enums can be represented by unsigned integer values starting from 0.

Example:

```
pragma solidity ^ 0.5.0;

// Creating a contract
contract Types {

    // Initializing Bool variable
    bool public boolean = false;

    // Initializing Integer variable
    int32 public int_var = -60313;

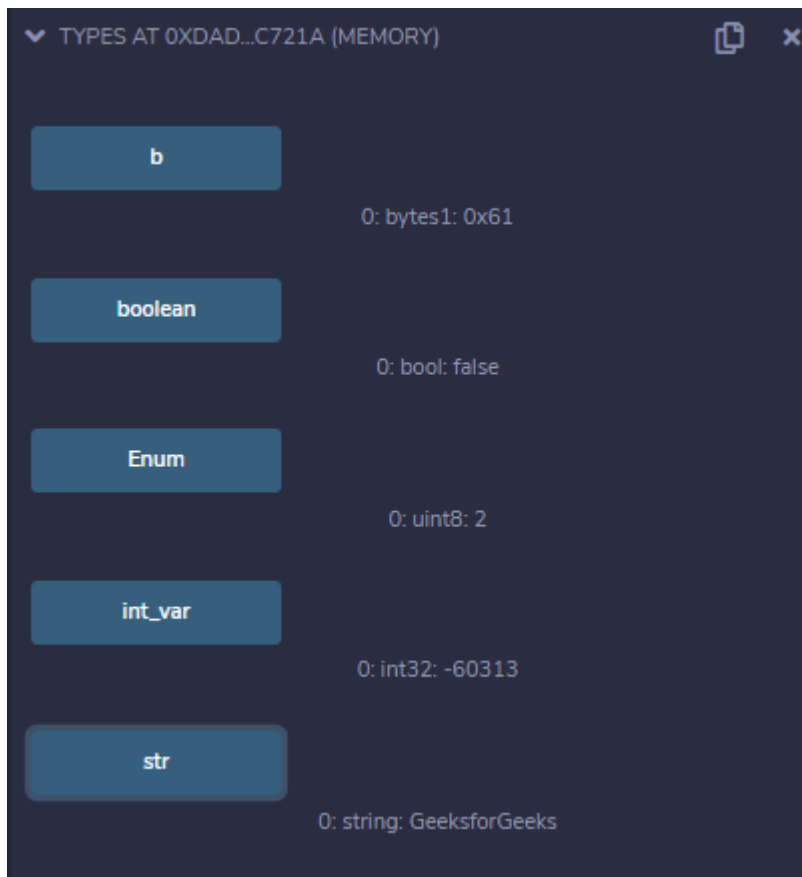
    // Initializing String variable
    string public str = "GeeksforGeeks";

    // Initializing Byte variable
    bytes1 public b = "a";

    // Defining an enumerator
    enum my_enum { geeks_, _for, _geeks }

    // Defining a function to return
    // values stored in an enumerator
```

```
function Enum() public pure returns(  
    my_enum) {  
    return my_enum._geeks;  
}  
}
```



- **Reference Types**

Reference type variables store the location of the data. They don't share the data directly. With the help of reference type, two different variables can refer to the same location where any change in one variable can affect the other one.

Reference types in solidity are listed below:

- **Arrays:** An array is a group of variables of the same data type in which the variable has a particular location known as an index. By using the index location, the desired variable can be accessed. The array size can be fixed or dynamic.
- **Strings:** Strings are like arrays of characters. When we use them, we might occupy bigger or shorter storage space.

- **Struct:** Solidity allows users to create and define their own type in the form of structures. The structure is a group of different types even though it's not possible to contain a member of its own type. The structure is a reference type variable that can contain both value type and reference type
- **Mapping:** Mapping is the most used reference type, that stores the data in a key-value pair where a key can be any value type. It is like a hash table or dictionary as in any other programming language, where data can be retrieved by key.

Example:

```
pragma solidity ^0.4.18;

// Creating a contract
contract mapping_example {

    // Defining an array
    uint[5] public array
        = [uint(1), 2, 3, 4, 5] ;

    // Defining a Structure
    struct student {
        string name;
        string subject;
        uint8 marks;
    }

    // Creating a structure object
    student public std1;

    // Defining a function to return
```

```

// values of the elements of the structure
function structure() public view returns(
    string memory, string memory, uint){
    std1.name = "John";
    std1.subject = "Chemistry";
    std1.marks = 88;
    return (
        std1.name, std1.subject, std1.marks);
    }
// Creating a mapping
mapping (address => student) result;
address[] student_result;
}

```

```

▼ array: uint256[5]
  length: 5
  0: 1 uint256
  1: 2 uint256
  2: 3 uint256
  3: 4 uint256
  4: 5 uint256
▼ std1: struct mapping_example.student
  name: "John" string
  subject: "Chemistry" string
  marks: 88 uint8
▶ result: mapping(address => struct mapping_example.student)
▶ student_result: address[]

```

Smart Contract in Remix IDE:



```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract BankingEG{

    int balance;

    constructor(){
        balance=50;
    }

    function deposit(int amt) public{
        balance=balance+amt;
    }
    function withdrawl(int amt) public{
        balance=balance-amt;
    }
    function checkbal() public view returns(int){
        return balance;
    }
}
```

▼ BANKINGEG AT 0XF8E...9FBE8 (ME)  

Balance: 0 ETH

deposit

50

▼


withdrawl

int256 amt

▼

checkbal

0: int256: 100

Low level interactions 

CALLDATA

Transact

