SOLIDITY NOTES

DATA-TYPES

https://docs.soliditylang.org/en/latest/types.html (AWESOME DOCUMENTATION)

1. **uint** : uint8, uint16, uint32, uint64, uint128 and uint256 (default : uint256)
2. **int** : int8, int16, int32, int64, int128 and int256 (default : int256)
3. **address** : specifically designed to hold 20B, or 160 bits (size of an Ethereum address)
   Types : address and address payable. The difference between the two is that `address payable` can send and transfer Ether while other one can just store address.
4. **bool** : true or false
5. **bytes** : dynamic byte array type that allows you to work with binary data in a hex form.
   For eg :- string public x = “Hello world”
   bytes public y = string(x)
   So, y will be 0x48656c6c6f20776f726c64
   Each two digits corresponding to hex value such as 48 = ‘H’, 65 = ‘e’ …
   It has other types as well like bytes1, bytes2, bytes3 … bytes32.
6. **string** : Strings of chars.

There is also typecasting in Solidity.

COMPLEX DATA-TYPES

1. Enums
2. Structs
3. Arrays
4. Mapping

1) Enum :- Provide names for integral constants which makes the contract better for maintenance and reading. Syntax :-

```
enum <enumerator_name> {
        element 1, element 2,....,element n
}
```

For ex :- enum status {done, pending, rejected} assigns :
status[done] = 0 | status[pending] = 1 | status[rejected] = 2.

Now this ‘status’ becomes a legit (user-defined) datatype. For example :-

- status constant default_value = status.pending;
- status finalValue = status.done;

Bigger Example :-

```
contract Types {
    // Creating an enumerator
    enum weekDays {
        Monday,
        Tuesday,
        Wednesday,
        Thursday,
        Friday,
        Saturday,
        Sunday
    }

    // Setting a default value
    weekDays public dayOfWeek = weekDays.Monday;

    // Defining a function to set value of choice
    function setValue(weekDays choice) public {
        dayOfWeek = choice;
    }
}
```

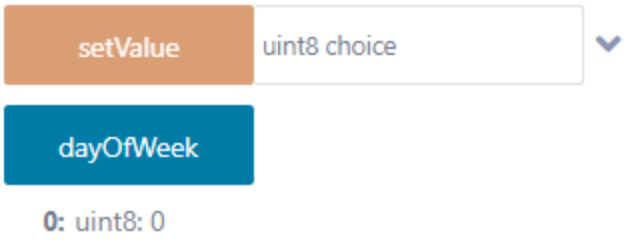


Fig 1: Initial State of dayOfWeek
(Notice that setValue only accepts input in ints from 0 to 6)

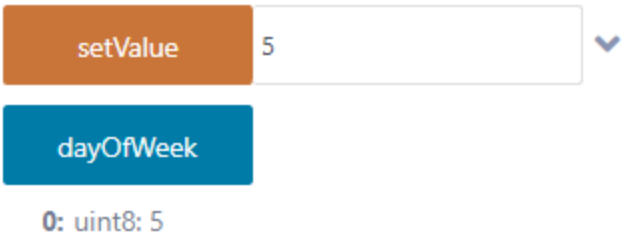


Fig 2 : After using setValue function

2) Structs :- Just like in C/C++. Syntax :-

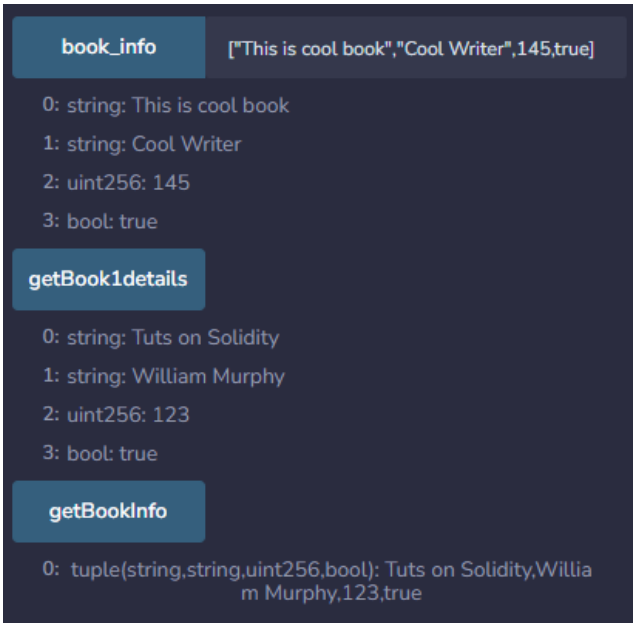
```
struct <struct_name> {
        <type1> <type_name_1>;
        <type2> <type_name_2>;
        <type3> <type_name_3>;
}
```

