Solidity

What is Solidity?

- Solidity is an object-oriented programming language for implementing smart contract for the ethereum blockchain.
- It is a high-level statically typed programming language
- It is Case sensitive.
- With solidity you can create contract for use such as voting, crowdfunding, blind auctions and multi-signature wallets.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract demo {

}
```

- // SPDX-License-Identifier: GPL-3.0 This comment allows you to make your smart contract private, but it's a good practice make your smart contract public.
- pragma solidity >=0.7.0 <0.9.0; Writing your contract like this allows you to select a range of compilers that can execute your smart contract.

Public

Public is a visibility specifier. Declaring a variable as public generates an automatic getter function that allows external contracts to read the value of the variable.

Type of variables

1. State variables (global) Any variable that is declared directly withing a contract is a state variable.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract demo {
    unint num = 5;

    // If you want to be able to access the value of num after you have deployed
it, make it public
    unint pulic num = 5;
}
```

- They are permanently stored on the blockchain.

- State variable cost gas (expensive).
- Reading of state variable is free but writing to it is costly.

2. Local variables

These are variable declared within a function.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract demo {

    function sum() public pure return (uint) {
        uint a;
        uint b;
        return a+b;
    }
}
```

- Local variable do not cost any amount of gas (writting or reading).
- They do not get permanently stored on the blockchain.
- They are kept on the stack, not on storage.

Functions

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract demo {
    uint public num;

    function setter(uint _num) public {
        num = _num;
    }

    function getter() public view returns(uint) {
        return num;
    }
}
```

- When you call a setter function it creates a transaction that needs to be mined and costs gas because it changes the blockchain.

```
function local() public returns (address, uint256, uint256) {
  uint256 i = 345;
  myNumber = 1;

  i += 45;
```

```
    address myAddress = address(1);

    return( myAddress, myNumber, i);
  }
```

// result: { "0": "address: 0x00000000.....001; "1"" "uint256": 345", "2": "uint256: 390 }

Function Types

- Declarative Function
    - Modifying State Variable
    - Emitting Events
    - Creating other Contracts
    - Using self-destruct
    - Sending Ether via calls
    - Marked Pure or View
    - Assembly Certain Opcodes
    - Using low-level calls
- Non Declarative Function

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Function {

  // Functions can return multiple values
  function returnMany() public pure returns(uint, bool, uint) {
    return (1, true, 2);
  }

  // Return values can be named
  function named() public pure returns(uint x, uint b, uint y) {
    return (1, true, 2);
  }

  // Return values can be assigned to their name
  // In this case the return statement can be omitted
  function assigned() public pure return(uint x, bool b, uint y) {
    x = 1;
    b = true;
    y = 2;
  }

  // Use destructuring assignment when calling another
  // function that returns multiple values
  function destructuringAssignments() public pure returns(uint, bool, uint, uint,
uint) {
    (uint i, bool b, uint j) = returnMany();
```

```
    // Values can be left out
    (uint x, , uint y) = (4, 5, 6);

    return (i, b, j, x, y);
  }


  // Cannot use map for either input or output


  // Can use array for input
  function arrayInput(uint[] memory _arr) public {}


  // Can use array for output
  uint[] public arr;

  function arrayOutput() public view returns(uint[] memory) {
    return arr;
  }

}

// Call function with key-value inputs
contract XYZ {
  function someFuncWithManyInputs(uint x, uint y, uint z, address a, bool b,
string memory c) public pure return(uint) { }

  function callFunc() external external pure returns (uint) {
    return someFuncWithManyInputs(1, 2, 3, address(0), true, "c")
  }

  function callFuncWithKeyValue() external pure returns (uint) {
    return someFunctWithManyInputs({a: address(0), b: true, c: "c", x: 1, y:1,
z:2})
  }
}
```

Events

Imagine you want to update a data when you an event occurs. When events are used to store data in the blockchain, it costs very less gas compared to using storage.

```
// SPDX-Licence-Identifier: MIT

pragma solidity ^0.8.13;

contract Event {
  // Event declaration
  // Up to 3 parameters can be indexed.
  // Indexed parameters helps you filter the logs by the indexed parameter
```

```
    event Log(address indexed sender, string message);
    event AnotherLog();

    function test() public {
     emit Log(msg.sender, "Hello, World!");
     emit Log(msg.sender, "Hello, EVM!");
     emit AnotherLog();
    }
}
```

view vs pure vs when neither is used

- If you are reading the value of a state variable use view.
- If your function only uses local variables use pure.
- If you are changing the value of a state variable, you wouldn't write any of the two (view or pure).

```
function setter(uint _num) public {
  num = _num;
}
```

- The only reason why we are using view below is that we are accessing a state variable.

```
uint num = 2 3;
function getter() public view returns(uint) {
        return num;
    }
```

- If you are accessing a local variable, use pure.

