Solidity, Data Types, Variables, VAM, Function, Control Structures, ERC Outlay

Prepared By: Abhilash Rajan
Date: 15 October 2025

SOLIDITY

The solidity language is **influenced by C++, JavaScript and Python**. Solidity is also known as a **Contract-Oriented programming language**. Smart contracts are a **piece of code that runs on EVM** and are metered in a very specific way. **Ethereum smart contracts can also be written in Vyper and LLL** (besides just writing EVM bytecode directly). **Solidity is JavaScript-like, Vyper is Python-like while LLL is Lisp-like**.

A **solidity contract** will have to **declare and define the following components** inside its definition.

- **State Variables:** Variables whose **values are permanently stored** in contract storage.
- **Functions:** The **executable units of code** within a contract.
- **Function Modifiers:** Used to **amend the semantics of functions** in a declarative way.
- **Struct Types: User-defined data types** that can group several variables together.
- **Enum Types: Used to create user-defined data types** with a finite set of constant values.
- **Events:** Used to **interact with the EVM logging functionalities**

Basic Data Types

Integer - Stores numbers. There are 2 types of integer data types

- **uint** - To store **unsigned integer values**, **only positive integers along with zero**. uint8 to uint256 are used to define integer from 8 bits to 256 bits. uint256 is the same as uint.
  Syntax used is: **uint variable_name = value**
- **int** - To store **signed integer values**, both **positive and negative integer numbers along with zero**. int256 is the same as int.

String - Written with **double or single quotes** ("foo" or 'bar').
Syntax used is:**string variable_name = value**

Address A special type provided to **store EOA and Contract Account addresses**. The size of an address is **20 bytes**. Syntax used is: **address variable_name = value**

The address comes in two identical flavors:

- **address:** Used **to hold the Ethereum address** of the size 20 bytes.
- **address payable:** Same as the address with **additional member functions transfer and send**.

Basically, we can send ether to address payable but not to address. address payable can be implicitly converted to address but to convert from address to address payable you have to do it explicitly by using payable(address).

Basic Data Types

Enum - Used to create user-defined data types in Solidity. It may help reduce errors due to invalid data assignments in the code.

1. **Define a new data type** with the set of predefined values using the enum keyword.
   The syntax is: **enum name_of_data_type { predefined_values }**
2. **Create a variable** for the new data type;
   The syntax is similar when using other variable data types. **enum name_of_data_type variable_name**

Boolean - Represents any scenario with a binary state True or False. The keyword used is bool. The default value of bool is false. The syntax used is: **bool variable_name = value**

Fixed-size Byte Arrays - Used to hold a sequence of bytes from 1 to 32 using keywords bytes1, bytes2, bytes3, …, bytes32 correspondingly. Another keyword for bytes1 is byte.

The length member function can be used to get the length of a particular variable. It is **a read-only property**. The following syntax is used: **bytes32 variable_name = value**

Implicit conversion from a string value to byte is possible. So it is possible to assign a string value to a byte variable in the program.

Solidity Variables

Variables are reserved memory locations to store data. When a variable is declared, some space is reserved in memory for storing the data. The syntax for creating a variable is as follows:
data_type variable_name = value;
Eg: **uint256 number = 123**;

There are 3 types of variables in Solidity

- **State Variables** − Values are **permanently stored in contract** storage.
- **Local Variables** − Values are **available only within a function where it is defined**. Function parameters are always local to that function.
- **Global Variables** − Special variables that exist in the global namespace are **used to get information about the blockchain**.

  Some examples are:
  msg.sender: returns an address, that of the sender of the message
  timestamp: current block timestamp UNIX timestamp format.
  getTime: return the block timestamp

State Variables & Local Variables

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;
contract Storage {
    uint256 result; // State variable
    // function parameter(s) are  also local variable
    function add(uint256 number1) public {
        uint number2 = 1234;  // local variable
        result = number1 + number2;
    }
    function retrieve() public view returns (uint256){
        return result;
    }
}
```

Global Variables

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;
contract Storage {
    function msgSender() public view
returns(address){
        return msg.sender;
    }
    function getTime() public view returns(uint){
        return block.timestamp;
    }
}
```

Access to Contract & Visibility/Access Modifiers

4 entities are able to access a smart contract that is deployed on the Ethereum network.