For example :-

```
// Declaring a structure
struct Book {
    string name;
    string writer;
    uint id;
    bool available;
}

// Declaring a structure object
Book public book1;

// 4 ways of init a struct object
// 1.
constructor(string memory _name, string memory _writer, uint _id, bool _avail){
    book1.name = _name;
    book1.writer = _writer;
    book1.id = _id;
    book1.available = _avail;
}

// 2.
function set_book1() public {
    book1.name = "Tutorials on Solidity";
    book1.writer = "Chris Dannen";
    book1.id = 123;
    book1.available = true;
}

// 3.
Book public book2 = Book("Building Ethereum DApps", "Roberto Infante", 562,
false);

// 4. (Here we change up the ordering too)
Book book1 = Book({
```

```
        name : "Tuts on Solidity",
        writer : "William Murphy",
        id : 123,
        available : true
    });
```

```
Book book1 = Book("Tuts on Solidity", "William Murphy", 123, true);

function getBook1details() public view returns (string memory, string memory, uint, bool) {
    return (book1.name, book1.writer, book1.id, book1.available);
}

function getBookInfo() public view returns (Book memory) {
    return book1;
}

function book_info(Book memory b) public pure returns (string memory, string memory, uint, bool) {
    return (b.name, b.writer, b.id, b.available);
}
```

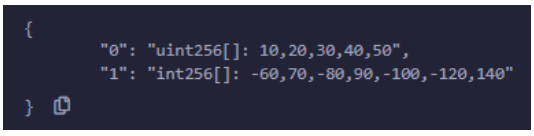


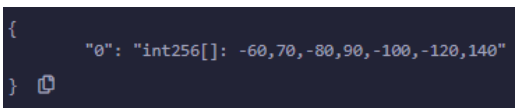
Tuple is written in form of [<item1> , <item2> …]

3) Arrays :- Just like C/C++. Syntax :- <type> <arrayName> [<arraySize>] = <initialization>;
Initialization is optional, can be done through a function as well.

```
uint[6] public data1 = [1,2,3,4,5,6];
uint[6] public data2;

function updateData2() public {
    for(uint i = 0 ; i < 6 ; i++){
        data2[i] = 2*i + 7;
    }
}
// Before updation data2 = [0,0,0,0,0,0] (by default)
// After updation data2 = [7,9,11,13,15,17]
```

```
uint[] data = [10, 20, 30, 40, 50];
int[] data1;

function arrays() public returns(uint[] memory, int[] memory){
    data1 = [int(-60), 70, -80, 90, -100, -120, 140];
    return (data, data1);
}

function dynamic_array() public returns(int[] memory){
    data1 = [int(-60), 70, -80, 90, -100, -120, 140];
    return data1;
}

function firstNumber(int[] memory a) public pure returns(int){
    return a[0];
}

function myfun() public returns(int){
    int[] memory d = dynamic_array();
    return firstNumber(d);
}
```





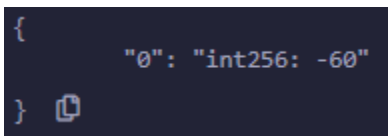


Fig1 : Output of `arrays`

Fig2 : Output of `dynamic_array`

Fig1 : Output of `myfun`

Dynamic Array:

The size of the array is not predefined when it is declared. As the elements are added the size of array changes and at the runtime, the size of the array will be determined. They are created using new keyword. Syntax :- <type> <arrName>[] = new <type>[](<arrSize>);

OR :- <type> <arrName>[] = [item1, item2, item3 ...];

Some new methods for dynamic arrays :- 1. push 2. pop 3. length

```
uint[] public data = [10, 20, 30, 40, 50];

function array_push() public returns(uint[] memory, uint){
    data.push(60);
    data.push(70);
    data.push(80);

    return (data, data.length);
}
```

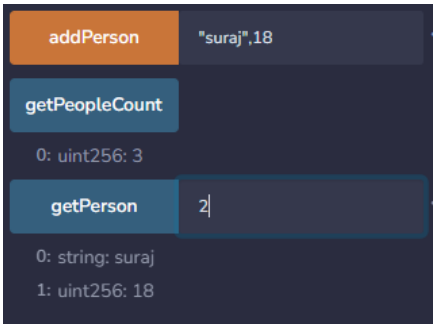
```
{
        "0": "uint256[]: 10,20,30,40,50,60,70,80",
        "1": "uint256: 8"
}
```

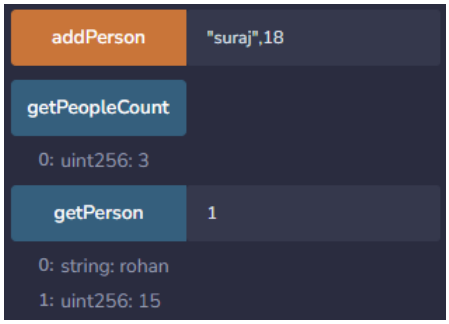
```
uint[] public data = [10, 20, 30, 40, 50];

function array_push() public returns(uint[] memory, uint){
    data.pop();
    data.pop();
    return (data, data.length);
}
```

