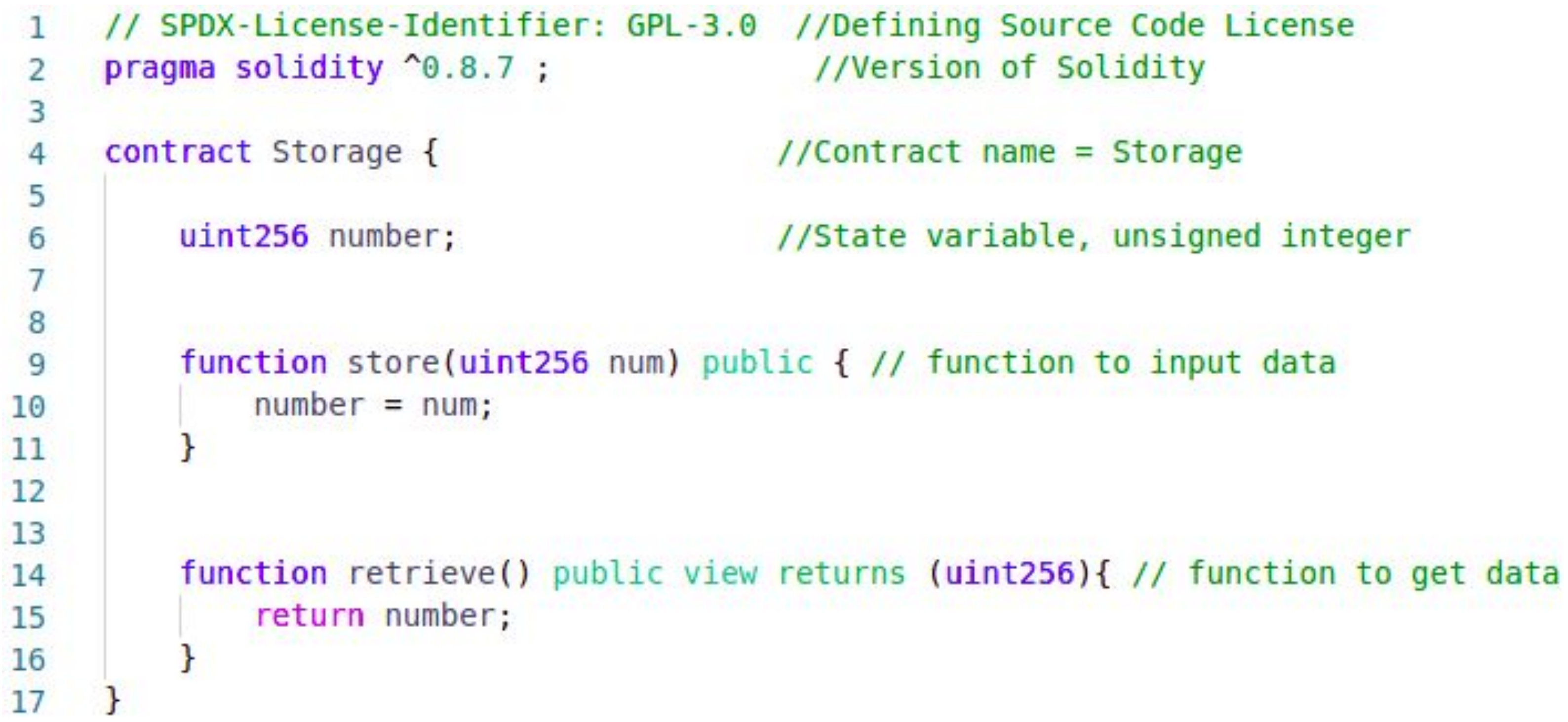
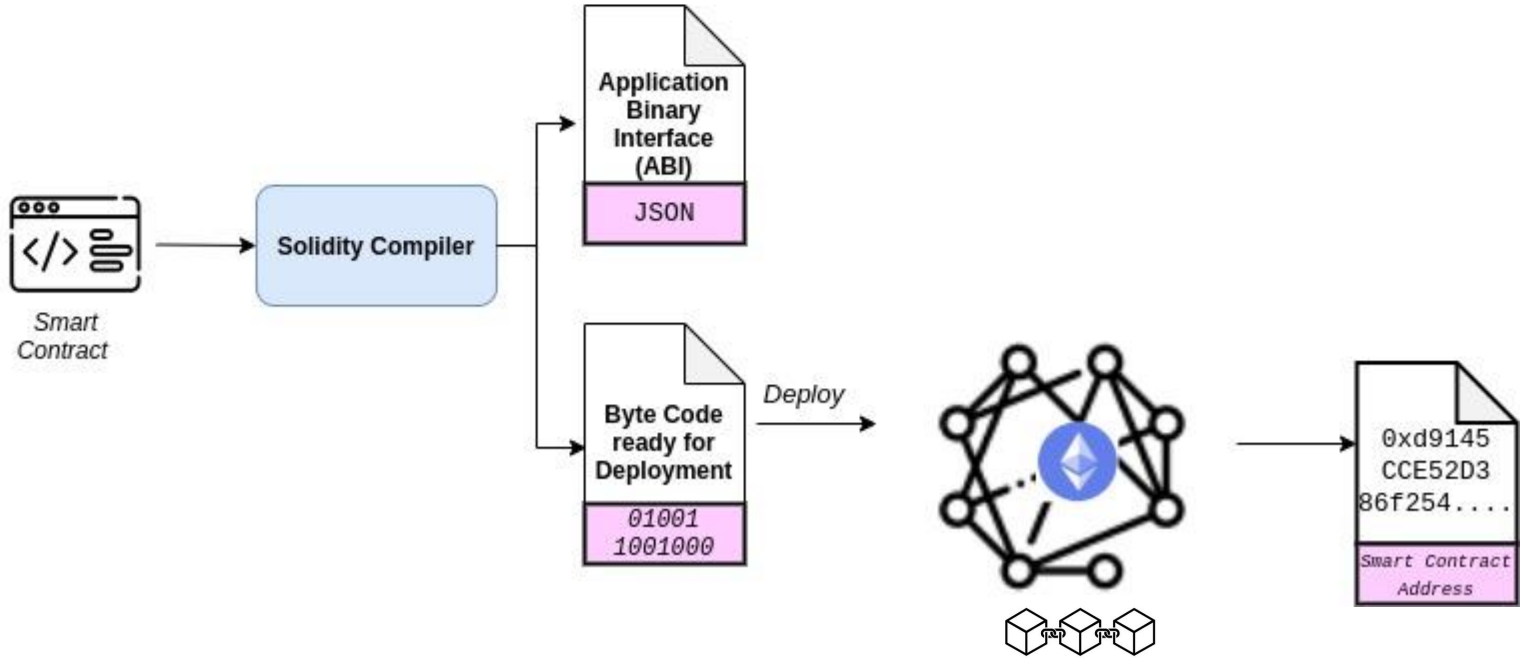
SOLIDITY

