SOLIDITY

Syntax

// SPDX-License-Identifier: MIT

Pragma Solidity ^0.8.7;

Contract C-name { (^ means = to or ① er than)

// Code

}

Data Types

- uint → unsigned Integer

means a no. is ① er than or = to zero.

uint public u = 123;

// uint = uint 256 0 to 2**256 - 1

uint 8 0 to 2**8 - 1

uint 16 0 to 2**16 - 1

- For -ve no.s (int public i = -123;)

int = int 256 -2** 255 to 2**255-1

int 128

- int public minInt = type(int).min;

" " maxInt = " ".max;

- address public addr = some_address;

- Bytes32 →

Use this data type when u're working with the Cryptographic hash funch. available in Solidity called Catch act 256.

bytes_32 public b32 = some_random_32 bytes.

Syntax for Funch. →

contract C {

function F.name (values u want to pass) external pure returns (uint) {

return x+y; // code

}

}

Data type u're going to return.

- Functions are not allowed to have the same name as contract

external: when we deploy this contract, u'll be able to call this funch.

pure: That this funch. is read only it doesn't write anything to the Blockchain.

Variables

Local State Global

State → variables that store data on Bc. Declared inside a contract but outside of a funch.

Local → variables declared inside a funch. are local var.

- They only exist while the funch. is executing.

Global → Store info. such as Bc Txn. & the account that call the funch.

var.	data type
a) ~~mess~~ msg.sender	(address)
b) block.timestamp	(uint)
c) block.number	(uint)

a) Stores the address that calls this Funch.

b) stores the unix timestamp of when this Funch. was called

c) stores the current Block no.

Pure Vs View

- View Funch. can read data from the Bc whereas Pure Funch. doesnot.
- f view' → read only Funch. view ensures that state variables can not be modified after calling them.
  Otherwise it throws warning.
- Pure → returns the values only using the parameters passed to the Funch or local var. present in it.

Public → we'll have read access after the contract is deployed.

Neither pure Nor View →

```
uint public count;
Function inc() external {
  count += 1;
}
```

// This Funch. is neither view nor Pure coz we're going to be modifying the count state variable.

- Default value for
  bool is False
  uint is 0
  int is 0
  address is 0x00000----0
  bytes32 is 0x000000---upto 32 bytes---

Const → By declaring a state variable as const., u'll be able to save gas when a Funch. is called that uses that state variable.
(uint public constant num = 86;)

Ternary operator →

```
if (x < 10) {
    return 1;
}
return 2;
```

} If-else

// same code in 1 line

```
{return x<10 ? 1 : 2;
```

(x<10 ? → If it is; 1 → otherwise)

Eg. of Ternary operator.

* The bigger the no. of the loops, the more gas it will use.

Error → 3 ways to throw an Error: require, revert, assert

- when an error is thrown inside a Txn. ur gas will be refunded & any state variables that were updated will be Reverted.
  Changes made will be undone.

- In solidity 0.8, use custom error to save gas.

Require → is mostly used to validate inputs & for access control, controlling who gets to ~~contract~~ Call the Funct.

Revert → Require & Revert does the same thing but Revert is a better option, if your condition is to check in nested a lot of if statements.

Assert → used to check for condition that should always be True. If condition is false, then there might be a bug in your Smart Contract.

Function Modifier → Allow u to reuse code.
// do it later.

Constructors → Special Funct's that are only called once when the Contract is deployed mainly used to initialize state variables.

- when u delete an element of array in Solidity, the size of array will remain same.
→ delete is just resets the element to its default value. viz 0.

Array Shift → For deleting the particular element, we shift its Right ones & delete the last one. (pop)

```
arr = [1,2,3]
delete arr[1]; // [1,0,3]
```

// [1,2,3] -- remove(1) -->
[1,3,3] --> [1,3]

// [1,2,3,4,5,6] → remove(2) →
[1,2,4,4,5,6,6] → [1,2,4,5,6]

// [1] → remove(0) → [1] → []

- arr.pop() → removes the last element by default.

