BITS, Pilani -Pilani Campus
CS F452 (Blockchain Technology)
Laboratory 2 (Week-2)

AIM:

To explore foundational Solidity concepts required for developing and deploying smart contracts on Ethereum.

Topics to be covered:

1. Introduction to Solidity
2. Basic Smart Contract Structure
3. Comments in Solidity
4. Variable Types: State, Local, and Global
5. Primitive Data Types
6. Operators
7. Control Flows
8. Functions in Solidity
9. Function Visibility and State Mutability
10. A Walkthrough Combined Code

What is Solidity?

Solidity is a high-level, object-oriented language that's used to create smart contracts that run on the Ethereum Virtual Machine (EVM). Smart contracts are programs that govern how accounts behave on the Ethereum state.

What a Smart Contract consist of ?

```
// put// SPDX-License-Identifier: MIT
pragma solidity ^0.8.8.0;

contract Tea {
    uint public qtyTeaCups;

    // Get the current Tea Cup quantity
    function get() public view returns (uint) {
        return qtyTeaCups;
    }

    // Increment Tea Cup quantity by 1
    function increment() public {
        qtyTeaCups += 1; // same as  qtyTeaCups = qtyTeaCups + 1;
    }

    // Function to decrement count by 1
    function decrement() public {
        qtyTeaCups -= 1; // same as  qtyTeaCups = qtyTeaCups - 1;
        // What happens if qtyTeaCups = 0 when this func is called?
    }
} your code here
```

First line Comment

```
// put// SPDX-License-Identifier: MIT
```

It clearly states that the code is licensed under the MIT License, making it easier for others to understand how they can use, modify, and distribute the code. Although code will compile(since it’s a comment) without this line also , but it's good practice to write it.

Comments in Solidity

// way to add a comment in Solidity: Any single comment in .sol file should start with prefix ‘//’.

/* Multi

Line

comment*/: Any multi line comment in .sol file should be enclosed with /* */.

Any Solidity file must have the first line of code as a pragma directive that tells the compiler which compiler version it should use to convert the human-readable Solidity code to machine readable bytecode.

```
pragma solidity ^0.8.8.0;
```

```
contract Tea {
    uint public qtyTeaCups;

    // Get the current Tea Cup quantity
    function get() public view returns (uint) {
        return qtyTeaCups;
    }

    // Increment Tea Cup quantity by 1
    function increment() public {
        qtyTeaCups += 1; // same as  qtyTeaCups = qtyTeaCups + 1;
    }

    // Function to decrement count by 1
    function decrement() public {
        qtyTeaCups -= 1; // same as  qtyTeaCups = qtyTeaCups - 1;
        // What happens if qtyTeaCups = 0 when this func is called?
    }
} your code here
```

The keyword (contract) tells the compiler that you're declaring a Smart Contract. Contracts are objects that hold data as a combination of variables and functions.

```
uint public qtyTeaCups;
```

In the above example the variable qtyTeaCups is called a "state variable". It holds the contract's state - which is the technical term for data that the program needs to keep track of to operate.

Declaration of Variables :

1. State variables
2. Local variables
3. Global variables

State variables: Persistent variables stored in blockchain and retains its value between function calls and transactions. Depending upon the access modifier , we restrict or give access to this variable.

Syntax: <type> <access modifier> <variable name> ;

Example: `uint8 public state_var;`

Local variables: Temporary, used within functions, and doesn't persist.There are no access modifiers for local variables because their scope is limited to the function or block in which they are declared.

Syntax: <type> <variable name> = <value>;

Example: uint local_var1 = 1;

Global variables: Global variables are built-in variables provided by Solidity that give information about the current state of the blockchain or the transaction.

Syntax:They do not have a type or access modifier syntax like user-defined variables. You simply reference them directly in your code.

Example:

`msg.sender`: The address of the account that called the current function.