Smart Contract Example

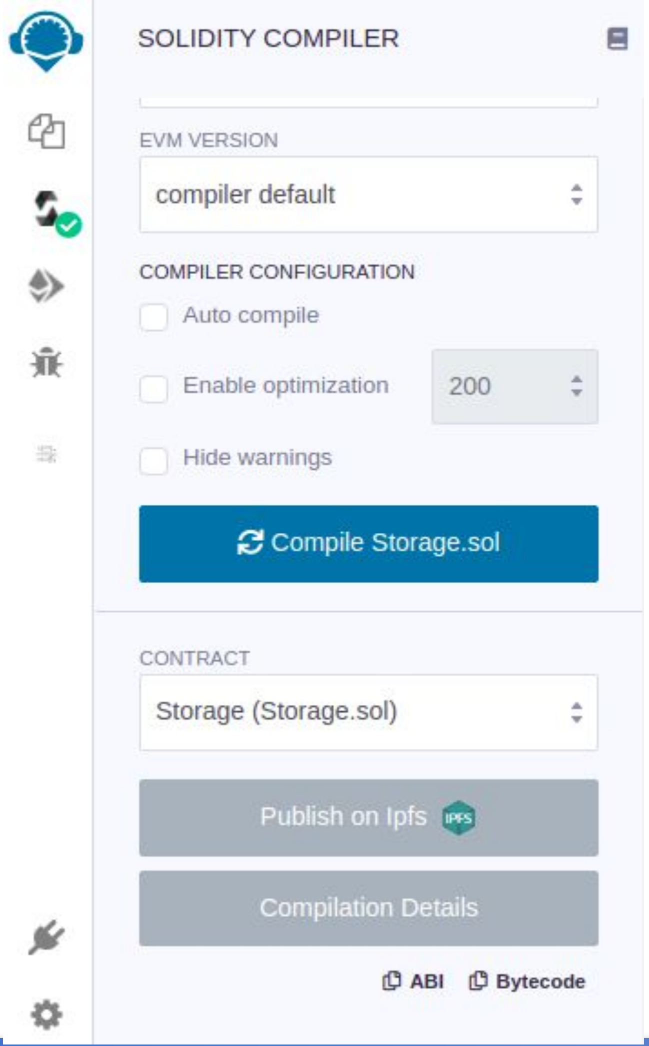


```
// SPDX-License-Identifier: GPL-3.0  //Defining Source Code License
pragma solidity ^0.8.7 ;               //Version of Solidity

contract Storage {                    //Contract name = Storage

    uint256 number;                   //State variable, unsigned integer


    function store(uint256 num) public { // function to input data
        number = num;
    }


    function retrieve() public view returns (uint256){ // function to get data
        return number;
    }
}
```