```
function random() public pure {
    uint abc;
}
```

Constructor

This is a function that is called by default. This function is executed only at the time of contract creation.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract demo {
    uint public num = 3;

    constructor (){
```

```
        num = 10;
    }
}
```

When the contract above gets deployed it will num will return 10.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract demo {
    uint public num = 3;

    constructor (uint _num){
        num = __num;
    }
}
```

Assuming we add a parameter to the constructor, we will be required to provide a value for _num immediately before the contract is deployed.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.13;

// Base contract X
contract X {
  string public name;

  constructor(string memory _name) {
    name = _name;
  }
}

// Base contract Y
contract Y {
  string public text;

  constructor(string memory _text) {
    text = _text;
  }
}

// There are 2 ways to initialize parent contract with parameters.

// Pass the parameters here in the inheritance list.

contract B is X("Input to X"), Y("Input to Y") {}

contract C is X, Y {
  // Pass the parameter here in the constructor
```

```
  // Similar to function modifers
  construct(string memory _name, string memory _text) X(_name) Y(_text) {}
}



// Parent constructors are always called in the order of inheritance
// regardless of the order of parent contracts listed in the
// constructor of the child contract

// Order of constructors called:
// 1. X
// 2. Y
// 3. D
contract D is X, Y {
  constructor() X("X was called") Y("Y was called"){}
}

// Order of constructors called:
// 1. X
// 2. Y
// 3. E
contract E is X, Y {
  constructor() Y("Y was called") X("X was called") {}
}
```

Data Types

- Boolean
- Integer
- Fixed Point Numbers
- Address
- Bytes and Strings
- Enums

Reference Data Types

- Arrays
- Struct
- Mapping

Integer Data Type

1. int - can hold positive and negative numbers (int8 - int256)
2. uint - can hold only positive numbers (int8 - uint256)

- By default, int and uint are initialized to zero.
- Overflow get detected at compile time.

Integer Range

NB: uint is the same uint256 int is the same as int256

int

int8 store any integer value within -128 to 127 int16 can store any integer value within -32768 to +32767 int128 int256

The formula - -2^(n-1) to 2^(n-1) - 1

Example (int8) - -2^(8-1) - 1 to -2^(8-1) - 1

uint

uint8 can store any integer value within 0 to 255 uint16 can store any integer within 0 to 65535 uint128 uint256

The formula - 0 to 2^(n) - 1

Example (uint8) - 0 to -2^(8) - 1 = 0 to 255

Finding min and max value of an integer data type

```
int public minInt = type(int).min;
int public maxInt = type(int).max;
```

bytes

Storing strings as bytes is one way of reduce gas fee (More Memory efficient).

```
bytes1 public a; //0x00
bytes1 public b; //0x00

a = 0xb5; // Easily under stood by the evm [010101010......]
a = 0xb5; // [010101010......]

bytes public myName = "Hey Korede" // 0x6864b........
```

Address

```
address public hey; //0x00000000........
address public addr = 0x5B38Da6a.....
```

Default Values of each data type

```
bool public defaultBool; // false
uint public number; // 0
int public defaultInt; // 0
address public hew; // 0x00000000000
```

Constants

By saying that a variable is a constant, it's that the variable's value will never change after a value has been assigned to it.

Using constants reduces gas.

```
// gas: 130242 without declaring a constant
address public myAdd = 0x5b38.....;


// gas: 104187 after declaring a constant
address public contstant MY_ADDR = 0x5838....;
```

Loops

1. while loop
2. for loop
3. do while loop

NB: Loop cannot be used directly within a contract, it must be declared in a function.

Using while loop

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

    function whileLoop() public pure returns(uint){
        uint sum;
        uint count;
        while (count<5) {
            sum=sum+count;
            count=count+1;
        }
        return sum;
    }
}
```

Using for loop

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

    function forLoop() public pure returns(uint){
        uint sum;

        for (uint count=0; count<5; count++) {
            sum=sum+count;
        }
        return sum;
    }
}
```

Using for Do While

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

    function doWhileLoop() public pure returns(uint){
        uint sum;
        uint count;

        do {
            sum=sum+count;
            count=count+1;
        } while (count<5)
        return sum;
    }
}
```

Conditionals

We can not use if else statement at contract level only within a function.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract checking(uint a, uint b) public pure returns(uint){
    if(a>b){
        reuturn 1;
    } else if (a == b){
        return 2;
    } else {
        return 0;
    }
}
```

Using ternary operator

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract checking(uint a, uint b) public pure returns(uint){

    return a>b ? 1 : a == b ? 2 : 0;
}

## Bool Data Type

- true
- false

e.g. bool public value = true.

- By default value is false if not initialized.

```sol
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract demo {
    bool public value = true;

    function checkEven(uint a) public pure returns(bool) {
        if (a%2==0){
            return true;
        }
        return false;
    }
}
```

Handling Error

Require Statement

When require statement is used, it prevents the next line from run (or transaction from executing) if its condition is false, by throwing an error.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract demo{
    function isZero(uint a) public pure returns(bool){
        require(a == 0, "a is not equal to zero")
        return true
    }
}
```
```

Revert Statement

Using if statement as require statement

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract demo {
  function isZeroIf(uint a) public pure returns(bool) {
    if(a==0) {
      return true;
    }

    revert("a is not equal to zero")

  }
}
```

Assert

```
contract AssertStatement {
  bool public result;

  funtion checkOverFlow(uint256 _num1, uint256 _num2) {
    uint256 sum = _num1 + _num2;
    assert(sum <= 255);
    result = true;
  }
}
```

Modifier

Modifier are codes that can be run before and after the function call

They are used for 3 reasons

- Restrict access
- Validate inputs
- Guard agains reentrancy hack

Using modifiers to avoid repetition

Assuming we have a smart contract like this.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
```

```
contract demo {

  function f1() public pure returns(uint) {
    require(true == true, "True is not true");
    return 1;
  }

  function f2() public pure returns(uint) {
    require(true == true, "True is not true");
    return 2;
  }

  function f3() public pure returns(uint) {
    require(true == true, "True is not true");
    return 3;
  }

  function f4() public pure returns(uint) {
    require(true == true, "True is not true");
    return 4;
  }

}
```