Mapping → It's like a dictionary in Python. It allows for efficient lookup.
// do it later.

- uint public x = 10 // state var.
funcn. ——— view —— {
return x;
}
view is there coz it reads a value outside its scope.

- uint pub. x = 10
funcn. ——— pure —— {
return 1;
}
pure is there —— it doesn't read any value.

- But when there is a cotn. variable funcn. will be pure.

Funcⁿ Modifiers → These are used to change or restrict the behaviour of a funⁿ. in S.Con.

- You can use a modifier to automatically check a condition prior to executing a funcⁿ.
- For Eg → U can check the balance of an Account, verify the sender is the owner, require access to an account.

Syntax →

define a Funcⁿ modifier

```
modifier name() {
    require ( yr requirement, "Err Msg");
        If it fails raise an Err. Msg.
    _;
        Instruction to execute the rest of the Funcn.
}
```

Array →

```
uint[] ~~num~~ public nums = [1, 2, 3];
uint[3] public nums = [4,5,6];
// U can push, get, updat, delete, pop, length in Array
// Creating array in memory
uint[] memory a = new uint[](5);
a[1] = 123.
```

Mapping → Acts like Hash table or dictionary.

- Used to store data in the key-value pairs,
- Key can be any of the built in-data types
- But reference types are not allowed while the value can be of any type.
- Mostly used to associate the Unique Ethereum address with the associated value type.

Syntax

```
mapping (key => value) public name;
```

- We cannot get the size of an Mapping & cannot iterate it just like we can do it in Arrays
- Unless we internally keep track of all the keys in the Mapping.

Access Modifiers

- Keywords used to specify the declared accessibility of a Funcⁿ. or a type.

4 Types → Public, Private, Internal, External

Public → Can be Inherited & can be accessed by external elements. All can access a public element.

Private → doesn't get Inherited & can't be accessed by external elements.

- Can be accessed from the Current contract Instance only.

Internal → can be Inherited but can't be accessed by external elements. Only the base contract & derived contract can access it.

External → Can't be inherited but can be accessed by external elements.
Current contract instance can't access it.
Can be accessed by externally only.

Struct → It allows u to group data together.

Code →

```
struct Car {
        string model;
        uint year;
        address owner;
}
Car public car;
Car[] public cars;
function initialize() external {        // 3 ways
① Car memory toyota = Car("Toyota", 1980, msg.sender);   // order should be taken seriously
② Car memory Lambo = Car({ year: 1980, model: "Lambo", owner: msg.sender});
        // order can be anything.
③ Car memory tesla;
   tesla.model = "Tesla";
   tesla.year = 2010;
   tesla.owner = msg.sender;

   cars.push(toyota);
   cars.push(Lambo);
   cars.push(tesla);

   Another way to store directly
   cars.push(Car("Ferrari", 2020, msg.sender));
```

declare

storing it to state variables so that we can use them later on.

memory Vs Storage Vs Calldata

- If we load anything on memory, then when the Funch. is done executing nothing is saved whereas storage would mean that we want to update the variable stored inside the Smart Contract.
- Once Funch. is done executing, change will be saved.
- Calldata is just like memory, except it can only be used for Funch. inputs

Enum →

Structs allow u to express multiple choices, for eg. bool allows to choose T or F but if we need to express more choices then enum is a great choice.

Immutable Vs Constn.

→ Immutable makes the contract that this object will not change.

→ Cotn. makes the Contract that in the scope of the variable, it will not be modified.

Payable

It adds functionality to send & receive ether.