- **User** - **Controls an EOA** (Externally Owned Account) in Ethereum.
- **Self** - The **contract itself** has access to all of its properties.
- **Derived Contract** - The **child contract** of that particular contract has access. It inherits another contract
- **External Contract** - Can **access the contract properties**.

The contract that gets inherited is known as the **base or parent contract**. We can only use a state variable name in the derived contract if it does not shadow any declaration in the base contract; otherwise, it will cause an error. The derived contract will have access to all public and internal state variables and all public, internal and external functions in the base contract to be used as its own. Scope of local variables is limited to the function in which they are defined, but for state variables it's different. Each state variable's scope visibility or access modifiers can be set; which decides who has access to these variables. The access modifiers available to state variables are:

- **public**: The variable declared public will be **accessible to everyone**.
- **private**: The variable declared private will **only be available to the contract**.
- **internal**: A variable declared as internal will be **accessible by the contract itself and by the child contract** (instantiate a contract from within another contract).

The default visibility of a state variable is internal.

Syntax to declare a variable with visibility/access modifier

Getter functions have external visibility. Compile and deploy the contract to any Remix VM using the Remix IDE.

datatype visibility_modifier name_of_variable = value

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

pragma solidity ^0.8.20;

contract ExampleStateVariable {

    uint public var1 = 123;
    uint private var2 = 456;
    uint internal var3 = 789;

}
```

Function

A block of code that **accepts an arbitrary length of solidity types as input and provides an arbitrary length of solidity types as result**. Function parameters can be **declared in the same way as a state variable** declaration. Function name and parameters are followed by the visibility modifier like public, private etc.. The function and parameter syntax are as follows:
function name of function (parameter_type parameter_label) visibility_modifier {}

- Use special symbols like underscore or dash
- Don't start the function name with a number

Eg: contract Storage { uint256 number;
// function to store value to number variable
function store(uint256 num) public { number = num; }
// calling store function from another function
function callStore() public { store(1001); }
// calling store function from another function by specifying input parameter name
function callStoreAnother() public { store({num: 1001}); }
// function to retrieve the data in number variable
function retrieve() public view returns (uint256){ return number; }}

Return Values

Syntax for the function with a return value is:
function name() visibility_modifier mutability returns(parameter_type parameter_label) {}

Eg: **function retrieve() public view returns (uint256){ return number; }**

Return values can be specified by specifying each return value data type and then using the return keyword to return the values. Another way is to label return values in the function header and then assign the result to that label.

```
// function to retrieve the data in number variable
// using output parameter label instead of return keyword
    function retrieveAnother() public view returns (uint256 _num){
        _num = number;    }
```

Function Mutability Modifier

The extent of interaction of a function with the blockchain state. The function mutability is categorized into 4:

- **default**: The function **can change the state of the blockchain and can read from the state of the blockchain**. If no mutability is specified, a function is considered to be of default mutability.
- **view**: The function **can only view and can't modify the state**. The following actions can be considered as **modifying the state**.
  Writing to a state variable, Emitting events, Creating contracts, Using selfdestruct (with the exception of msg.sig and msg.data), Sending Ether via calls, Calling any function not marked view or pure, Using low-level calls, Using inline assembly that contains certain opcodes.
- **pure**: The function **can't read or modify the state**. Apart from the above list of state modifications, the pure functions **can't perform the below actions**:
  Reading from state variables, Accessing address(this).balance or <address>.balance., Accessing any of the members of the block, tx, msg (with the exception of msg.sig and msg.data), Calling any function not marked pure, pure and view functions do not make any changes to the blockchain, thus they do not have any cost for execution. So we do not need to worry about ether balance for calling pure/view functions.
- **payable**: Function is able **to receive ether on behalf of the contract**.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

contract FunctionMutability {
    uint number;
    function get() view public returns(uint) {
        return number;
    }
    function set(uint _number) public {
        number = _number;
    }
    function calculate(uint a, uint b) pure public returns(uint) {
        return a + b;
    }
    function sendEther() public payable returns(uint) {
        return address(this).balance;
    }
}
```

Control Structures

Solidity provides the features of control structures, which helps us **to control the flow of smart contract execution**. The list of available control structures in Solidity are.