Solution

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract demo {

  modifier isTrue () {
    require(true == true, "True is not true");
    _;
  }
  function f1() public pure isTrue returns(uint) {
    return 1;
  }

  function f2() public pure isTrue returns(uint) {
    return 2;
  }

  function f3() public pure isTrue returns(uint) {
    return 3;
  }

  function f4() public pure isTrue returns(uint) {
    return 4;
  }
}
```

The `_;` copies the code within the function that the modifier is applied in, so that the content of the function can begin execution immediately after the statements within the modifier is executed.

Using modifiers with parameters

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract demo {
  modifer isEven(uint a) {
    require(a%2 == 0, "a is not even");
    _;
  }

  function f1(uint a) public pure isEven(a) returns(bool) {
    return true;
  }
}
```

Other Examples

```
modifier onlyOwner)() {
  require(msg.sender == owner, "Not Owner");
  _;
}

modifier validateData(address _addr) {
  require(_addr != address(0), "Not valid address");
}

function changeOwner(address _newOwner) public onlyOwner validateData(_newOwner) {
  owner = _newOwner;
}

modifier noReEntrancy() {
  require(!locked, "No Re-entrancy");
  _;
}
```

Reentrancy Attack

A reentrancy attack is a type of vulnerability that can occur in smart contracts on blockchains like Ethereum. It allows an attacker to repeatedly call and re-enter a function before the first invocation is completed, potentially draining funds from the contract.

The root cause of the reentrancy vulnerability is the order of operations: the contract sends funds before updating its internal state. By exploiting this ordering issue, an attacker can repeatedly call the vulnerable function and receive multiple payments before the balance is correctly updated.

```
uint256 public x = 10;
  bool public locked;

  modifier nonReentrancy() {
      require(!locked, "ReentrancyGuard: reentrant call");
      locked = true;
      _;
      locked = false;
  }

  function decrement(uint256 i) public nonReentrancy {
    x -= 1;

    if(i > 1) {
      decrement(i - 1);
    }
  }
```

- locked = true;: Once the function is called, it sets the locked variable to true, indicating that the function is currently being executed. This prevents other callers from invoking the function concurrently.
- _;: This is a placeholder for the body of the function that the modifier is applied to. It represents where the actual function code will be executed.
- locked = false;: After the function body (_😉 has executed, the locked variable is set back to false, indicating that the function has completed its execution and is now available to be called by other parties.

Address Data Type

The address data type is usually used to store an address.

```
address public addr = "0xBE4024a7461933F930DD3CEf5D1a01363E8f284"
```

- The address type is a 160-bit value that does not allow any arithmethic operations.

Basic Data Types

Reference Data Type

- strings
- arrays
- mappings

- struct
- Data types such as Arrays, Structs, Mapping are known as reference data type.
- If we use a referenct type, we always have to explicitly provide the data area where the type is stored.
- Every reference type has an additional annotation the "data location", about where it is stored.
- There are three data locations: **memory, storage and calldata**.

NB: When declaring a reference data type note that if it doesn't have a `public` access specifier it will have a the `memory` data location.

Data locations

1. Memory - Lifetime is limited to an external function call.
2. Storage - The location where state variables are stored, where the lifetime is limited to the lifetime of a contract.
3. calldata - Special data location that contains the function arguments. (Similar to memory in lifetime)

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract DataLocations {
  uint[] public arr;
  mapping(uint => address) map;
  struct MyStruct {
    uint foo;
  }

  mapping(uint => MyStruct) myStructs;

  function f() public {
    // Call _f with state variables
    _f(arr, map, myStructs[1]);

    //get a struct from mapping
    MyStruct storage myStruct = myStructs[1];

    // Create a struct in memory
    MyStruct memory myMemStruct = MyStruct(0);
  }

  function _f(
    uint[] storage _arr,
    mapping(uint => address) storage _map,
    MyStruct storage _myStruct
  ) internal {
    // do something with storage variables
  }

  // You can return memory variables
```

```
    function g(uintl[] memory _arr) public returns (uint[] memory) {
      // do something with storage array
    }

    function h(uint[] calldata _arr) external {
      // do something with calldata array
    }
  }
```

Types of Array

- Fixed Array
- Dynamic Array

Fixed Array

This is an array that has a fixed size at compile time

- uint[number_of_elements] public arr;

e.g. uint[5] public arr; by default [0, 0, 0, 0]

example 2: uint[5] public arr = [10, 20, 30, 40, 50]

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract demo {
  uint[5] public arr = [10, 20, 30, 40, 50];

  function insert(uint index, uint _item) public {
    arr[index] = _item;
  }

  function returnArr(uint index) public view returns(uint) {
    return arr[index]
  }

  function returnAllElements() public view returns(uint[5] memory) {
    return arr;
  }

  function getLength() public view returns (uint) {
    return arr.length;
  }

  function remove(uint index) public {
    // Delete does not change the array length
    // It resets the value at index to it's default value
    // in this case 0
    delete arr[index]
```

```
  }
}
```

Any time you want to use an array(reference data type) as a parameter in a function alway specify the memory, because functions are temporal storage.