`block.timestamp`: The current block timestamp (in seconds).

```
pragma solidity ^0.8.0;

contract PrimitiveDataTypes {

    // Boolean: Can be true or false
    bool public isActive = true;

    // Integer (signed int): Stores positive and negative integers
    int public temperature = -15;
    uint transaction_ID = 100012;


    // Unsigned Integer (uint): Stores only positive integers
    uint public totalSupply = 1000;
    uint8 public smallUint = 255; // uint8 ranges from 0 to 255

    // Address: Holds Ethereum addresses (20 bytes)
    address public owner = msg.sender;

    // Byte Fixed Array: Stores a fixed number of bytes
    bytes1 public smallByte = 0x01;  // single byte (byte)
    bytes32 public largeByte = 0x1234567890abcdef1234567890abcdef1234567890abcdef1234567890abcdef;

    // String: Stores a sequence of characters
    string public greeting = "Hello, Solidity!";

    // Enum: Custom type with fixed possible values
    enum Status { Pending, Shipped, Delivered }
    Status public orderStatus = Status.Pending;

    // Function to demonstrate usage
    function updateData() public {
        isActive = !isActive;          // Toggling the boolean
        temperature += 10;             // Modifying signed integer
        totalSupply += 500;            // Modifying unsigned integer
        orderStatus = Status.Shipped;  // Updating Enum value
        transaction_ID += 1; // Updating Trans. ID
    }
}
```

State Var

Local Var

Global Var

Primitive data type table

Data Type	Description	# Size (Bits)	Example
Unsigned Integers			
uint	Unsigned integer (0 and positive)	256	uint256 public myUint;
uint8	Unsigned integer (0 to 2^8-1)	8	uint8 public myUint8;
uint16	Unsigned integer (0 to 2^16-1)	16	uint16 public myUint16;
uint32	Unsigned integer (0 to 2^32-1)	32	uint32 public myUint32;
uint64	Unsigned integer (0 to 2^64-1)	64	uint64 public myUint64;
uint128	Unsigned integer (0 to 2^128-1)	128	uint128 public myUint128;
uint256	Unsigned integer (0 to 2^256 - 1)	256	uint256 public myUint256;
Signed Integers			
int	Signed integer (positive and negative)	256	int256 public myInt;
int8	Signed integer (-2^7 to 2^7-1)	8	int8 public myInt8;
int16	Signed integer (-2^15 to 2^15-1)	16	int16 public myInt16;
int32	Signed integer (-2^31 to 2^31-1)	32	int32 public myInt32;
int64	Signed integer (-2^63 to 2^63-1)	64	int64 public myInt64;
int128	Signed integer (-2^127 to 2^127-1)	128	int128 public myInt128;
int256	Signed integer (-2^255 to 2^255 - 1)	256	int256 public myInt256;
Boolean	True or false	8	bool public isActive;
Address	Holds a 20-byte Ethereum address	160	address public owner;
Bytes	Fixed-size byte arrays		
bytes1	1 byte	8	bytes1 public myBytes1;
bytes2	2 bytes	16	bytes2 public myBytes2;
bytes3	3 bytes	24	bytes3 public myBytes3;
bytes32	32 bytes	256	bytes32 public myBytes32;
bytes	Dynamic-size byte array	Variable	bytes public dynamicBytes;

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

contract PrimitiveDataTypes {

    // Boolean: Can be true or false
    bool public isActive = true;

    // Integer (signed int): Stores positive and negative integers
    int public temperature = -15;

    // Unsigned Integer (uint): Stores only positive integers
    uint public totalSupply = 1000;
    uint8 public smallUint = 255; // uint8 ranges from 0 to 255

    // Address: Holds Ethereum addresses (20 bytes)
    address public owner = msg.sender;

    // Byte Fixed Array: Stores a fixed number of bytes
    bytes1 public smallByte = 0x01;  // single byte (byte)
    bytes32 public largeByte = 0x1234567890abcdef1234567890abcdef1234567890abcdef1234567890abcdef;

    // String: Stores a sequence of characters
    string public greeting = "Hello, Solidity!";

    // Enum: Custom type with fixed possible values
    enum Status { Pending, Shipped, Delivered }
    Status public orderStatus = Status.Pending;

