Pakistan Blockchain Institute

PAKISTAN BLOCKCHAIN INSTITUTE

MODULE-2 BLOCKCHAIN AND SMART CONTRACT BASICS

Class-5

Raja Rizwan Saleem
Lead Blockchain Trainer

Edversity.

Variables in Solidity - Practical Examples

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

Variables in Solidity - Practical Examples

```
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
   }
}
```

Variables in Solidity - Practical Examples

```
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 storedData; //access the state variable
   }
}
```

2. Functions

Functions

1. Functions are the heart of Ethereum blockchain and Solidity.
2. Ethereum maintains the current state of state variables and executes transaction to change values in state variables.
3. When a function in a contract is called or invoked, it results in the creation of a transaction.
4. Functions are the mechanism to read and write values from/to state variables.

Functions

5. Functions are a unitof code that canbe executed on-demand by calling it.
6. Functions canaccept parameters, execute its logic, and optionally return values to the caller.
7. Solidity provides named functions with the possibility of only one unnamed function in a contract called the fallback function.

Functions

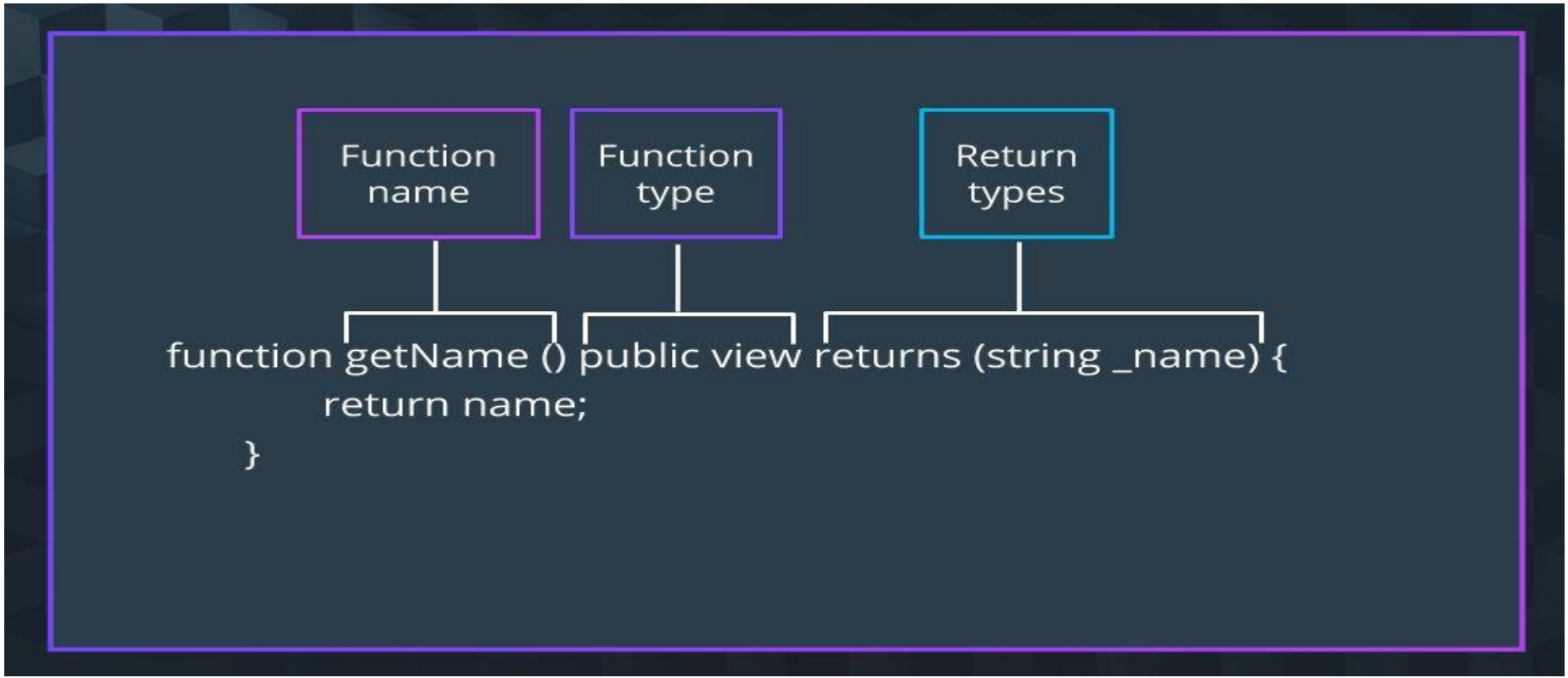
A function is basically a group of code that can be reused anywhere in the program, which generally saves the excessive use of memory and decreases the runtime of the program. Creating a function reduces the need of writing the same code over and over again. With the help of functions, a program can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