- **If… else… if…** Statement - Used to check for a condition and depending on the result of the condition, being true, we can execute a block of codes. The syntax is:
  if (condition) { // execute the code } else { // execute the code }
- **while** Loop - Will repeatedly execute a set of code as long as the condition is true. Also known as the **entry control loop** as the condition is evaluated before processing a body of the loop. The syntax is: **while (condition) { // execute the code }**
- **do... while** Loop - Executes a set of codes until the condition turns out to be false.The condition check happens after executing the loop body, and due to this fact do...while loop is known as an **exit control loop**. The syntax is: **do { // execute code } while (condition);**
- **for** Loop - Used to iteratively execute a code block as long as a condition holds. Also known as the **entry control loop**. The syntax is:
  for (initialize counter; condition check; update counter) { // code block to be executed}
- **Loop** Controllers - We can control the loops' execution flow up to some limit **using continue and break**.
  continue - used to **skip the current iteration** in the loop,
  for(uint i = 1; i <= 5; i++) {if (i == 1) { continue; } sum = sum + i; }
  break - will **exit the loop**.
  for(uint i = 1; i <= 5; i++) { if (i == 5) { break; } sum = sum + i; }

Arrays

Used to **group together variables of the same type**, in which each individual variable has an index location using which it can be retrieved. The array size can be fixed or dynamic.

To create a fixed-sized array, the following syntax is used.
Type[size] array name —-> **unit[8] numbers;**

To create a dynamic array, just leave the size field blank.
Type[] array name —----> **unit[] numbers;**

Member Functions

- **length**: can be used to get **the size of both fixed and dynamic array**.
- **push**: is used to **insert elements to a dynamic array**.
- **pop**: is used to **remove the last element of a dynamic array**. This also implicitly calls delete on that element.

Structs - Allows to group the different or the same type of variables together to form user-defined data types.

```
struct Book {
    uint id;
    string title;
    string author;   }
```

Nested Struct

We can nest the structs together.

```
// Define a struct to hold KYC information
    struct kyc {
        uint256 iD;
        string firstName;
        string lastName;
        AddressDetails location;
    }

// Define a nested struct to hold address details
    struct AddressDetails {
        string buildingName;
        string street;
        string state;
        uint256 pinCode;
    }
```

Working with ERC 20 Tokens

use **OpenZeppelin** — a trusted library of secure smart contract templates.
The ERC-20 standard contains 6 key functions that must be implemented to meet the standard.

Contracts/ New File/ MyERC20Token.sol

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

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract MyERC20Token is ERC20 {
    constructor(uint256 initialSupply) ERC20("MyToken", "MTK") {
        _mint(msg.sender, initialSupply);
    }}
```

Solidity Compiler Tab - Compile MyERC20Token.sol
Deploy & Run Transactions tab - MyERC20Token - initial supply (e.g., 10000000000000000000 for 1000 tokens in wei) - Deploy

ERC-20 token is deployed

Working with ERC-721 Token (NFT)

Contracts/ New File/ MyNFT.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
import "@openzeppelin/contracts/token/ERC721/ERC721.sol";

contract MyNFT is ERC721 {
    uint256 public tokenCounter;
    constructor() ERC721("MyNFT", "MNFT") {
        tokenCounter = 0;    }
    function createNFT(address recipient) public returns (uint256) {
        uint256 newTokenId = tokenCounter;
        _safeMint(recipient, newTokenId);
        tokenCounter++;
        return newTokenId;    } }
```

Compile - Deploy

Call createNFT(address) from the deployed contract

Working with ERC-1155 Token

Contracts/ New File/ MyERC1155.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
import "@openzeppelin/contracts/token/ERC1155/ERC1155.sol";

contract MyERC1155 is ERC1155 {
    constructor() ERC1155("https://api.example.com/metadata/{id}.json") {}
    function mint(address to, uint256 id, uint256 amount, bytes memory data) public {
        _mint(to, id, amount, data);    } }
```

Compile - Deploy

Call mint with token ID, amount, and empty data (0x)