Memory keyword cannot be used at contract level.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract demo {
  uint[3] public arr = [10, 20, 30]; // arr is created at storage area

  function fmemory() public view{
    // view is used in this function because we are assigning a state variable to
a local variable
    // local variable array
    uint[3] memory arr1 = arr; //
    arr1[0] = 90;
  }

  function fstorage() public {
    // view is not used here because arr is assign to another variable on a
storage level which is a pointer to arr
    uint[3] storage arr2 = arr; //arr2 is a pointer to arr
    arr2[0] = 90;
  }

}
```

Dynamic Array

Is an array whose size is not fixed at compile time.

- uint[] public arr;
- To insert an element we use arr.push(element)
- To remove an element we use arr.pop(). It removes the last element every time it is executed.
- To find the length of an array we use arr.length(). It returns the length of the array in uint data type.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract demo {
  uint[] public arr;

  function insert(uint item) public {
    arr.push(item);
```

```
  }

  function removeLast() public {
    arr.pop();
  }

  function lengthArr() public view returns(uint) {
    return arr.length;
  }

  function returnAll() public view returns(uint[] memory){
    return arr;
  }
}
```

Removing the value in an array (By Shifting): METHOD 1

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract ArrayRemoveByShifting {
  // [1, 2, 3] -- remove(1) --> [1, 3, 3] --> [1,3]
  // [1, 2, 3, 4, 5, 6] -- remove(2) --> [1, 2, 4, 5, 6, 6] --> [1, 2, 4, 5, 6]
  // [1, 2, 3, 4, 5, 6] -- remove(0) --> [2, 3, 4, 5, 6, 6] --> [2, 3, 4, 5, 6]
  // [1] -- remove(0) --> [1] --> []

  uint[] public arr;

  function remove(uint _index) public {
    require(_index < arr.length, "Index out of bound");

    for (uint i = _index; i < arr.length - 1; i++) {
      arr[i] = arr[i + 1];
    }

    arr.pop();
  }

  function test() external {
    arr = [1, 2, 3, 4, 5];
    remove(2); // remove the value at index 2 which is no. 3

    // [1, 2, 4, 5]
    assert(arr[0] == 1);
    assert(arr[1] == 2);
    assert(arr[2] == 4);
    assert(arr[3] == 5);
    assert(arr.length == 4);

    arr = [1];
    remove(0);
```

```
    // []

    assert(arr.length == 0);
  }
}
```

Removing the value in an array (By Replacing from End): METHOD 2

NB: Use this method when the order data doesn't matter.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract ArrayReplaceFromEnd {
  uint[] public arr;

  // Deleting an element creates a gap in the array.
  // One trick to keep the array compact is to
  // move the last element into the place to delete
  function remove(uint index) public {
    // Move the last element into the place to delete
    arr[index] = arr[arr.length - 1];

    // Remove the last element
    arr.pop();
  }

  function test() public {
    arr = [1, 2, 3, 4];

    remove(1);
    // [1, 4, 3]
    assert(arr[0] == 1);
    assert(arr[1] == 4);
    assert(arr[2] == 3);
    assert(arr.length = 3)

    remove(2);
    // [1, 4]
    assert(arr[0] == 1);
    assert(arr[1] == 4);
    assert(arr.length == 2);
  }
}
```

Enum

Enum means Enumerable. Enum can be thought of as way of checking the status of a variable, it always return an index number (e.g. 3), while the index is a labeled varible in the index.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Enum {
  // Enum representing shipping status

  enum Status {
    Pending,
    Shipped,
    Accepted,
    Rejected,
    Canceled
  }

  // Default value is the first element listed in
  // definition of the type, in this case "Pending"

  Status public status;

  // Returns uint
  // Pedding - 0
  // Shipped - 1
  // Accepted - 2
  // Rejected - 3
  // Cancelled - 4

  function get() public view returns (Status) {
    return status
  }

  // Update status by passing uint into input
  function set(Status _status) public {
    status = _status;
  }

  // You can update to a specifi enum like this
  function cancel() public {
    status = Status.Cancled;
  }

  // Delete resets the enum to its first value, 0
  function reset() public {
    delete status;
  }
}
```

Importing Enum

Some times smart contract can be huge, and enums may be defined in a seperate solidity file.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

enum Status {
  Pending,
  Shipped,
  Accepted,
  Rejected,
  Canceled
}

import "./EnumDeclaration.sol";

contract Enum {
  Status public status
}
```

Memory keyword vs Calldata keyword

By using memory keyword the argument passed will be mutable.

- Only applicable to reference data type.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract demo {
  uint[] public arr;

  function insertArr(uint[5] memory _arr) public{
    arr = _arr
  }

  function returnArr() public view returns(uint[5] memory){
    return arr;
  }
}
```

Calldata is readonly and immutable

- call data can only be used within function argument
- Cannot be used in returns modifier

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract demo {
  uint[] public arr;
```

```
  function insertArr(uint[5] calldata _arr) public{
    arr = _arr
    // _arr[0] = 12; this will prompt an error
  }
}
```

Struct

Struct is a complex data type. A complex data type usually a composite of other existing data types.

For example:

```
struct Student {
    string name;
    uint roll;
    bool pass;
}
```

- Declaring a struct data type. `struct_type public var_name'`

e.g. Student public s1;

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract demo {
  struct Student {
    string name;
    uint roll;
    bool pass;
  }

  Student public s1;

  function insert(string memory _name, uint _roll, bool _pass) public {
    // s1.name = _name;
    // s1.roll = _roll;
    // s1.pass = _pass;
    // OR
    s1 = Student(_name, _roll, _pass)
    // OR
    // s1 = Student(name: _name, roll: _roll, pass: _pass);
  }

  function getter() public view returns(Student memory) {
    return s1;
  }
```

```
  function getName() public view returns(string name) {
    return s1.name
  }
}
```

Array of Struct

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract demo {
  struct Student {
      string name;
      uint roll;
      bool pass;
    }

  Student[4] public s;

  function insertStudent(uint index, string memory _name, uint _roll, bool _pass)
public {
    s[index] = Student(_name, _roll, _pass);
  }

  function returnStudent(uint index) public view returns(Student memory) {
    return s[index];
  }
}
```

Importing struct

Struct within a solidity (.sol) file can also be imported into another

StructDeclaration.sol

```
struct Todo {
  string text;
  bool completed;
}
```

Todo.sol

```
import "./StructDeclaration.sol";

contract Todos {
  // An array of 'Todo' structs
```

```
  Todo[] public todos;
}
```