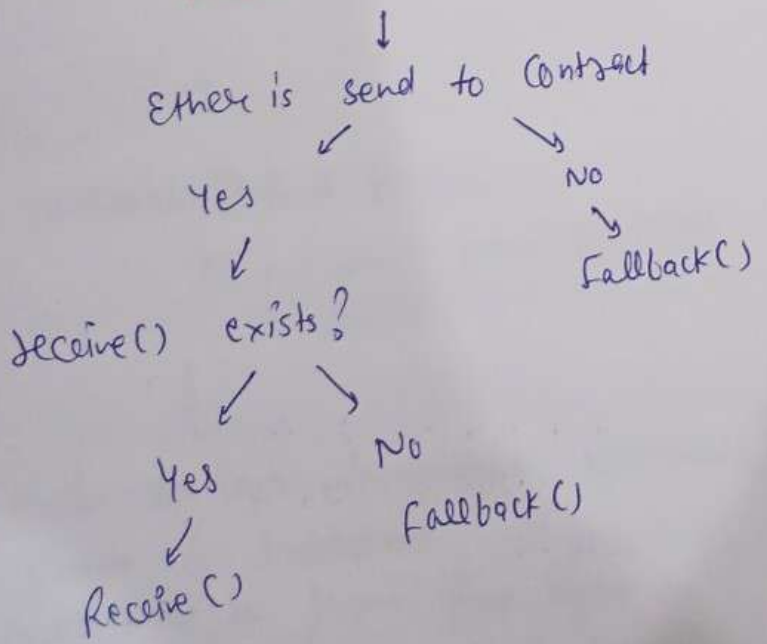
Fallback

It's a spedal Func. that gets called when a func. that u call doesn't exist inside the Contract.

Main UseCase → directly send Eth.

meaning when another Contract tries to directly send eth to the Fallback Contract, the contract in which Fallback is there will be executed.

Fallback() or Received()?



Send Eth → 3 ways to send Eth.

// transfer → 2300 gas, reverts

// Send → 2300 gas, returns bool

// Call → all gas, returns bool & data.

* As of now, recommended way to transfer Eth is to use Call.

Interface →

→ To call Functns. in another Contract.

→ most useful in scenarios where yr application requires extensibility but u donot want to introduce added complexity.

They reduce code duplication & overhead.

Characteristics →

- Interface keyword u've to write
- a name starts with "I" to easily identify them in code.
- All interface Funcn. are Implicitly Virtual.
- U can override an Intefe. Func.
- Contract can inherit interfaces as they would inherit other contracts.

Restrictions → (Interface)

- They cannot inherit from other contracts, but they can inherit from other Interfaces.
- Funcs. of interface can be only of type external.
- They neither declare constructor nor state Variables.

Library → Libraries allow u to separate & reuse code & to enhance data types.

Syntax →

```
library <libraryName> {
    // block of code
}
```

* U cannot declare a state var inside ~~of~~ Library.
* coz this code is not inside any Contract, we can also use this Lib. in another Contract.

This is how libraries separate code logic so that we can reuse it in another Contracts.

Imposting a Lib →

import <lib.Name> from "./lib-File.sol";

- Single file can contain x/e libraries that can be specified using curly braces in the import statement separated by Commas.
- A lib. can be accessed within S.C. by using 'for' keyword.

Syntax

<LibName> for <dataType>;

Inbuilt Libraries

① Modular ② OpenZeppelin ③ Dapp-bin

① very useful for implementatn. like ArrayUtils, Token, CrowdSale, Vesting, StringUtils, LinkedList, Wallet, etc.

② other supporting libraries are Roles, Merkleproof, ECDSA, Math, Address, SafeERC20, ERC165Checker, SafeMath, Arrays etc which protects from overflow.

③ Created by Ethereum includes interesting & useful libraries like DoublyLinkedList, StrigUtils, IterableMapping etc.

* Crypto Graphic Hash Func. is an Algo. that accepts any amt. of data & outputs a fixed size encrypted text.
- The Tiniest alteration in the data can cause an entire shift in the output.

Keccak256 ?