    // Function to demonstrate usage
    function updateData() public {
        isActive = !isActive;           // Toggling the boolean
        temperature += 10;              // Modifying signed integer
        totalSupply += 500;             // Modifying unsigned integer
        orderStatus = Status.Shipped;   // Updating Enum value
    }
}
```

Each primitive data type example is declared at the beginning, with sample values, and updateData() demonstrates some operations on these values.

All the variables store and allow manipulation of different types of values and use data types to declare themselves like in the above snippet isActive , temperature , totalSupply etc.

1. **bool(Boolean) :**A `bool` is a data type that can hold either `true` or `false` and is used for logical statements. It's used to define a flag or status indicator.

   Eg: isActive variable is acting as flag in the above code

2. **int(Signed Integer):** An int data type is used to store both positive and negative whole numbers.We can specify the number of bits required for that variable and accordingly use int8,int 16... int256. If you just write int , by default its assumed 256 bits.

   Eg: temperature is given this data type since it can have both values.

3. **uint(Unsigned Integer):** An uint data type is used when we are sure that the variable will only have positive values. Similar to int , here also we can mention bits required.

   Eg:totalSupply can never be negative, smallUnit specifies uint8.

4. **address:**This type stores Ethereum addresses (20-byte values), which represent accounts or contracts. It's essential in Solidity for identifying other accounts.

   Eg:owner's address will be stored using this data type.

5. **byte**: These types are used to store a fixed number of bytes in hexadecimal form. They're often used for lower-level data or identifiers.

Eg: `smallByte` is a single byte, while `largeByte` is a 32-byte fixed array. These can hold data or unique identifiers.

6. **string:**`string` is a dynamic array of characters, used for storing text or messages.

   Eg:`greeting` holds the string `"Hello, Solidity!"`, which could display a message or status in the contract.

7. **enum(enumeration):**enum is a user-defined type that holds fixed possible values, making it ideal for representing choices or states (like statuses).

Eg:This `enum` defines three states (`Pending`, `Shipped`, `Delivered`) for an order. `orderStatus` is initially set to `Pending`, and this can be updated based on the contract's logic.

```
// Function to demonstrate usage
function updateData() public {
    isActive = !isActive;            // Toggling the boolean
    temperature += 10;               // Modifying signed integer
    totalSupply += 500;              // Modifying unsigned integer
    orderStatus = Status.Shipped;    // Updating Enum value
}
```

This code snippet displays all the data types and their use to implement a **function updateData()** . We will discuss functions in a moment , before that we must understand Operators and Control Flows which are fundamental to build logic and decision making in smart contracts.

Operators:

Operators are special symbols or keywords in Solidity that perform operations on variables and values. They allow you to manipulate data and execute calculations within your smart contracts.

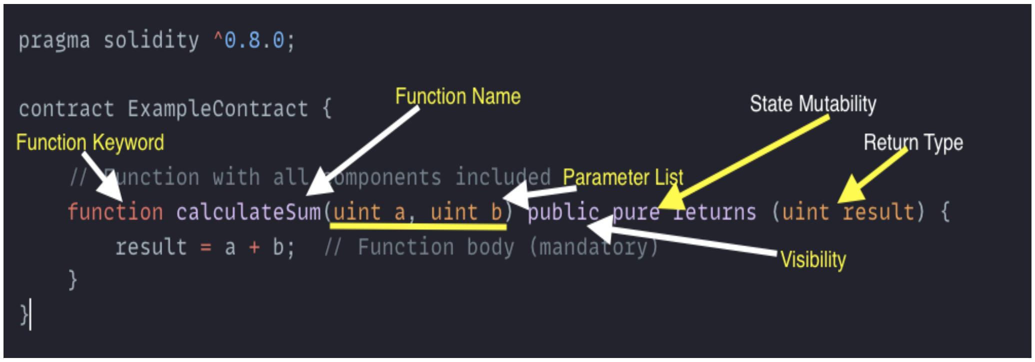
Type of Operator	Description	Example
Arithmetic Operators	Used for mathematical calculations.	`+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), `%` (modulo)
Comparison Operators	Used to compare two values and return a boolean result.	`==` (equal), `!=` (not equal), `>` (greater than), `<` (less than), `>=` (greater than or equal to), `<=` (less than or equal to)
Logical Operators	Used to perform logical operations on boolean values.	`&&` (logical AND), `
Bitwise Operators	Operate at the bit level and perform bitwise operations.	`&` (bitwise AND), `
Assignment Operators	Used to assign values to variables.	`=` (simple assignment), `+=` (add and assign), `-=` (subtract and assign)
Ternary Operator	A shorthand for if-else, allowing conditional assignment.	`condition ? valueIfTrue : valueIfFalse`

Control Flows in Solidity:

Control flows are constructs that determine the order in which instructions are executed in a smart contract. They allow for conditional execution of code, looping through blocks of code, and handling exceptions. The main types of control flows in Solidity include conditional statements (like `if`, `else`, and `switch`), loops (like `for`, `while`, and `do-while`), and exception handling.

Control Flow	Description	Syntax / Example
If Statement	Executes a block of code if the specified condition is true.	`if (condition) { /* code */ }`
Else Statement	Executes a block of code if the previous `if` condition is false.	`if (condition) { /* code */ } else { /* code */ }`
Else If Statement	Specifies a new condition to test if the previous conditions were false.	`if (condition1) { /* code */ } else if (condition2) { /* code */ }`
Switch Statement	Allows a variable to be tested for equality against a list of values (not common in Solidity).	`switch (variable) { case value1: /* code */ break; }`
For Loop	Repeats a block of code a specified number of times.	`for (uint i = 0; i < n; i++) { /* code */ }`
While Loop	Repeats a block of code as long as the specified condition is true.	`while (condition) { /* code */ }`
Do-While Loop	Executes a block of code once and then repeats it as long as the specified condition is true.	`do { /* code */ } while (condition);`
Break Statement	Exits a loop or `switch` statement prematurely.	`break;`
Continue Statement	Skips the current iteration of a loop and proceeds to the next iteration.	`continue;`

Function: It performs repetitive general tasks , thus called multiple times. They are implemented to perform a certain task or action. Like in this case we are updating the values which variables when that function is called while a contract is being executed.



Mandatory Components in a Solidity Function:

1. **“`function`”**: The `function` **keyword** is required.
2. **Function Name:** `calculateSum` – every function must have a unique name.
3. **Parameter List:** `(uint a, uint b)` – includes the function’s inputs, if no inputs are needed to implement a function empty parentheses must come “()”.
4. **Visibility**: `public`, `private`, `internal`, or `external` is now mandatory. Here, we use `public`.
5. **Function Body:** `{ result = a + b; }` – this is required, even if it's just an empty block.

Optional Components:

1. **State Mutability:** `pure` here specifies that the function doesn’t modify or read the blockchain state. Other options include `view` (only reads state) or `payable` (allows the function to receive Ether).
2. **Return Type:** `returns (uint result)` specifies that the function returns a single `uint` value and names the return variable `result`.

Lets deep dive into Visibility and StateMutability , as they are key properties of Function