Mapping

- is a concept of keys and values
- mapping(keys => value)

For Example: mapping(uint => string) public data;

Roll - Name 1 - Ravi 5 - John 40 - Alice

```
```sol
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract demo {
  mapping(uint => string) public data;

  function insert(uint _roll, string memory _name) public {
    data[_roll] = _name;
  }

  function getter(uint _roll) public view returns(string memory) {
    return data[_roll]
  }

}
```

Example 2:

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

pragma solidity ^0.8.13;

contract Mapping {
  // Mapping from address to uint
  mapping(address => uint) public myMap;

  function get(address _addr) public view returns (uint) {
    // Mapping always returns of a value
    // If the value was never set, it will return the default value
    return myMap[_addr];
  }

  function set(address _addr, uint _i) public {
    // Update the value at this address
    myMap[_addr] = _i
  }
```
```

```
  function remove(address _addr) public {
    // Reset the value to the default value
    delete myMap[_addr];
  }
}
```

Using mapping with struct

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract demo {
  struct Student {
      string name;
      uint roll;
      bool pass;
    }

    mapping(uint => Student) public data;

    function insert(uint index, string memory _name, uint _roll, bool _pass)
public {
      data[index] = Student(_name, _roll, _pass);
    }

    function returnValue(uint index) public view returns(Student memory) {
      return data[index];
    }
}

### Nested Mapping

Think of nested mapping as a 2 dimensional array.

Example:
mapping(address => mapping(address => bool)) private check;

check[address1][address2] = true
check[address2][address3] = true
check[address3][address1] = true

```sol
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract demo {
  mapping(uint => mapping(uint => bool)) public data;

  function insert(uint row, uint column, bool value) public {
    data[row][column] = value;
```
```

```
  }

  function returnValue(uint row, uint column) public view returns(bool) {
    return data[row][column]
  }
}
```

Mapping with Array

For Example: mapping(address => uint[]) private marks;

Address - uint[] 0xabc - [10, 20 , 30, 40, 50] 0xdef - [1, 2, 3]

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract demo {
  mapping(address => uint[]) public marks;

  function insertMarks(address _address, uint[] _marks) {
    marks[_address] = _marks;
  }

  function returnMarks(address _address) public view returns(uint[]) {
    return marks[_address];
  }

}
```

Global Variables

- blockhash(uint blockNumber) returns (byte32)
- block.basefee(uint); // current vb
- block.chainid(uint); // current chain id
- block.coinbase(address payable); // current block miner's address
- block_difficulty(uint); // Current block difficulty
- block.gaslimit(uint); // Current block gaslimit
- block.number(uint); // Current block number
- block.timestamp(uint); // Current block timestamp as seconds since unix epoch
- gasleft() returns (uint256); // remaining gas
- msg.data(bytes calldata); // Complete calldata

- msg.sender(address); // sender of the message (current call)
- msg.sig(bytes4); // First four bytes of the calldata (i.e. function identifier)
- msg.value(uint); // number of wei sent with the message
- tx.gasprice(uint); // gas price of the transaction
- tx.origin(address); // sender of the transaction (full call chain)

Example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract demo {
  // msg.sender

  function CallerAddress() public view returns(address) {
    return msg.sender;
  }

  function returnBlockTimestamp() public view returns(uint) {
    return msg.sender;
  }

}
```

Payable

Receiving Ether/wei using payable modifer & Checking contract balance

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract demo {
  // transfaer ether to my contract

  function sendEtherToContract() public payable {
    // you may have some statments here if you want to
    // but having the payable modifier is suffient enought to
    // make the contract recieve ether.
    // Set the value of the ether you want to transfer you wnat to transfer then
CALL THE FUNCTION
  }

  // see the balance of your contract
  function balanceOfContract() public view returns(uint) {
    return address(this).balance;
```

```
  }
}
```

Using smart contract to send ether from your account/address to another address

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract demo {
  function sendETH(address payable receiver) public payable{
    receiver.transfer(msg.value);
  }


}
```

Inheritance

Solidity supports multiple inheritance. Contracts can inherit other contract by using the `is` keyword.

A function that is going to be overridden by a child contract must be declared as `virtual`.

A Function that is going to override a parent function must be use the keyword `override`.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.13;

/* Graph of inheritance
          A
        /   \
       B     C
      / \   /
   v F   D,E
*/

contract A {
  function foo() public pure virtual returns (string memory) {
    return "A"
  }

  // Contract inherit other contracts by using the keyword 'is'
  contract B is A {
    // Override A.foo()
    function foo() public pure virtual overide returns (string memory) {
      return "B";
    }

    contract C is A {
      function foo() public pure virtual overide returns(string memory){
```

```
        return "C";
      }
    }


    // Contracts can inherit form multiple parent contracts.
    // When a function is called that is defined multiple time in
    // different contracts, parent contracts are searched from (left to right, and
in depth-first manner)
    // the right most inherited contract will be excuted

    contract D is B, C {
      // D.foo() return "C"
      // since C is the right most parent contract with the function foo() public
override(B, C) returns (string memory) {
        return super.foo();
      }
    }

    contract E is C, B {
      // E.foo() return "B"
      // Since B is the right most parent contract with function foo()
      function foo() public pure override(C, B) returns (string memory) {
        return super.foo();
      }
    }

    // Inheritance must be ordered from "most base-like" to "most derived"
    // Swapping the order of A and B will throw a compilation error.
    contract F is A, B {
      function foo() public pure override(A, B) returns(string memory) {
        return super.foo()
      }
    }

  }
}
```

Shadowing Inherited Contracts

It allows one to inherit state variables. Unlike functions, state variables can no be overidden by redeclaring in the child contract.

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

pragma solidity ^0.8.13;

contract A {
  string public name = "Contract A";

  function getName() public view returns(string memory) {
```

```
    return name;
  }
}


// Shadowing is disallowed in solidity 0.6
// This will not compile
// contract B is A {
//    string public name = "Contract B";
//}

contract C is A {
  // This is the correct way to overide inherited state variables
  constructor() {
    name = "Contract C";
  }

  // C.getName returns "Contract C"
}
```

Super

Super keyword allow you to inherit parent contract(s).