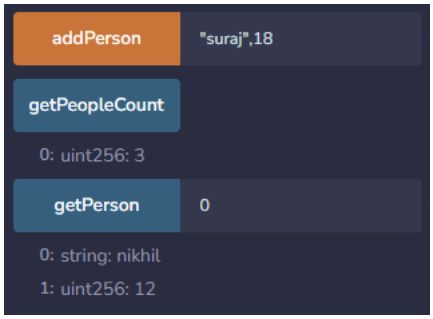
```
{
        "0": "uint256[]: 10,20,30",
        "1": "uint256: 3"
}
```

Arrays of structs :-

```
struct Person {
    string name;
    uint age;
}

// Declare an array of structs
Person[] people;

// Function to add a person to the array
function addPerson(string memory _name, uint _age) public {    infinite gas
    Person memory newPerson = Person(_name, _age);
    people.push(newPerson);
}

// Function to get the number of people in the array
function getPeopleCount() public view returns (uint) {    2467 gas
    return people.length;
}

// Function to get a person by index
function getPerson(uint index) public view returns (string memory, uint) {
    return (people[index].name, people[index].age);
}
```



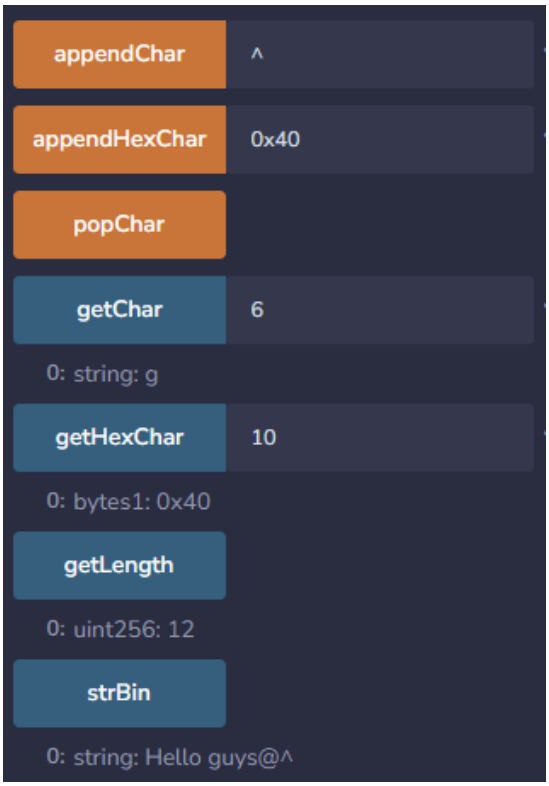


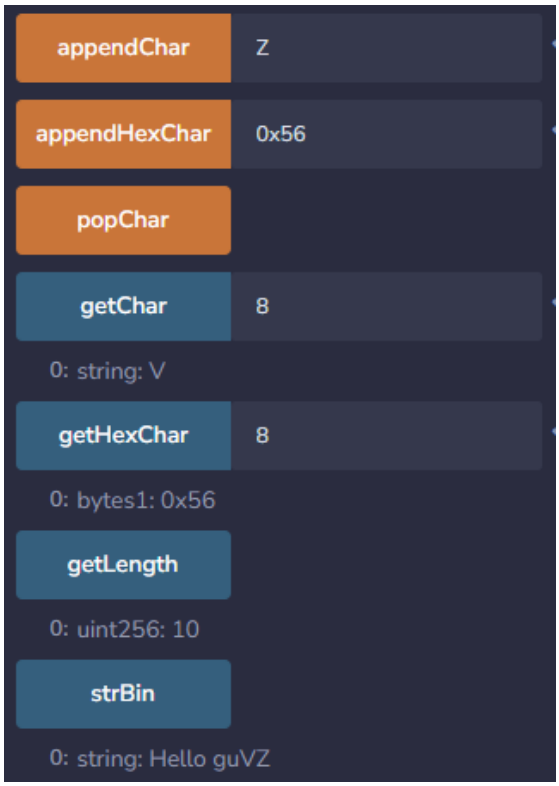


Note : (dynamic bytes-array) : Fixed size is like bytes[SIZE]. For ex :- `bytes32` or `bytes10` etc.
Dynamic is like `bytes x = "Hello world"`
It also has similar functions like dynamic arrays.
Push, Pop, Length

```
bytes b = "Hello guys";
function strBin() public view returns(string memory){    infini
    return string(b);
}
function appendChar(string memory inp) public {    infinite gas
    bytes memory c = bytes(inp);
    b.push(c[0]);
}
function appendHexChar(bytes1 _char) public {    75116 gas
    b.push(_char);
}
function getHexChar(uint ind) public view returns(bytes1){
    return b[ind];
}
function getChar(uint ind) public view returns(string memory){
    bytes memory strBytes = new bytes(1);
    strBytes[0] = b[ind];
    return string(strBytes);
}
function popChar() public {    50618 gas
    b.pop();
}
function getLength() public view returns(uint){    2614 gas
    return b.length;
}
```

Here we assume that `appendChar(string memory inp)` has input of one char. If longer string is input then the 1st char of that `inp` is added only.