Functions Syntax

1. Define the function with the function keyword
2. Create a name for the function, which is unique and does not coincide with any of the reserved keywords
3. List any parameters containing the name and data type of the parameter or include no extra parameters
4. Create a statement block surrounded by curly brackets

```
function function-name(parameter-list) scope returns() {
        // statements
}
```

Functions Syntax

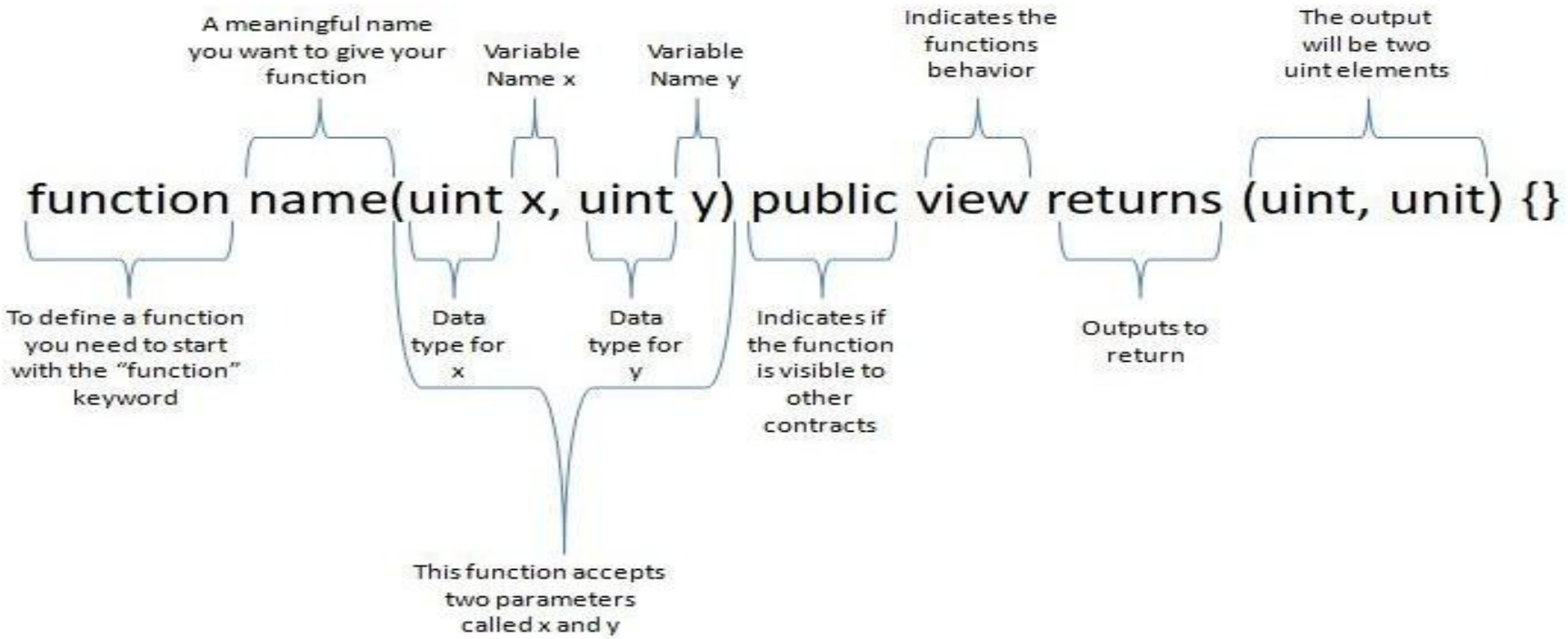


Functions Syntax

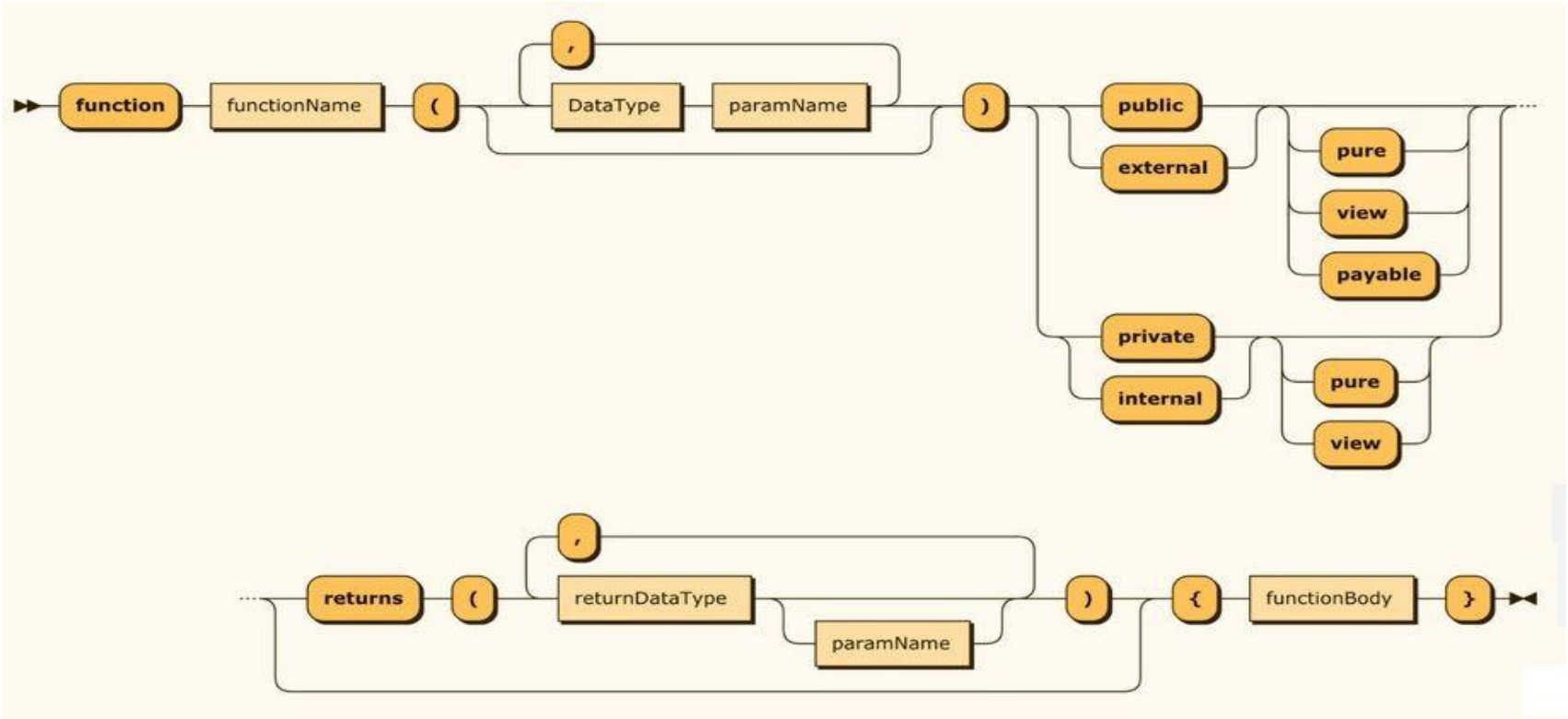
Basic Syntax

```
function setAge(int a) public{
}
function getAge(int a) public returns (int){
        return 45;
}
```

Functions Syntax



Functions Syntax



Functions Example-1

```
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;
}
}
```

Functions Example-1

```
// Solidity program to demonstrate
// function declaration
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;
}}
```

Functions Example-2

```
pragma solidity ^0.5.0;


contract Test {
    function getResult() public view returns(uint){
        uint a = 1; // local variable
        uint b = 2;
        uint result = a + b;
        return result;
    }
}
```

Function Visibility Qualifiers

1. Functions has visibility qualifier associated with them similar to state variables
    a. public
    b. internal
    c. private
    d. external

Function Visibility Qualifiers

public - Anyone can call this function
private - Only this contract can call this function.
view - This function returns data and does not modify the contract's data
constant - This function returns data and does not modify the contract's data
pure - Function will not modify or even read the contract's data
payable - When someone call this function they might send ether along

Function Visibility Qualifiers -- public

1. public: This visibility makes function access directly from outside. They become part of the contracts interface and can be called both internally and externally.

```
function setAge(int a) public{
}
```