Used to process the keccak-256 hash of the data input & can be used for →

* To create a deterministic, one of a kind ID from a set of Data.
* Commit Reveal Scheme.
* Cryptographic Signature with a small size (by signing the hash instead of a larger input).

Encode Vs Encode Packed

Encode just encodes data into bytes whereas encodePacked compresses these data.

Self-Destruct

- when u call it, u can delete the Contract from the B.C.
- Also, when u call it, other than deleting, u'll also be able to send ether to any address even if that address is a contract then it doesn't have a fallback Func^{n}.
- Simply Selfdestruct() sends all remaining Ether stored in the Contract to a designated address which must be type payable.

Func^{n}. Selector →

- The first 4 bytes of Calldata specify which Func^{n}. has to be called - This is called F.S.
- The Func below uses call to execute the transfer action.

addr.call(abi.encodeWithSignature("transfer(address, uint256)", 0xSomeAddrs, 123))

- To be more precise, the first 4 bytes returned from abi.encodeWithSignature(....) is the F.S.

Key-value Inputs

- when u call Func. in solidity, u'll have to pass the inputs in the order that they are declared in.
- This can be Problematic when u've a lot of inputs to pass in, u'll have to remember the order of the inputs.

→ That's where Key-value pair inputs comes in.

Event

- Event is an inheritable member of a contract. An event is emitted, it stores the arguments passed in txn. logs.
- These logs are stored on BC & are accessible using address of the Contract till the Contract is present on the BC.
- An event generated is not accessible from within Contracts, not even the one which have created & emitted them.

```
event <name>(parameters);
```

- we can also add an Index to our event. On adding the diff. fields to our event, we can help add an index to them it helps them later but of course, it's going to cost some more gas.

* we can add atmost 3 indexes in 1 event.

```
// declare event
event deposit(address indexed _from,
    bytes32 indexed _id, uint value);
```

Transfer / Send / Call

In solidity, these are 3 ways in which one can send Ether.

Transfer → The receiving smart contract should have a fallback func. defined or else the transfer ~~func~~ call. will throw an error.

- There is a gas limit of 2300 gas, viz enough to complete the transfer operation.
- It is hardcoded to prevent reentrancy attacks.

Send → Same as Transfer call but it returns the status as a Bool.

Call →
- Recommend way of sending ETH to a Smart Contract. The Empty argument triggers the fall back Func.n of the receiving address.

```
(bool sent, memory data) =
    _to.call{value: msg.value}("");
```

- Using Call, one can also trigger other Funcns. defined in the Contract & send a fixed Amt. of gas to execute the funch.

```
(bool sent, bytes memory data) = _to.call
{gas: 10000, value: msg.value}
("func_signature(uint256 args)");
```

Delegate → Assignment of Authority of work to Another Person.

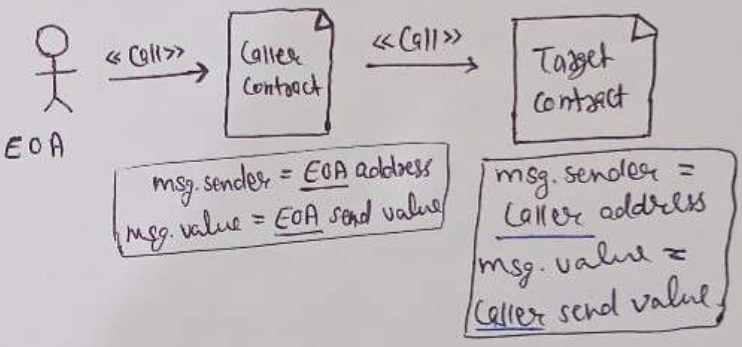
DelegateCall →

We can call another contract's func. from the contract.