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.13;

/* Inheritance tree

      A
    /   \
   B     C
    \   /
      D
*/

contract A {
  // This is called an event. You can emit events from you function
  // and they logged into the transaction log.
  // In our case, this will be useful for tracing function calls.
  event Log(string message);

  function foo() public virtual {
    emit Log("A.foo called");
  }

  function bar() public virtual {
    emit Log("A.bar called");
  }
}
```

```
contract B is A {
  function foo() public virtual override {
    emit Log("B.foo called");
    A.foo();
  }

  function bar() public virtual override {
    emit Log("B.bar called");
    super.bar(); // using super here is similar to writing A.bar()
  }
}

contract C is A {
  function foo() public virtual override {
    emit Log("C.foo called");
    A.foo();
  }

  function bar() public virtual override {
    emit Log("C.bar called");
    super.bar(); // using super here is similar to writing A.bar()
  }
}

contract D is B,C {
  // Try:
  // - Call D.foo and check the transaction Logs.
  //   Although D inherits A, B and C, it only called C and then A.
  // - Call D.bar and check the transaction Logs
  //   D called C, then B, and finally A.
  // Although super was called twice (by B and C) it only called A

  functionl foo() public override(B,C) {
    super.foo();
  }

  function bar() public overide(B,C) {
    super.bar()
  }
}
```

In the function above D inherits for multiple contracts. Calling super in contract D allows the function to be called from left to right (i.e. the reverse order of inheritance)

> One thing to note here from calling D.bar() is that Calling super.bar() calls the parent contract C.bar() (the right most)

Visibility

Functions and state variables havet to declare whether they are accessible by other contracts.

Functions cna be declared as:

- public - any contract and account can call
- only inside the contract that defines the function
- internal - only inside contract that inherits an internal function
- external - only other contracts and accounts can call

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.13;

contract Base {
  // Private function can only be called
  // - Inside this contract
  // Contracts that inherit this contract cannot this contract cannot call this
function
  function privateFunc() private pure returns (string memory) {
    return "Private functon called";
  }

  function testPrivateFunc() public pure returns (string memory) {
    return privateFunc();
  }

  // Internal function can be called
  // - inside this contract
  // - inside contracts that inherit this contract
  function internalFunc() internal pure returns(string  memory) {
    return "internal function called";
  }

  function testInternalFunc() public pure virtual returns (string memory) {
    return internalFunc();
  }

  // Puvlic functions can be called
  // - inside this contract
  // - inside contract that inherit this contract
  // - by other contracts and accounts
  function publicFunc() public pure returns (string memory) {
    return "public function called";
  }

  // External function can only be called
  // - by other contracts and accounts
  function externalFunc() external pure returns (string memory) {
    return "external function called";
  }

  // This function will not compile since we're trying to call
  // an external function here.
  // function testExternalFunc() public pure returns (string memory) {
  //    return externalFunc();
```

```
  // }

  // State variables
  string private privateVar = "my private variable";
  string internal interalVar = "my interanal variable";
  string public publicVar = "my public variable";
  // State variable cannot be external so it won't compile.
  // string external externalVar = "my external variable";
}

contract Child is Base {
  // Inherited contracts do not have access to private functions
  // and state variables
  // function testPrivateFunct() public pure returns (string memory) {
  //   return privateFunction();
  // }

  // Internal function call be called inside child contracts
  function testInternalFunction() public pure override returns (string memory) {
    return internalFunc();
  }
}
```

Interface

You can interact with other contracts by declaring an Interface.

Interface

- cannot have any functions implemented
- can inherit from other interfaces
- all declared functions must be external
- cannot declare a constructor
- cannot declare state variables

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.13;

contract Counter {
  uint public count;

  function increment() external {
    count += 1;
  }
}

interface ICounter {
  function count() external view returns (uint); // this function allows another
contract to get the value of count in the contract abover

  function increment() external;
```

```
}

contract MyContract {
  function incrementCounter(address _counter) external {
    ICounter(_counter).increment();
  }

  function getCount(address _counter) external view returns (uint) {
    return ICounter(_counter).counter();
  }
}

// Uniswap example
interface UniswapV2Factory {
  function getPair(address tokenA, address tokenB) external view returns(address
pair) {

  }
}

interface UniswapV2Pair {
  function getReserves() external view returns(uint112 reserve0, uint112 reserve1,
uint32 blockTimestampLast);
}
```

contract UniswapExample { address private factory = 0x5C69bEe701ef814a2B6a3EDD4B1652cB9cc5aA6f; address private dai = 0x6B1....d0F; address private weth = 0xC02...Cc2;

function getTokenReserves() external view returns (uint, uint) { address pair = UinswapV2Factory(factory).getPair(dai, weth); (uint reserve0, uint reserve1,) = UniswapV2Pair(pair).getReserves(); return (reserve0, reserve1); } }

Payable

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

pragma solidity ^0.8.13;

contract Payable {
  // Payable address can receive Ether
  address public payable owner;

  // Payable constructor can recieve Ether
  constructor() payable {
    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 {}
```