Function Visibility Qualifiers -- internal

1. internal: This visibility means that this function can only be used within the current contract and any contract that inherits from it. These functions cannot be accessed from outside. They are not part of the contracts interface.

```
function setAge(int a) internal{
}
```

Function Visibility Quali fiers -- private

1. private: Private functions can only be used in contracts declaring them. They cannot be used even within derived contracts. They are not part of the contracts interface.

```
function setAge(int a) private{
}
```

Function Visibility Qualifiers -- external

1. external: This visibility makes function access directly from externally but not internally. These functions become part of the contracts interface.

```
function setAge(int a) external{
}
```

Function Behavior Qualifiers

1. Functions can also have additional qualifiers that change their behavior in terms of having the ability to change contract state variables
    a. view
    b. pure
    c. payable

Function Behavior Qualifiers -- view

1. These functions do not have the ability to modify the state of blockchain. They can read the state variables and return back to the caller
2. They **cannot** modify any variable, invoke an event, create another contract, call other functions that can change state, and so on.
3. Think of view functions as functions that can read and return current state variable values.

Function Behavior Qualifiers -- view

1. Things that modify state:
   a. Writing to state variables
   b. Emitting events
   c. Creating other contracts
   d. Using self destruct
   e. Sending Ether via calls
   f. Calling any function not marked view or pure
   g. Using low-level calls
   h. Using inline assembly that contains certain opcodes

Function Behavior Qualifiers -- view

```
int age = 45;

function getAge() public view returns (int) {
    // Not allowed to change state variables
    // age = 10;
    return age;

}
```

Function Behavior Qualifiers -- pure

1. Pure functions further constraints the ability of functions. Pure functions can neither read and write—in short, they cannot access state variables. Functions that are declared with this qualifier should ensure that they will not access the current state and transaction variables.

Function Behavior Qualifiers -- pure

1. The additional activities not allowed in pure functions
   a. Reading from state variables
   b. Accessing this.balance or <address>.balance
   c. Accessing any of the members of block, tx, and msg (with the exception of msg.sig and msg.data)
   d. Calling any function not marked pure

Function Behavior Qualifiers -- pure

```
int age = 45;

function getAge() public pure returns (int) {
// Not allowed to access or modify state variables
    // int a  = age;
    // return age;
    return 5;

}
```

Function Behavior Qualifiers -- payable

1. Functions declared with the payable keyword has ability to accept Ether from the caller. The call will fail in case Ether is not provided by sender. A function can only accept Ether if it is marked as payable.
2. Contract which have payable function will receive Ether so when we call function and provide Ether so its contract will receive that Ether. (E-Commerce)

Function Behavior Qualifiers -- payable

3. No need to write any logic inside function just mark function as payable and send Ether to that function, it will be added in contract's balance

Function Behavior Qualifiers -- payable

```
function receivePayment() public payable {
    // No logic needed
}
```

Function Behavior Qualifiers -- payable

```
uint public myBalance;


function receivePayment() public payable {
    //msg.value contains ether from sender
   //we can store value in state variable
    myBalance += msg.value;
}
```

Function Behavior Qualifiers in Practice

```
pragma solidity ^0.8.1;

contract First{

    uint age = 56;

    function doSomeWork() public view {

    }

    function getAge() public view returns (uint) {
        uint a = age * 2;
        doSomeWork ();
        return age;
    }

    }
```

Function Behavior Qualifiers in Practice

```
pragma solidity ^0.5.0;
contract Test {

function receivePayment() public payable {

}

function checkBalance() public view returns (uint) {
     return address (this).balance;
}

}
```

Function Returns and Return

1. Returning data is an integral part of a Solidity function
2. Solidity provides **two** different syntaxes for returning data from a function
3. In first one is function mention datatype in returns without naming return variable name
4. In second case function mention datatype and variable name in returns

Function Returns with datatype only

1. Function mention datatype in returns without naming return variable name
2. In this case developer need to use return keyword explicitly to return value from function
3. If we will not use return keyword to return then function will return default value of datatype
    a. 0 in case of int or uint
    b. false in case of bool

Function Returns with datatype only

```
uint age = 45;
function getAge() public view returns (uint){
    return age; // 45

}
function getAge1() public pure returns (uint){
    // it will returns 0
}
```

Function Returns with datatype and variable name

1. Function can mention datatype and variable name in returns
2. In such cases, developers can directly use and return variable from a function without using the return keyword
3. Variable mentioned with datatype in returns will directly returns the value