Recap: Compilation & Deployment of Smart Contract


Smart Contract
Solidity Compiler
Application Binary Interface (ABI)
JSON
Byte Code ready for Deployment
01001 1001000
Deploy
0xd9145 CCE52D3 86f254....
Smart Contract Address

Remix IDE: Compile Smart Contract

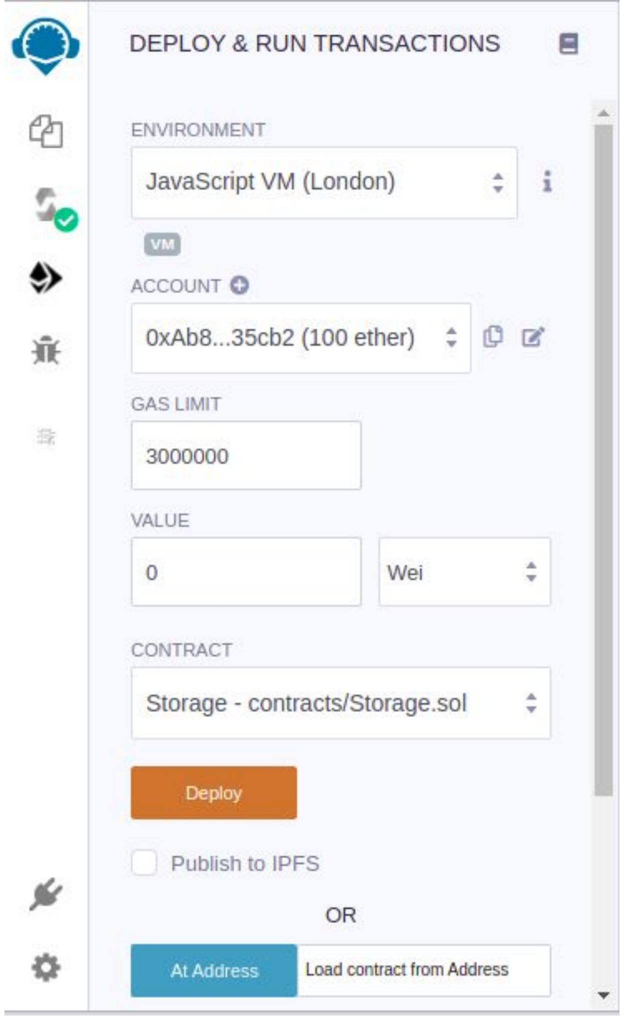

SOLIDITY COMPILER

EVM VERSION
compiler default

COMPILER CONFIGURATION
- [ ] Auto compile
- [ ] Enable optimization 200
- [ ] Hide warnings

CONTRACT
Storage (Storage.sol)

Remix IDE: Deploy Smart Contract



ENVIRONMENT: JavaScript VM (London)

ACCOUNT: 0xAb8...35cb2 (100 ether)

GAS LIMIT: 3000000

VALUE: 0 Wei

CONTRACT: Storage - contracts/Storage.sol

Publish to IPFS

Load contract from Address

Remix IDE: Deploy & Run Options

Environment: The environment to which the contract is deployed, can be one of the below.

Remix VM: A simulation of the Ethereum node by Remix developers for testing purposes inside the Remix IDE.

Injected Web3: Establish a connection to Metamask or similar wallet applications, which provides a connection object.

Web3 Provider: Connect to an Ethereum node (eg: Geth) running on a computer. This option can also be used to connect to Ethereum node simulation tools.

L2 provider:

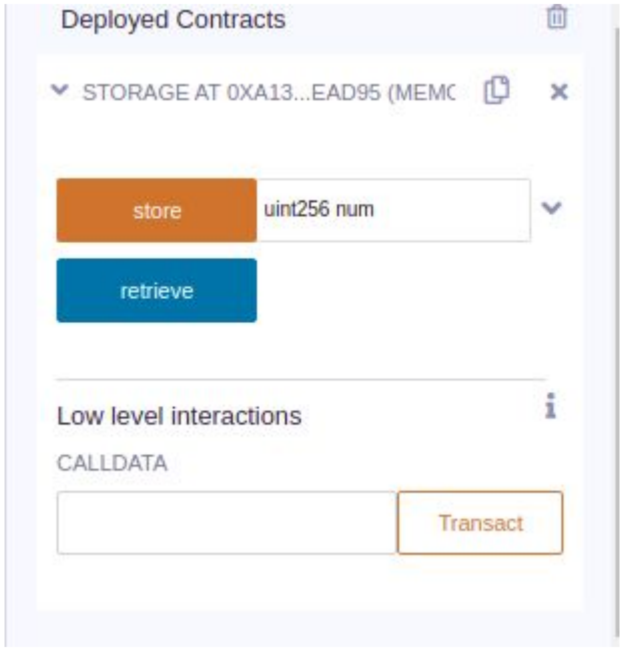
Account: Shows the list of unlocked accounts in the connected environment.