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

  // Function to withdraw all Ether from this contract
  function withdraw() public {
    // get the amount of Ether stored in this contract
    uint amount = address(this).balance;

    // Send all Ether to owner
    // Owner can receive Ether since the address of owner is payable
    (bool success, ) = owner.call{value: amount}("");
    require(success, "Failed to send Ether");
  }

  // Function to transfer Ether from this contract to address from input
  function transfer(address payable _to, uint _amount) public {
    // Note that "to" is declared as payable
    (bool success,) = _to.call{value: _amount}("");
    require(succes, "Failed to send Ether");
  }
}
```

Sending Ether

There are different ways to send ether.

- transfer()
- send()
- call()

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.13;

contract ReceiveEther {
  /*
   Which function is called, fallback() or receive()?

                Send Ether
                    |
              msg.data is empty?
                  /   \
                yes    no
                /       \
     recieve() exists?   fallback()
             /    \
      recieve()  fallback()

    - if msg.data is empty it will call receive()
    - if msg.data is not empty it will call fallback()
```

```
    */

    // Function to receive Ether. msg.data must be empty
    receive() external payable {}

    // Fallback function is called when msg.data is not empty
    fallback() external payable {}

    function getBalance() public view returns (uint) {
      return address(this).balance;
    }
  }

  contract SendEther {
    function sendViaTransfer(address payable _to) public payable {
      // This function is no longer recommended to send Ether
      _to.transfer(msg.value);
    }

    function sendViaSend(address payable _to) public payable {
      // Send returns a boolean value indicating success or failure
      // This function is not recommended for sending Ether
      bool sent = _to.send(msg.value);
      require(sent, "Failed to send Ether");
    }

    function sendViaCall(address payable _to) public payable {
      // Call returns a boolean value indicating success or failure
      // This is the current recommended method to use
      (bool sent, bytes memory data) = _to.call{value: msg.value}("");
      require(sent, "Failed to send Ether");
    }
  }
```

Fallback

When you use fallback in a contract, and someone send ether to the contract using ether the transfer or send method, the minimum gas they have to pay 2300.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.13;

contract Fallback {
  event Log(string func, uint gas);

  // Fallback function must be declared as external
  fallback() external payable {
    // send / transfer (forwards 2300 gas to this fallback function)
    // call (forwads all the the gas)
```

```
    emit Log("fallback", gasleft());
  }

  // Receive is a variant of fallback that is triggered when msg.data is empty
  recieve() external payable {
    emit Log("receive", gasleft());
  }

  // Helper function to check the balance of this contract
  function getBalance() public view return(uint){
    return account(this).balance;
  }
}

contract SendToFallback {
  function transferToFallback(address payable _to) public payable {
    _to.transfer(msg.value);
  }

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

// Use gasleft() to know the amount of gas used in a transaction

1. `receive()` (`external` function) is called when Ether is sent directly to the contract's address **without any data or function call**.
    - It must have an `external` visibility and no arguments
    - If you don't define a recieve() function, but your contract receives Ether without a function call, the `fallback()` function is called.
2. `fallback()` (`external` function) is called when a function call doesn't match any of the other functions defined in the contract without any data or function call (if receive() is not defined).
    - It must have an `external` visibility and no argurments.
    - If you don't define a `fallback()` function, and Ether is sent to the contract without a function call, the transaction will be reverted.

> **NB**: Declaring receive() and fallback() function is not compulsory, but it's recommended to have them if you want your contract to be able to recieve Ether. If you don't define these functions, and someone tries to send Ether to your contract without a function call, the transaction will be reverted.

1. `Send()` returns a boolean indicating whether the transfer was successful or not.
    - If the transfer fails, it doesn't revert the transaction, but it propagates an error that can be caught.

- It doesn't trigger the `receive()` or `fallback()` function in the recieving contract.

2. `transfer()` - It reverts the transaction if the transfer fails, consuming all the remaining gas.

- It does not trigger the `recieve()` or `fallback()` functions in the receiving contract.

3. `call()` - It can be used to send Ether and trigger the `recieve()` or `fallback()` functions in the receiving contract, depending on the data provided.
    - If you send Ether along with data (a function call), it will trigger the specified function in the receiving contract. If the function is not found, if will trigger the trigger the fallback() function.

SAMPLE CODE: WHEN ETHER IS SENT, WITH DATA (FUNCTION CALL)

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

pragma solidity ^0.8.0;

contract ReceivingContract {
  event Received(address sender, uint value, byte data);
  event FunctionCalled(address sender, uint value, bytes data);

  receive() external payable {
    emit Received(msg.sender, msg.value, "");
  }

  fallback() external payable {
    emit Received(msg.sender, msg.value, msg.value);
  }

  function deposit() external payable {
    emit FunctionCalled(msg.sender, msg.value, msg.data);
  }
}

contract SendingContract {
  ReceivingContract public receivingContract;

  constructor(address _receivingContractAddress) {
    receivingContract = ReceivingContract(_receivingContractAddress);
  }

  function sendEtherWithData() external payable {
    bytes memory data = abi.encodeWithSignature("deposit()");
    (bool success, ) = address(receivingContract).call{value: msg.value}(data);
    require(success, "Transaction failed");
  }

  function sendEtherWithoutData() external payable {
    (bool success, ) = address(receiveContract).call{value: msg.value}("");
    require(success, "Transaction failed");
  }
}
```

`sendEtherWithData()` sends Ether to the `ReceivingContract` along with data (a function call to `deposit()`). It encodes the function signature using `abi.encodeWithSignature` and sends it along with the Ether using the `call()` function.

Call