```
contract VisibilityExample {
    // Public variable: accessible from anywhere
    uint public publicVar = 1;

    // Private variable: accessible only within this contract
    uint private privateVar = 2;

    // Internal variable: accessible within this contract and derived contracts
    uint internal internalVar = 3;

    // Public function: can be called from anywhere
    function getPublicVar() public view returns (uint) {
        return publicVar;
    }

    // Private function: can be called only within this contract
    function getPrivateVar() private view returns (uint) {
        return privateVar;
    }

    // Internal function: can be called within this contract and derived contracts
    function getInternalVar() internal view returns (uint) {
        return internalVar;
    }

    // Function to demonstrate calling the private and internal functions
    function demoVisibility() public view returns (uint) {
        uint sum = getPrivateVar() + getInternalVar();
        return sum;
    }
      // External function, only callable from outside this contract
    function externalFunction() external pure returns (string memory) {
        return "This function is external and can't be called internally.";
    }
}
// Derived contract to demonstrate `internal` accessibility
contract DerivedContract is VisibilityExamples {
    function callInternalFunction() public view returns (uint) {
        // Can call internal functions from the parent contract
        return internalFunction();
    }
}
```

Visibility:

Visibility controls where functions and variables can be accessed from within and outside a contract. Solidity has four visibility types: `public`, `private`, `internal`, and `external`.

Public (`publicVar` & `publicFunction`):

- `publicVar` is a `public` variable, accessible from outside the contract
- `publicFunction` is a `public` function, accessible internally and externally the contract

Private (`privateVar` & `privateFunction`):

- `privateVar` is only accessible within the contract it is declared into.
- `privateFunction` can only be called from within `the contract`.

Internal (`internalVar` & `internalFunction`):

- `internalVar` and `internalFunction` can be accessed within the contract and by derived contracts like `DerivedContract`.

External (`externalFunction`):

- `externalFunction` can only be called externally, not within the contract calling them within the contract will cause error.

State Mutability

State mutability specifies whether or not a function modifies or reads the blockchain's state. There are four mutability keywords: `pure`, `view`, `payable`, and default (no keyword).

Pure

A pure function **does not read or modify** any state variables. It operates only on its input parameters and does not access any contract data. It performs calculations and returns a result based solely on the input values.

```
contract Math {
    function add(uint a, uint b) public pure returns (uint) {
        return a + b; // Pure function, no state modification or reading
    }
}
```

View:

These functions can read state variables but cannot modify them. They can return state variables or perform calculations based on them without changing the state.