Gas Limit: The maximum gas that can be used for a transaction. .

Value: When you are required to transfer ether to contract, the ether value can be given here.

Remix IDE: Deployed Smart Contract

Blue: *constant* or *pure* function,
not a transaction (no fee), no state change
Orange: State changes, new transaction
Red: payable functions, new transaction , accept *value*



Solidity Data Types

Integer

Address

Boolean

String

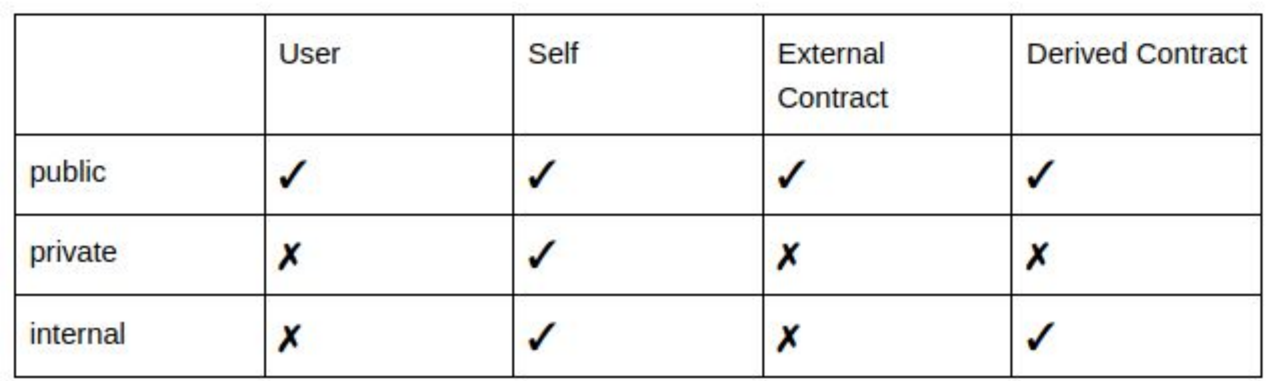
Solidity: Data Types

- Integers: int (signed) uint(unsigned)
  - uint8 to uint 256 or int8 to int256 in steps of 8, default is int256 / uint256
  - Uint is alias for uint256
  - Default value is 0.
  - Uint8 : range is 0 - (2^8 -1), ie (0-255)   int8(-128 to +127)
  - type( ): inbuilt function which returns max and min value that can be rep by int/uint

Solidity: Data Types

- Boolean: bool.
  - Value can be true or false, default: false.
  - Explicit type conversion is not allowed from int to bool in solidity.
  - Support logical operators: !   && || ==   !=

Solidity State variables: Access Specifier



	User	Self	External Contract	Derived Contract
public	✓	✓	✓	✓
private	✗	✓	✗	✗
internal	✗	✓	✗	✓

The compiler will automatically create a getter function with external visibility, for *public* variables, using the name of the variable.

Default access specifier: *internal*

Solidity : EVM Data Store

Storage: stores the state variables of a contract.

Each contract has independent storage

Its persistent and expensive to use.

Memory: stores temporary values. [used for dynamic types like *string* and *byte*]

Values are cleaned between(external) function calls.

Function arguments are stored in memory by default.

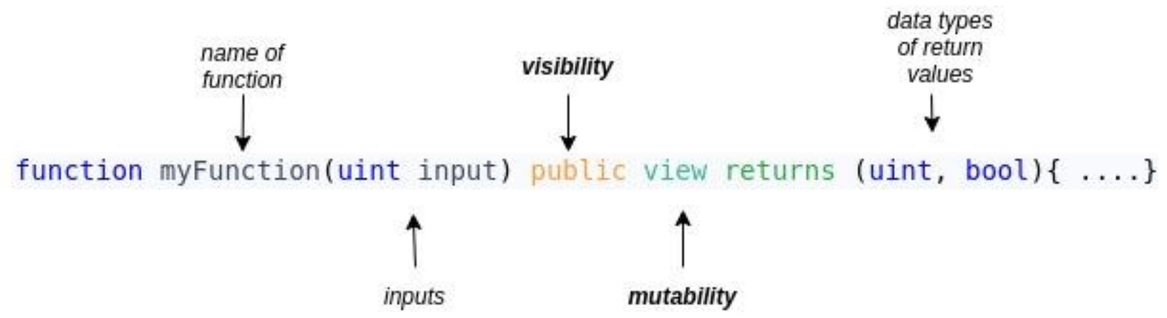
Stack: Used to hold small local variables (ones which are declared inside a function).

It is almost free to use but can only hold a limited number of values.