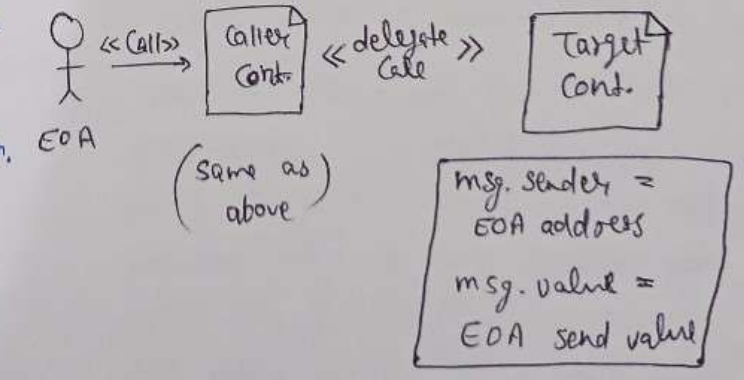
→ If we know the ABI of the target Func., we can directly use the target Func. signature.

→ But if we don't know the ABI, we can use call() or delegatecall().

In case of DelegateCall, we need to care about the order of the Field variable.



Context when the Contract Calls another Cont.



Context when the Contract delegateCalls Another Cont.

Verify Signature →

The process of verifying a signature using Solidity is in 4 steps →

1) Message to sign.
2) Hash the message.
3) u would sign the message, this will be done off chain. This will not be done inside the S.C, so should be done using a wallet.
   sign (hash (message), private key).
4) Calling the Func. ecrecover, passing in the Hash of the message & the Signature
   ecrecover (hash (message), Signature) == signer

CREATE 2

- This opcode gives us the ability to predict the address where a contract will be deployed, without ever having to do so.
- This opens up a lot of possibilities to improve user onboarding & scalability.
- The whole Idea behind this is to make the resulting address independent of future events.
  Regardless of what may Happen on the Blockchain, it'll allways be possible to deploy the contract at the precomputed address.

new_address = hash (0xFF, sender, salt, byte code)

→ 0xFF, a const^n. that prevents collision with CREATE

→ Salt: an arbitrary value provided by the Sender.

Encode data →

There are 3 ways to encode data to be passed to the low-level Func. Call.

① encode With Signature
② encode With Selector
③ encode Call

① In this, u can mistype the Func. that u're going to be Calling & the Contract still Compiles.

② u can put in the wrong types ~~of inputs~~ & the wrong amount of Inputs but we won't be able to make a mistake with a typo for the Function name.

③ Both Func. & Inputs must be correct.

* 0x20 is = to 32 bytes in ~~Hex~~ Hexadecimal.
* See video # 26 by Solidity with Eg.

◦ Address (this)

→ Used in Solidity version >= 5.0.0

Code

address pub myadr = address(this);
uint Pub balance = address(this).balance;

Output →

0: address: 0xDA0bab ——— B53

Balance : 0

→ This keyword gives us the Contract Address.

Event → Later # video 30

Inheritance → Put virtual after view/ Pure. This keyword will tell solidity that this Func. can be inherited & customized by the other contract

Syntax

```
Contract A {
    Func ——— pure virtual :- {
    }

contract B is A {
    Func. —— pure override —{
    ——
    }
```

◦ In multiple Inheritance Contracts, order is important. like Order must be from Base ~~right~~ to derived.

◦ Some Eg of orders

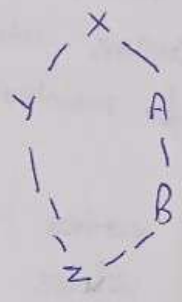
① 

∵ Z is derived from both Y & X. It is most derived.

Y is derived from X

X is not derived from someone so it is the most baselike.

∴ order → X, Y, Z

② 

order → X, Y, A, B, Z

Syntax of xle Inheritance of ~~ge~~

```
Contract X {
    Func. — pure virtual — {
    ——
    }
}
contract Y is X {
    Func. — pure virtual override —{
    ——
    }
}
Contract Z is X, Y {
    Func. — pure override (X, Y) —{
    ——
    }
}
```

◦ ll video #33, 34 emit?