```
// View function: can read the state variable
    function getValue() public view returns (uint) {
        return value; // Returns the current state variable
    }
```

Eg: here "value" is the state variable which got returned.

Default Function (Regular Function):

When you declare a function without specifying state mutability (neither `pure` nor `view`), it **can both read and modify** state variables. This is the default behavior. It means that the function can return state variable values and also change them.

```
pragma solidity ^0.8.0;

contract ExampleContract {
    uint public value; // State variable

    // Constructor to initialize the state variable
    constructor(uint _initialValue) {
        value = _initialValue;
    }

    // Default mutability type function: can read and modify state
    function incrementValue(uint _newValue) public {
        value = value + 1; // Modifies the state variable 'value'
    }
}
```

Lets create a simple wallet contract where users can withdraw and deposit funds using the concepts we have learnt till now , This contract serves as a simple wallet that allows users to deposit and withdraw funds while maintaining a balance.

Key Components of the Contract

1. State Variables:
   - `address public owner;`: This variable stores the address of the wallet's owner, making it publicly accessible on the blockchain.
   - `uint public totalBalance;`: This variable tracks the total balance of the wallet and is also publicly accessible.
   - `bool private initialized;`: This flag ensures that the wallet can only be initialized once. It is set to `false` by default.
2. Initialization Function:
   - The `initialize` function acts as a constructor. It is called to set the owner of the wallet and initialize the balance. Constructors will be discussed in next lab on Advanced Topics of Solidity.
   - The function first checks if the wallet has already been initialized by verifying the `initialized` flag. If it has not, it sets the owner's address and initializes the balance to zero, then sets the flag to `true`.
3. Deposit and Withdraw Functions:

- The `deposit` function allows the owner to add funds to the wallet. It checks if the deposit amount is greater than zero before updating the balance.
- The `withdraw` function enables the owner to withdraw funds from the wallet. It checks if the caller is the owner and if sufficient balance exists.

4. View Function:
   - The `getBalance` function returns the current balance of the wallet without modifying the state.
5. Pure Function:
   - The `calculateNewBalance` function is a pure function that takes the current balance and deposit amount as parameters and returns the updated balance.

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


contract SimpleWallet {
    address public owner;        // Address of the wallet owner (state variable)
    uint public totalBalance;    // Stores the wallet's balance (state variable)

    // Flag to check if the contract has been initialized
    bool private initialized = false;

    // Function to initialize the wallet (similar to a constructor)
    function initialize() public {
        if(!initialized, "Already initialized."); // Ensure it can only be called once
        owner = msg.sender;  // Set the contract deployer's address as the owner
        totalBalance = 0;    // Initialize the balance to zero
        initialized = true;  // Set the initialized flag to true
    }

    // --- Functions ---

    // Function to deposit funds into the wallet (public visibility)
    function deposit(uint amount) public {
        if (amount > 0) {  // Basic check to ensure amount is positive
            totalBalance = calculateNewBalance(totalBalance, amount); // Calculate the new balance using a pure function
        }
    }

    // Function to withdraw funds from the wallet (public visibility)
    function withdraw(uint amount) public {
        if (msg.sender == owner && amount <= totalBalance) {  // Checks if caller is owner and balance is sufficient
            totalBalance -= amount;  // Decrease the total balance
        }
    }

    // --- View Function ---

    // View function to get the wallet balance
    function getBalance() public view returns (uint) {
        return totalBalance;  // Returns the wallet balance
    }

    // Pure function to calculate the new balance after a deposit
    function calculateNewBalance(uint currentBalance, uint depositAmount) public pure returns (uint) {
        return currentBalance + depositAmount;  // Returns the new balance
    }
}
```

Exercise:

Create a Solidity smart contract named `SimpleFundraiser` with the following specifications:

1. State Variables:
    - `address public owner`: Stores the contract owner's address.
    - `uint public totalRaised`: Tracks total funds raised through donations.
    - `bool private initialized`: Indicates whether the contract has been initialized.
2. Initialization:
    - Implement an `initialize` function that:
        - Sets the owner to `msg.sender`.
        - Initializes `totalRaised` to zero.
        - Ensures the contract can only be initialized once.
3. Functions:
    - `donate` function:
        - Allows users to contribute funds.
        - Updates `totalRaised` and checks if the donation amount is greater than zero.
    - `withdraw` function:
        - Allows only the owner to withdraw funds.
        - Checks if the caller is the owner and if sufficient funds are available.
    - `getTotalRaised` view function:
        - Returns the total amount raised.
4. Function Visibility:
    - Specify function visibility (public/private) based on access requirements.