Memory Vs Storage Vs Stack

Storage	Memory	Stack
permanent	temporary	temporary
(key-value) pairs	byte array	byte array
contract state	local variables in function calls	temporary values of value types
very expensive	no gas cost	Has gas cost but less expensive

Solidity : Function

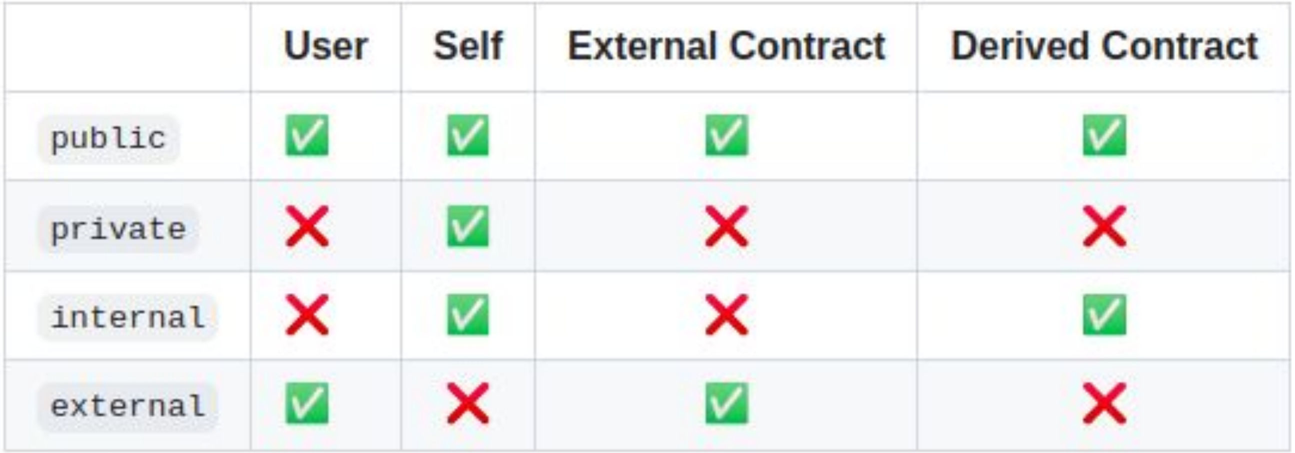


```
function myFunction(uint input) public view returns (uint, bool){ ....}
```

Function Visibility: `public, private, internal, external`

Function Mutability: `default, pure, view, payable`

Function Visibility



	User	Self	External Contract	Derived Contract
`public`	✅	✅	✅	✅
`private`	❌	✅	❌	❌
`internal`	❌	✅	❌	✅
`external`	✅	❌	✅	❌

❖ Visibility types for functions defined in contracts have to be specified explicitly, they do not have a default.

Function Mutability

- defines the extent of interaction of a function with the blockchain state
- Read operations are performed by referring to a local node (**call**); no need to inform peers in the network
- Write operations should be informed to all peers (**transactions**)

Function Mutability

- Blockchain state is altered by below operations
  - Writing to a state variable
  - Emitting events
  - Creating contracts
  - Using selfdestruct
  - Sending Ether via calls
  - Calling any function not marked view or pure.
  - Using low-level calls.
  - Using inline assembly that contains certain opcodes.

K B A

Function Mutability

- *Default*: function can change the state of the blockchain and the function can read from the state of the blockchain.
- *View*: The function can only view and cannot modify the state.
- *Pure*: Almost the same as view, but pure function cannot read or modify the state. It can call only pure functions.
- *Payable*: Function is able to receive ether on behalf of the contract.

K B A

Solidity: Control Structure

- ❖ If-else
- ❖ If-else if—
- ❖ While loop
- ❖ Do-while loop
- ❖ For loop
- ❖ Continue & break
- ❖ Ternary operator (? :)

Solidity: Fixed point data type

- Fixed point (floating point)
  -

ufixedMxN and fixedMxN are reserved: M- no of bits, N- number of decimal points

Solidity: Addresses and msg object

Account: Externally Owned & Contract

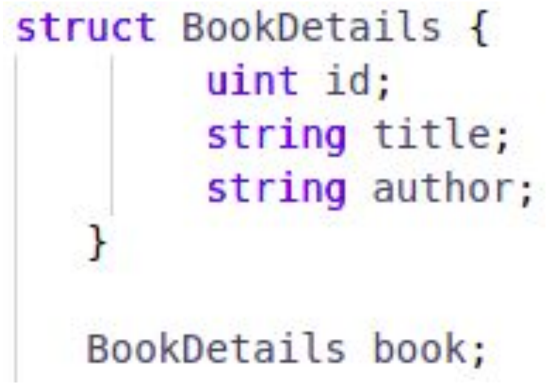
- Addresses are public
- Addresses can hold & transfer ether.
- Externally Owned Account (EoA) is identified by a private key
  - Transactions can be triggered only by an EoA.
  - Msg.sender: address of account which initiated the transaction
  - Msg.value: amount of ether transferred (expressed in wei)
- Contract Account
  - Identified by an address generated after deployment.
  - A contract cannot initiate a transaction by itself.