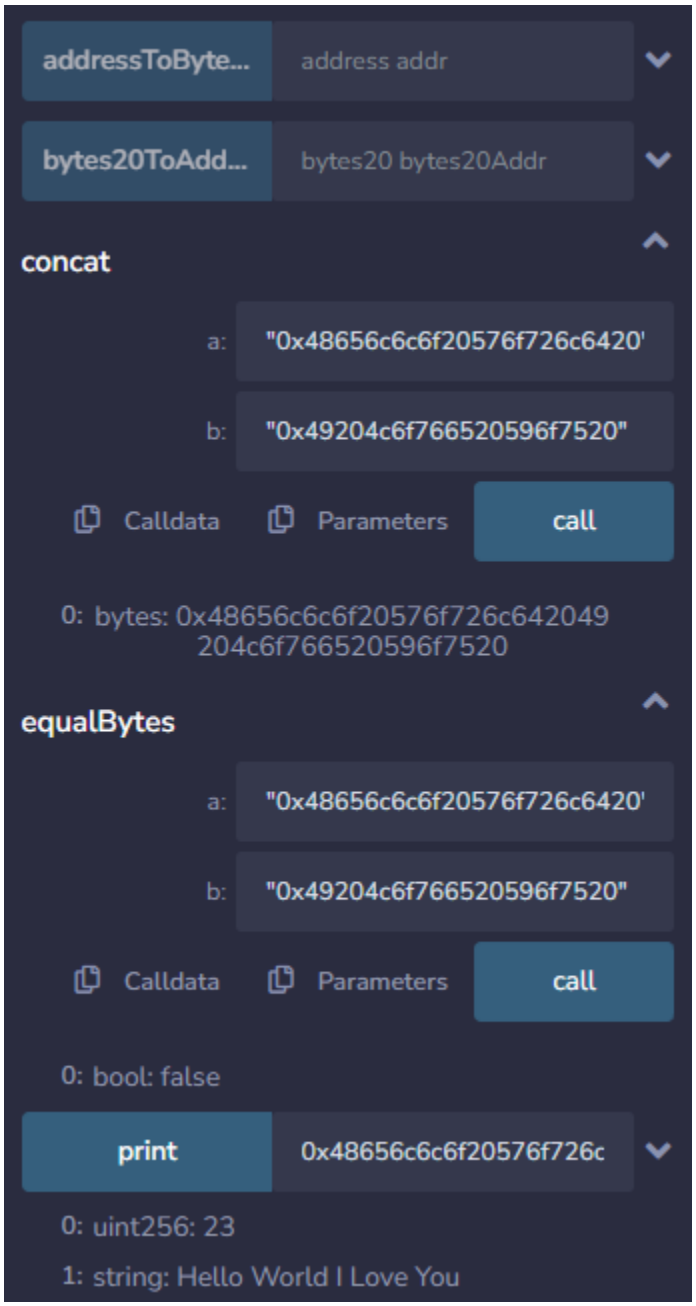



```
function print(bytes calldata data) public pure returns (uint, string memory)
                {
    return (data.length, string(data));
}

// An Ethereum address is 20 bytes long, and you can convert it to a bytes20
                type or vice versa.
// To convert an address to bytes20, you can use an explicit typecast.

// Converts an Ethereum address to bytes20
function addressToBytes20(address addr) public pure returns (bytes20) {
    return bytes20(addr);
}
// Converts a bytes20 value to an Ethereum address
function bytes20ToAddress(bytes20 bytes20Addr) public pure returns (address) {
    return address(bytes20Addr);
}
```

```
// Concatenates two byte arrays
function concat(bytes calldata a, bytes calldata b) public pure returns (bytes
                memory) {
    bytes memory result = new bytes(a.length + b.length);
    for (uint i = 0; i < a.length; i++) {
        result[i] = a[i];
    }
    for (uint j = 0; j < b.length; j++) {
        result[a.length + j] = b[j];
    }
    return result;
}
```

OR

```
function concat(bytes calldata a, bytes calldata b) public pure returns (bytes
                memory) {
    return abi.encodePacked(a, b);
}

// Compares two byte arrays for equality
function equalBytes(bytes calldata a, bytes calldata b) public pure returns
                (bool) {
    if (a.length != b.length) {
        return false;
    }
    for (uint i = 0; i < a.length; i++) {
        if (a[i] != b[i]) {
            return false;
        }
    }
    return true;
}
```


addressToByte...
address addr
bytes20ToAdd...
bytes20 bytes20Addr
concat
a:
"0x48656c6c6f20576f726c6420"
b:
"0x49204c6f766520596f7520"
Calldata
Parameters
call
0: bytes: 0x48656c6c6f20576f726c642049
204c6f766520596f7520
equalBytes
a:
"0x48656c6c6f20576f726c6420"
b:
"0x49204c6f766520596f7520"
Calldata
Parameters
call
0: bool: false
print
0x48656c6c6f20576f726c
0: uint256: 23
1: string: Hello World I Love You

4) Mapping :- Associates keys with values, somewhat like a dictionary or hash map. Syntax :-

mapping(<key datatype> => <value datatype>) <access specifier> <name>;