Function Returns with data type and variable name

```
function getAge() public pure returns (uint){
    a = 50;

    // returns 50
}
function getAge1() public pure returns (uint a){
    // it will returns 0
}
```

Function Returns in Practice

```
pragma solidity ^0.8.1;

contract First{

    function doSomeWork() public view returns (uint a) {
        a = 45;
    }

     function doSomeWork1() public view returns (string memory) {
        return "Lets code";
    }

    function doSomeWork3() public view returns (bool) {

    }
}
```

Function Returns multiple values

1. In solidity function can returns multiple values
2. In multiple return value case it will return tuple
3. A tuple is a custom data structure consisting of multiple variables

Function Returns multiple values

```
function getValues() public pure returns (uint, bool){
    uint age = 65;                                  //tuple
    bool isFeePaid = true;
    return (age, isFeePaid);

}
function getValues1() public pure returns (uint a,
bool b){
    // it will returns 0 and false
}
```

Function Returns multiple values

```
function getValues2() public pure returns (uint a,
bool b){

    a = 12;
    b = true;
    // it will returns 12 and true
}
```

Function Returns multiple values

```
pragma solidity ^0.5.0;
contract Test {
// Defining a public view function to
// demonstrate return statement
function return_example() public view returns(uint, uint, uint,
string memory){
    uint num1 = 10;
    uint num2 = 16;
    uint sum = num1 + num2;
    uint prod = num1 * num2;
    uint diff = num2 - num1;
    string memory msg = "Multiple return values";
    return (sum, prod, diff, msg);
}}
```

Function calling

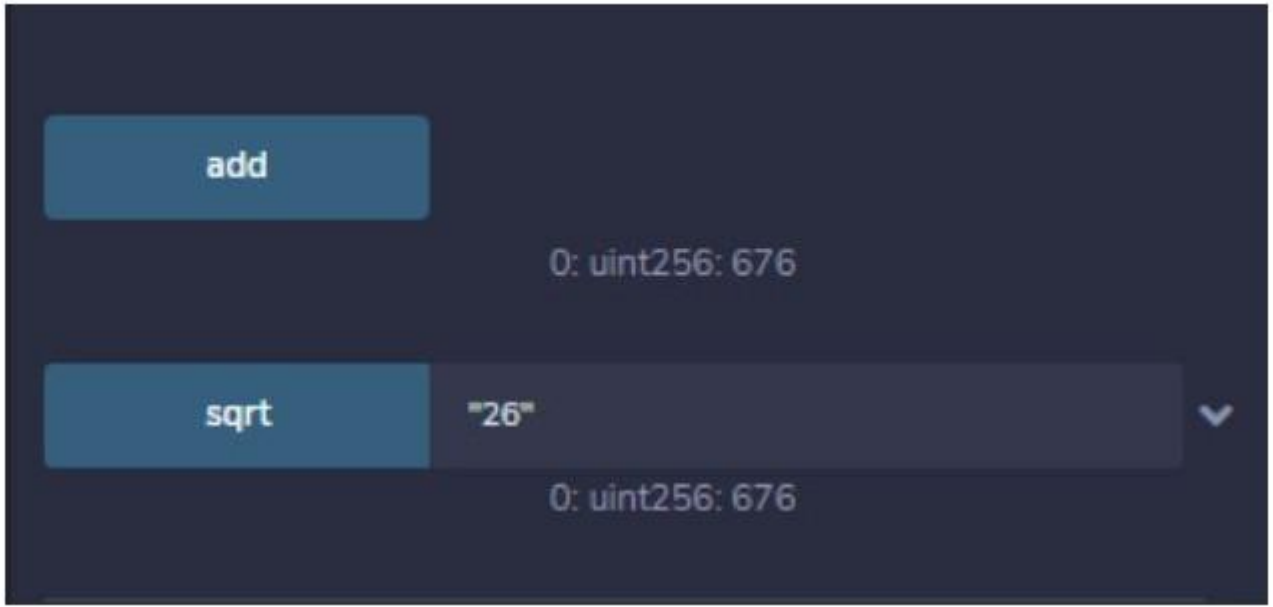
A function is called when the user wants to execute that function. In Solidity the function is simply invoked by writing the name of the function where it has to be called. Different parameters can be passed to function while calling, multiple parameters can be passed to a function by separating with a comma.

Function calling

```
// Solidity program to demonstrate
// function calling
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;
   }
}
```

Function calling

Output :



Thanks

End of Lecture-2 (Module-5)