Solidity: Data Types: Address

- To store Ethereum address of EoA and Contract Account.
- ***`address`*** and ***`address payable`***.
- If an address needs to receive ether , it should be declared as payable.
- address can be converted to address payable explicitly by using payable(<address>).
- Functions
  - balance: returns balance of an account in wei
  - transfer: transfer ether from smart contract account to another account. Transaction will be reverted if transfer fails/get rejected. *Syntax;* `<recipient_address>.transfer(amount_in_wei)`
  - send: similar to transfer,
    - does not revert in case of failure, instead sends a false value

Solidity: struct

- User defined data type used to group same and/or different type of variables.
- Example:



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

   BookDetails book;
```

Solidity: Dynamically sized array

- bytes:

  Store raw data of arbitrary length

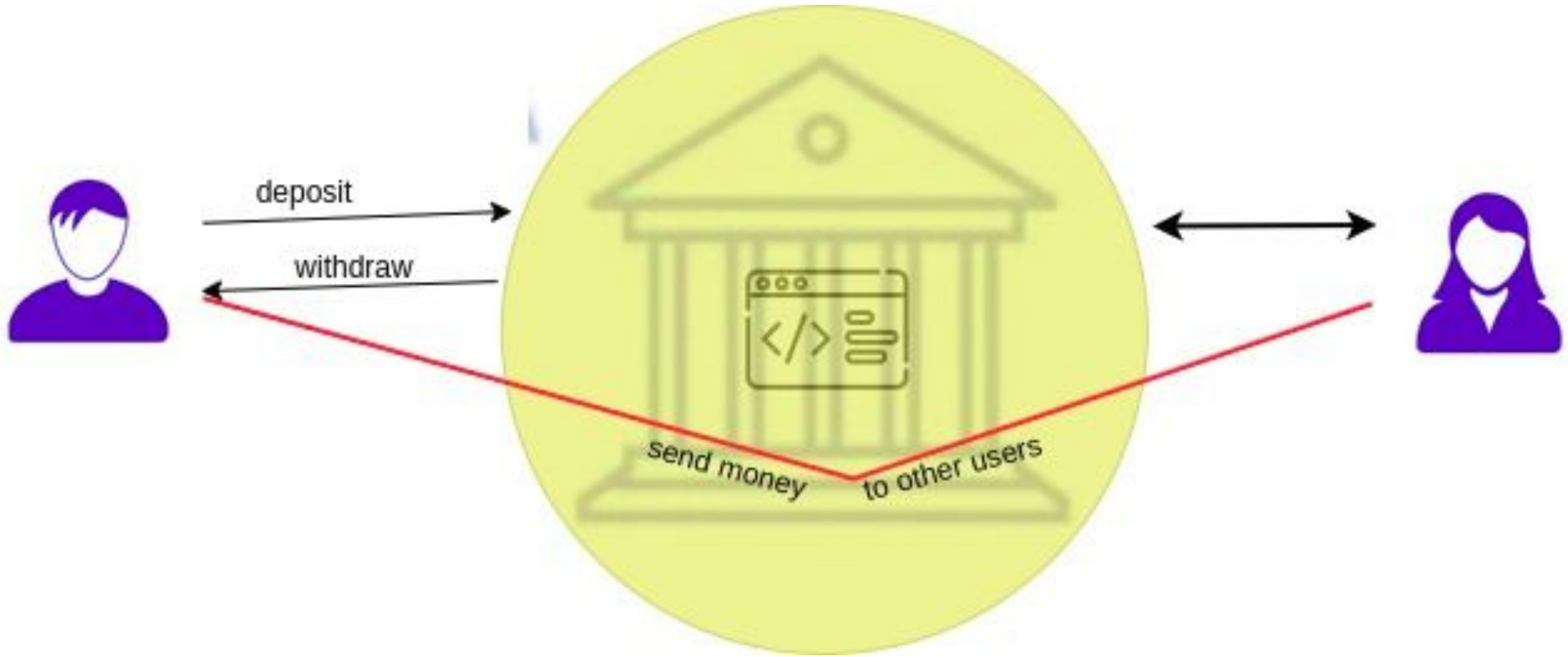
- String

  Dynamically sized UTF-8 encoded strings

Solidity: Arrays

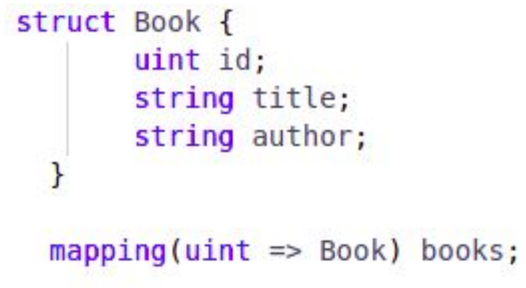
- group together variables of the same type.
- Fixed size array: syntax: type[<size>] <var_name>
- Dynamic size array: type[] <var_name>
- Member functions:
  - Length : number of elements in an array
  - Push: insert an element into an array
  - Pop: remove (delete) element

Bank smart contract



Solidity: Mapping

- ❖ Key-value data structure.
- ❖ Each key has an associated value. A value can be accessed only if its key is known.
- ❖ Syntax: mapping(<*key_datatype*> => <*value_datatype*>*)* <*mappingName*>
- ❖ Example:

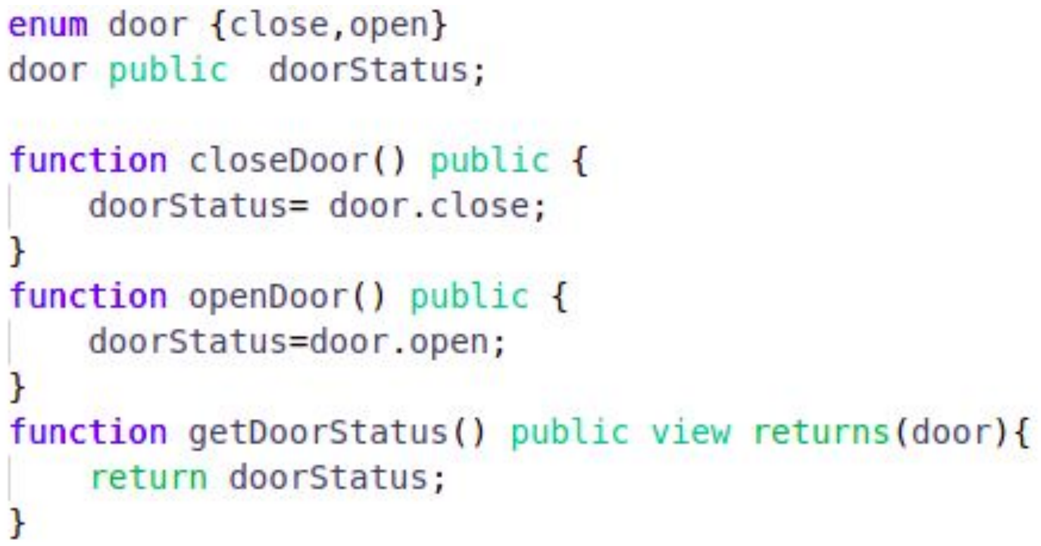