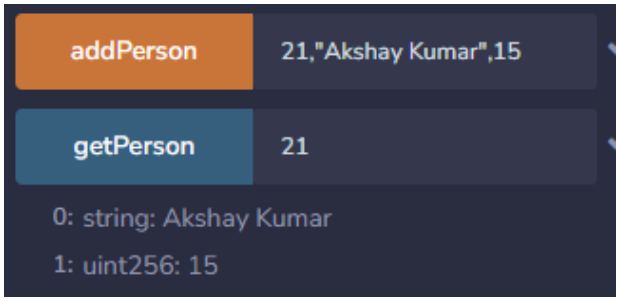
For ex :- mapping (uint => string) public rollNo
It maps roll no. to names of students.
Update function :-

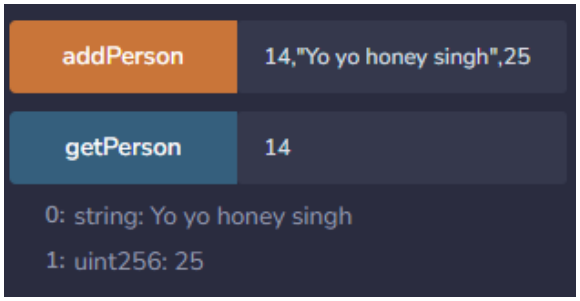
```
function addData(uint no, string memory name) public {
    rollNo[no] = name;
}
```

Note :- Mappings are always stored in ‘storage’ irrespective if they are declared in contract or not. Basically takes up Gas every time.

Mapping of Structs :-

```
// Define a struct
struct Person {
    string name;
    uint age;
}

// Declare an mapping of structs
mapping (uint=>Person) people;

// Function to add a person to the mapping
function addPerson(uint rollno, string memory _name, uint _age) public {
    Person memory newPerson = Person(_name, _age);
    people[rollno] = newPerson;
}

// Function to get a person by mapping index
function getPerson(uint index) public view returns (string memory, uint) {
    return (people[index].name, people[index].age);
}
```





LOOPS

1. for loops 2. while loops 3. do while loops (SYNTAX same as C/C++)
   `continue` and `break` are also available.

IF ELSE-IF ELSE (Same Syntax as C/C++)

State Variables

- Any variable created at the contract level is called the state variable (variables that aren't inside a function) and it is stored permanently on the blockchain i.e., they are persistent and cannot be deleted.
- An important thing to note here is, you'll have to pay some level of gas for every state variable declared. Hence, you should be careful every time you create a state variable.
- Scope of State Variables :- using the following access modifiers for state variables:
  - Public.
  - Internal.
  - Private.

  (Note : There are total 4 types of access modifiers but only 3 of them can be applied to variables.) Syntax :- `<data-type> <access-modifier> <var-name>;`
- Initialization of State Variable :-
  - Initialization at the Time of the Declaration
  - Initialization Using Constructors
  - Initialization Using Setter Method

Local Variables

- Local variables are defined within functions or code blocks. They are only available inside that function or block and deleted when that scope ends.
- They hold temporary data that is only relevant inside a function or block of code.
- They are stored in "memory" by-default (not in contract storage).
- But some variable types like strings, structs, arrays are inherently stored in "storage" so for those we have to use keyword "memory" to prevent conflict as function says that it has to be stored in "memory" but they use "storage". [Memory keyword is to not be used in state variable]

Storage, Memory and Callback

- *Storage* :-
  - It is where all state variables are stored.
  - Because state can be altered in a contract, storage variables must be mutable.

    - However, their location is persistent, and they are stored on the blockchain.
    - Also will incur gas fees since it is written to the blockchain.
    - However, constant state variables are not saved into a storage slot. Rather, they are injected directly into the contract bytecode — whenever those variables are read, the contract automatically switches them out for their assigned constant value.
    - Some data types are by default of ‘storage’ type. Eg :- string, array, structs, mapping, enum etc.
    - It creates a reference to an existing ‘referred’ variable.
- Memory :-
    - Variables that are defined within the scope of a function.
    - They only persist while a function is called, and thus are temporary variables that cannot be accessed outside this scope
    - However, they are mutable within that function.
    - Stored temporarily, kind of like RAM in a computer.
    - Almost negligible gas costs.
    - it creates a copy of the existing ‘referred’ variable.
- Calldata :-
    - Used with function arguments and behave mostly like memory.
    - Calldata is an immutable, temporary location.
    - It is recommended to try to use calldata because it avoids unnecessary copies and ensures that the data is unaltered.
    - Less gas costs as immutable and no-copies.