```
struct Book {
        uint id;
        string title;
        string author;
  }

  mapping(uint => Book) books;
```

- ❖ *Arrays*, *struct* or *mapping* data types cannot be used as keys
- ❖ In DApp most of the data are stored with a combination of mapping and struct.

Solidity: Enum

- ❖ User defined data type.
- ❖ Example:



```
enum door {close,open}
door public  doorStatus;

function closeDoor() public {
    doorStatus= door.close;
}
function openDoor() public {
    doorStatus=door.open;
}
function getDoorStatus() public view returns(door){
    return doorStatus;
}
```

Solidity: Advanced Concepts

Solidity: Fixed size byte arrays

- The value types bytes1, bytes2, bytes3, ..., bytes32 hold a sequence of bytes from one to up to 32.
- .length returns length

*[NOTE: **string** and **bytes** are dynamically sized special **arrays**. They are not value types]*

Error Handling

- Atomic transactions
- Keywords: **assert**, **require** and **revert**
- Older versions may be using **throw** (currently *deprecated*)

Error Handling

- Require is used to check conditions related to ***user inputs***
- `require(msg.sender==owner,"Access Denied");`

  If the condition is false, transaction is reverted which**returns remaining gas**

Error Handling

- Assert is used to check conditions related to ***state variables***
-
```
assert(currentBalance + uint8(msg.value) > currentBalance);


If the condition is true, transaction is considered valid.



Other examples:

    Out of bounds

    Invalid conversions
```

Error Handling

- Revert is similar to require

```
if(msg.sender!=owner){
            revert("Access Denied");
       }
```

Function Modifier

❖ If require statements are to be repeated

```
  modifier onlyOwner(){
        require(msg.sender==owner,"Insufficient Privileges");
        _;
   }

// function can be written as

  function setMessage(string memory _message) public onlyOwner{
          message = _message;
   }
```

Deleting a contract

- **delete** is used to set value of a value type to default
- Self destruct is used to delete a contract
-
```
selfdestruct(payable(msg.sender));  // currently deprecated
```

Fallback function

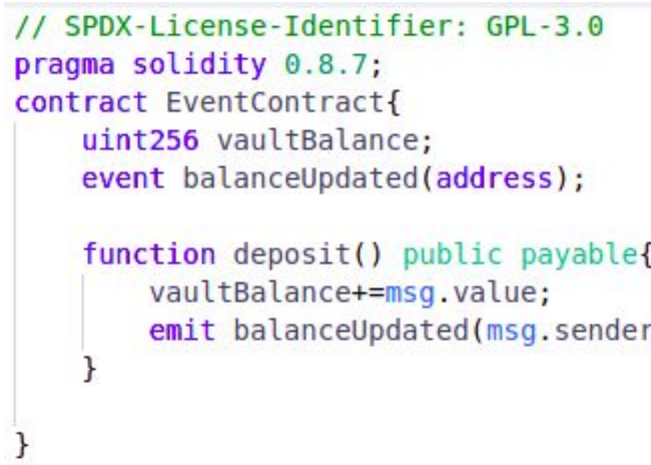
❖ Fallback function is called if no matching function signature is found

❖ It should have **external** visibility