Some examples :-

```
contract newContract{
    uint[] public arr = [1,2,3];

    function useMemory() public view {
        uint[] memory arrInFunc = arr;
        arrInFunc[0] = 100;
    }

    function useStorage() public {
        uint[] storage arrInFunc = arr;
        arrInFunc[0] = 100;
    }
}
```

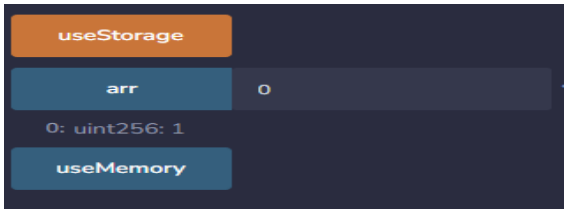


Fig 1 : No function used, initial arr[0]

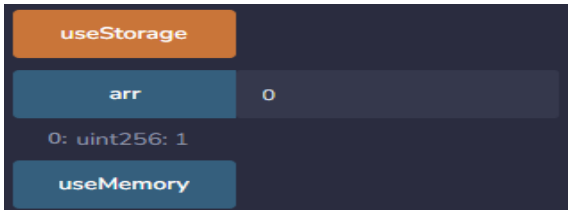


Fig 2 : useMemory function used,then arr[0]

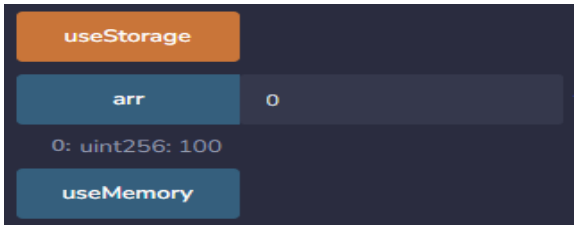


Fig 3 : useStorage function used,then arr[0]

```
contract newContract{
    function useMemory(string memory str) public {}
    function useCalldata(string calldata str) public {}
}
```

Fig 1 : useMemory function used,execution gas cost

Fig 2 : useCalldata function used,execution gas cost

```
function useMemory(string memory str) public {
    // NOTE : error when mem to call
    getInCalldata(str);
    getInMemory(str);
}
function useCalldata(string calldata str) public {
    // In this case, (call to call) no copying of data
    // takes place, a reference to the variable is just
    // simply passed so gas costs relatively low
    getInCalldata(str);
    // In this case, (call to memory) data is copied and
    // passed so gas costs relatively high
    getInMemory(str);

    // In separate function :
    //     getInMemory - execution cost : 514 gas
    //     getInCalldata - execution cost : 482 gas
}

function getInMemory(string memory str) public {}
function getInCalldata(string calldata str) public {}
```

Global Variables

Very awesome doc : https://docs.soliditylang.org/en/latest/units-and-global-variables.html

Pure and View

- **View function :** declares that no state variable will be changed.
- **Pure function :** declares that no state variable will be changed or read.

Constructor

Solidity provides a constructor declaration inside the smart contract and it invokes only once when the contract is deployed and is used to initialize the contract state. (Not compulsory to write)

Syntax:
```
constructor(parameters){
}
```

Constructors are very useful in a smart contract, a parameter value can be defined at the run time and can also restrict the method call. For example :-

```
address public owner;

constructor() {
    owner = msg.sender;
    // Set the `owner` to address of sender (person who deploys contract)
}
```

ERROR HANDLING

1. **Require** :-
   - The 'require' statements declare prerequisites for running the function i.e. it declares the constraints which should be satisfied before executing the code.
   - It accepts a single argument and returns a boolean value after evaluation, it also has a custom string message option.
   - If false then an exception is raised and execution is terminated.
   - The **unused gas is returned back** to the caller and **the state is reversed to its original state**.
   - Examples :-

```
string public result = "";
function checkInput(int input) public {
    require(input > 0);
    // If input < 0 then exception raised without explanation
    require(input <= 255, "Overflow for uint8");
    // If input >= 255 then exception raised with explanation
    result = "Input is Uint8";
}
```

2. **Assert** :-
   - Just like "require". But the only difference is : Instead of returning the unused gas, it consumes the entire gas supply and the state is then reversed to the original state.
   - The primary purpose of assert is to catch critical issues and make sure that the contract state remains secure. Gas efficiency is not a primary concern when using assert.
   - Assert should only be used to test for internal errors, and to check invariants.
   - For example :- Ensuring that a value, which is expected to remain unchanged, remains consistent over the entire lifetime of the contract. This could include verifying the constancy of the contract owner's address or the account balances of users interacting with the contract.

- Syntax is :- `assert(<condition>);` [Note that no error string unlike 'require']

```
bool public result = false;
function checkSumOverflow(uint num1, uint num2) public {
    assert(num1 + num2 <= 255);
    result = true;
}
```

3. **Revert** :-
   - This statement is similar to "require". But it does not evaluate any condition and is just used in if-else case.
   - Calling a revert statement implies an exception is thrown, the unused gas is returned and the state reverts to its original state.
   - Sometimes can be used with custom error-raising functions.
   - Syntax and examples :-
     - Custom error-handler function

```
address owner = msg.sender;
// Here 'owner' is 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
// We try to access with 0x617F2E2fD72FD9D5503197092aC168c91465E7f2
error errorHandle(address,string);

function checkAccess() public view {
    if(msg.sender != owner){
      revert errorHandle(msg.sender,"Not allowed access");
    }
}
```

```
Error provided by the contract:
errorHandle
Parameters:
{
  "0": {
    "value": "0x617F2E2fD72FD9D5503197092aC168c91465E7f2"
  },
  "1": {
    "value": "Not allowed access"
  }
}
```

     - Normal error (default)

```
string public result = "";
function checkInput(int input) public {
    if(input < 0){revert("Input is -ve");}
    else if(input > 255){revert("Overflow for uint8");}
    result = "Input is in Uint8";
}
```

Function Modifiers

Function behavior can be changed using function modifiers.
Helps in reducing functional code repetition.
Merge Wildcard (aka _; symbol) is replaced by the function definition during execution.

- In other words, after this wildcard has been used, the control is moved to the location where the appropriate function definition is located.
- This symbol is mandatory for all modifiers.
- The modifier may contain this wildcard anywhere.
- When the wildcard is placed at the end of the modifier, the condition is verified and the appropriate function is executed if it is satisfied.
- When it is placed at the beginning, the appropriate function is executed first followed by the condition verification.

Before using : func modifier

```
function func1() public {    144
    // Common code (Before Part)
    // Code for func1 ONLY
    // Common code (After Part)
}
function func2() public {    166
    // Common code (Before Part)
    // Code for func2 ONLY
    // Common code (After Part)
}
function func3() public {    122
    // Common code (Before Part)
    // Code for func3 ONLY
    // Common code (After Part)
}
```

After using : func modifier

```
modifier commonFunc() {
    // Common code (Before Part)
    _;
    // Common code (After Part)
}
function func1() public commonFunc {
    // Code for func1 ONLY
}
function func2() public commonFunc {
    // Code for func2 ONLY
}
function func3() public commonFunc {
    // Code for func3 ONLY
}
```

Using Modifiers with Argument

```
struct employee {
    uint emp_id;
    string emp_name;
}
modifier isExperienced(uint exp){
    if(exp >= 5)
        _;
    else
        revert("Min. 5 yrs of exp needed");
}
employee e;
function enterDetails (    infinite gas
    uint applicantId,
    string memory applicantName,
    uint applicantExp
) public isExperienced(applicantExp) {
    e.emp_id = applicantId;
    e.emp_name = applicantName;
}
```

Events

Stores the arguments passed in the transaction logs when emitted.
Generally, events are used to inform about the current state of the contract, with the help of the logging facility of EVM.
For example :-

```
event NewTrade(
    uint256 date,
    address from,
    address to,
    uint256 amount
);

function trade(address to, uint256 amount) public {
    emit NewTrade(block.timestamp, msg.sender, to, amount);
}
```

```
logs    [
            {
                    "from": "0x48e24370326d1695566b56b49cBB60806a243A48",
                    "topic":
        "0xa6b5ddd331f9dc412a8c258207b1c66f53c1740c72628d9913aafcb6b28d8f73",
                    "event": "NewTrade",
                    "args": {
                            "0": "1698335845",
                            "1": "0x17F6AD8Ef982297579C203069C1DbfFE4348c372",
                            "2": "0x17F6AD8Ef982297579C203069C1DbfFE4348c372",
                            "3": "20",
                            "date": "1698335845",
                            "from": "0x17F6AD8Ef982297579C203069C1DbfFE4348c372",
                            "to": "0x17F6AD8Ef982297579C203069C1DbfFE4348c372",
                            "amount": "20"
                    }
            }
        ]
```

You can use indexing in events to make it easier to filter and search for specific events.Then, logs are stored in a way that allows for more efficient retrieval and filtering.
IMP: We can add **at most 3 indexes** in one event.
Example :-

```
event NewTrade(
    uint256 indexed date,
    address from,
    address indexed to,
    uint256 indexed amount
);
```

Payable

Functions and addresses declared payable can receive ether into the contract.

```
contract Payable {
    // Payable address can send Ether via transfer or send
    address payable owner;

    // Payable constructor can receive Ether
    // Useful in case when you want to send some amt of ether
    // into the contract and want it to stay constant since constructor
    // runs only once. And then keep all other functions non-payable
    constructor() payable {    70941 gas 46600 gas
        owner = payable(msg.sender);
    }

    // Function to deposit Ether into this contract.
    // Call this function along with some Ether.
    // The balance of this contract will be automatically updated.
    function deposit() public payable {}    142 gas

    // Call this function along with some Ether.
    // The function will throw an error since this function is not payable.
    function notPayable() public {}    144 gas

    // Returns balance of the contract
    function getBalance() public view returns (uint){    317 gas
        return address(this).balance;
    }
}
```

Sending Ether Methods

You can send Ether to other contracts by

- transfer (2300 gas, throws error)
  - Revert changes to state variables in case the transaction fails.
  - Returns unused gas in transaction.
- send (2300 gas, returns bool)
  - Uses up all gas even if transaction fails due to 2300 gas limit
  - Doesn’t revert changes to state variables in case transaction fails so needs ‘require’
- call (forward all gas or set gas, returns bool)
  - Can set gas limit ourselves
  - Doesn’t revert changes to state variables in case transaction fails so needs ‘require’