```
fallback () external payable{
    // write the default code here eg: accept the input payment
}

_

receive() external payable { } //A function to receive ether
```

Events

- Function calls do not directly reply back to user
- used as ***return values*** of functions/transactions or to ***store data*** or as ***a trigger***
- `event <eventName>  (parameters)`
- `emit <eventName>(parameters)`
- Arguments may be **indexed** for easy look-up(max 3)
- Event logs are not stored on the chain.
- Event logs are not accessible from within the contract.



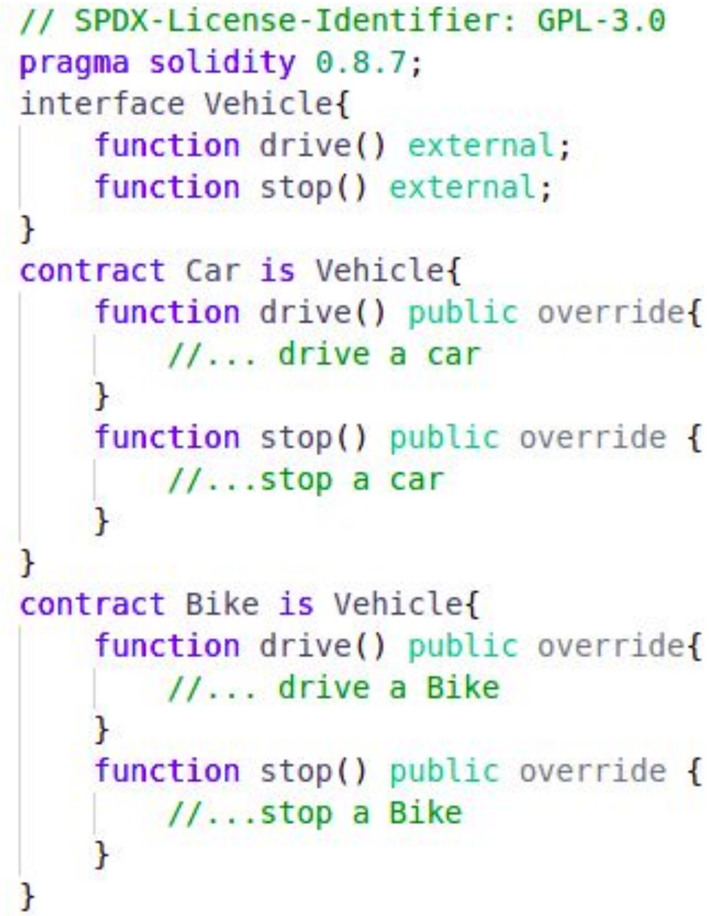
```
// SPDX-License-Identifier: GPL-3.0
pragma solidity 0.8.7;
contract EventContract{
    uint256 vaultBalance;
    event balanceUpdated(address);

    function deposit() public payable{
        vaultBalance+=msg.value;
        emit balanceUpdated(msg.sender
    }

}
```

Inheritance

- One contract can inherit multiple contracts using **is** keyword.
- Allows polymorphism
- Base contract may be accessed by using **super.**
- Even if a contract inherits multiple contract, it will be deployed only once and only one contract address will be generated

Interface

- Template for a contract
- External function declarations
- No local or state variables
- Cannot create an instance of interface



```
// SPDX-License-Identifier: GPL-3.0
pragma solidity 0.8.7;
interface Vehicle{
    function drive() external;
    function stop() external;
}
contract Car is Vehicle{
    function drive() public override{
        //... drive a car
    }
    function stop() public override {
        //...stop a car
    }
}
contract Bike is Vehicle{
    function drive() public override{
        //... drive a Bike
    }
    function stop() public override {
        //...stop a Bike
    }
}
```

Library

- Contains reusable code
- Libraries are stateless !!
- uses the storage from the calling contract.
- Cannot hold ether
- allow you to add functionality to types.

- `import "./MyLibraries.sol";`
- `import {MyLib1, MyLib2 as xyz} from "./MyLibraries.sol";`

Solidity: Contract Type

- Each contract defines its own type
- A variable of the type of a deployed contract can be declared
  - Syntax: ContractA = objA(<addressofContractA>)
- Similar to class-object relation
- use **new** keyword to deploy a new contract.

Thank You

K B A