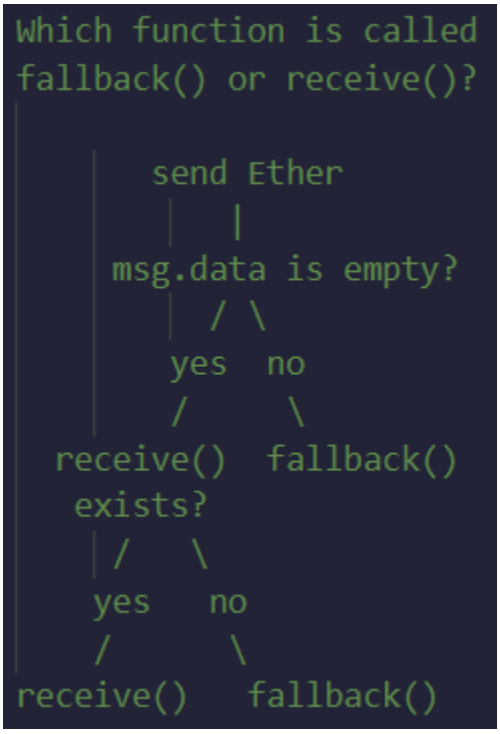
‘call’ is the recommended method to use after Dec 2019.
‘require’ is also recommended to be used with any of these to revert changes to state variables by raising an error in case the gas limit is exceeded.

Receiving Ether

A contract receiving directly Ether must have at least one of the functions below

- receive() external payable
- fallback() external payable

receive() is called if msg.data is empty, otherwise fallback() is called.



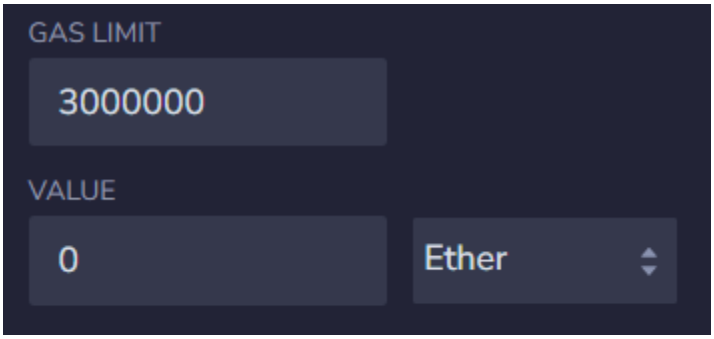


Fig 1 : Gas limit and amt of ethers to be deposited can be set here (msg.value)

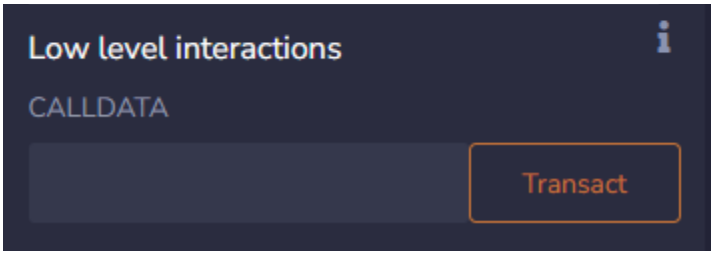


Fig 2 : Calldata can be put here (msg.data)

```
contract SendEther {
    function sendViaTransfer(address payable _to) public payable {
        _to.transfer(msg.value);
    }

    function sendViaSend(address payable _to) public payable {
        bool sent = _to.send(msg.value);
        require(sent, "Failed to send Ether");
    }

    function sendViaCall(address payable _to) public payable{
        (bool sent, bytes memory data) = _to.call{value: msg.value}("");
        require(sent, "Failed to send Ether");
    }
}
```

```
event Log(string);
receive() external payable { emit Log("receive"); }
fallback() external payable { emit Log("fallback"); }

// in case of no msg.data and no msg.value => "receive"
// in case of no msg.data => "receive"
// in case of no msg.value => "fallback"
// in case of both present => "fallback"
```

```
contract Caller {
    function callExample(    undefined gas
        address payable target,
        uint256 valueInWei,
        bytes memory data,
        uint256 gasLimit
    ) public returns (bool success, bytes memory result) {
        (success, result) = target.call{value: valueInWei, gas: gasLimit}(data);
    }
}

// target:     The address of the external contract or function you want to call.
// valueInWei: The amount of Wei to send along with the call.
// data:       The data that specifies the function and its arguments to be called in the external contract.
// gasLimit:   The gas limit you want to set for the external function call.
```

```
function callExample(    undefined
    address payable target,
    uint256 valueInWei,
    uint256 gasLimit
) public returns (bool res){
    (res,) = target.call{
                value: valueInWei,
                gas: gasLimit
            }("");
}
```

Visibility

Best given here : https://solidity-by-example.org/visibility/

Public	Private	Internal	External
Outside	x	x	Outside
Within	Within	Within	x
Derived	x	Derived	Derived
Other	x	x	Other