```
contract Receiver {
  event Received(address caller, uint amount, string message);

  fallback() external payable {
    emit Received(msg.sender, msg.value, "Fallback was called");
  }

  function foo(string memory _message, uint _x) public payable returns (uint) {
    emit Received(msg.sender, msg.value, _message);
    return _x + 1;
  }
}

contract Caller {
  event Response(bool success, bytes data);

  // Lets's imagine that contract Caller does not have the source code for ___
  // contract Receiver, but we do know the address of contract Receiver and ______
  function testCallFoo(address payable _addr) public payable {
    // You can send ether and specify a custom gas amount
    (bool success, bytes memory data) = _addr.call{value: msg.value, gas: }
(abi.encodeWithSignature("foo(string, uint256)", "call foo", 123))

    emit Response(succes, data);
  }

  // Calling a function that does not exist triggers the fallback function
  function testCallDoesNotExist(address _addr) public {
    (bool success, bytes memory data) =
_addr.call(abi.encodeWithSignature("doesNotExist()")

    emit Response(success, data);
  }
}
```

Delegatecall

When a contract A, executes a delegated call to contract B, in that scenerio contract B will also execute. In contract A, the storage of msg.sender and msg.value, will be accessible.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.13;
```

```
// NOTE: Deploy this contract first
contract B {
  // NOTE: Storage Layout must be the same as contract A
  uint public num;
  address public sender;
  uint public value;

  setVars(uint _num) public payable {
    num = _num;
    sender = msg.sender;
    value = msg.value;
  }
}

contract A {
  uint public num;
  address public sender;
  uint public value;

  function setVars(address _contract, uint _num) public payable {
    // A's storage is set, B is not modified
    (bool success, bytes memory data) =
_contract.delegatecall(abi.encodeWithSignatures("setVars(uint256)", _num))
  }
}
```

Calling Other Contracts

Contracts can be called in two ways.

1. By name
2. By low level call

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

contract callee {
  uint public x;
  uint public value;

  function setX(uint _x) public returns (uint) {
    x = _x;
    return x;
  }

  function setXandSendEther(uint _x) public payable returns (uint, uint) {
    x = _x;
    value = msg.value;

    return (x, value);
  }
}
```

```
contract Caller {
  function setX(Callee _callee, uint _x) public {
    uint x = _called.setX(_x);
  }

  function setXFromAddress(address _addr, uint _x) public {
    Callee calle = Callee(_addr);
    callee.setX(_x);
  }

  function setXandSendEther(Callee _callee, uint _x) public payable {
    (uint x, uint value) = _callee.setXandSendEther{value: msg.value}(_x);
  }
}
```

// Callee type in Caller contract is the address of the deployed Callee contract.

Contracts that Creates other Contracts

This is achievable using the new keyword.

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

contract Car {
  address public owner;
  string public model;
  address public carAddr;

  contructor (address _owner, string memory _model) payable {
    owner = _owner;
    model = _model;
    carAddr = address(this)
  }
}

contract CarFactory {
  Car[] public cars;

  function create(address _owner, string memory _model) public {
    Car car = new Car(_owner, _model);
    cars.push(car);
  }

  function createAndSendEther(address _owner, string memory _model) public payable
{
    Car car = (new Car){value: msg.value}(_owner, _model);
    cars.push(car);
  }

  function create2 (address _owner, string memory _model, bytes32 _salt) public {
```

```
    Car car = (new Car){salt: _salt}(_owner, _model_);
    cars.push(car);
  }

  function create2AndSendEther(address _owner, string memory _model, bytes32
_salt) public payable{
    Car car = (new Car){view: msg.value, salt: _salt}(_owner, _model);
    cars.push(car);
  }

  function getcar(uint _inde) public view returns (address owner, string memory
model, address carAddr, uint balance) {
    Car car = cars[_index];
    return (car.owner(), car.model(), car.carAddr(), address(car).balance);
  }
}
```

When you use a contract as an input in another contract, the expected input in application is the contracts address.

Try and Catch

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

pragma solidity ^0.8.13;

// External contract used for try / catch example
contract foo {
  address public owner;

  constructor(address _owner){
    require(_owner != address(0), "Invalid address");
    asser(_owner != 0x0000000000000000000000001);
    owner = _owner;
  }

  function myFunc(uint x) public pure returns (string memory) {
    require(x != 0, "require failed");
    return "my function was called";
  }

}

contract Bar {
  event Log(string message);
  event LogBytes(bytes data);

  Foo public foo;

  constructor() {
    // This Foo contract is used for example of try catch with external call
```

```
    foo = new Foo.(msg.sender);
  }

  // Example of try / catch with external call
  // tryCatchExternalCall(0) => Log("external call failed")
  function tryCatchExternalCall(uint _i) public {
    try foo.myFunc(_i) returns(string memory result) {
      emit Log(result);
    } catch {
      emit Log("external call failed");
    }
  }

  // Example of try / catch with contract creation
  // tryCatchNewContract(0x0000000000000000000000000000) => Log("Invalid
Address")
  // tryCatchNewContract(0x0000000000000000000000000001) =>
  // tryCatchNewContract(0x0000000000000000000000000002) =>
  function tryCatchNewContract(addres _owner) public {
    try new Foo(_owner) returns (Foo foo) {
      // you can use variable foo here
      emit Log("Foo created");
    } catch Error(string memory reason) {
      // catch failing revert() and require()
      emit Log(reason);
    } catch (bytes memory reason) {
      // catch failing assert()
      emit LogBytes(reason);
    }
  }
}
```

Library

- How to create a library
- How to use a library

> A library is created using the keyword library.

```
// SPDX-Library-Identifier: MIT
pragma solidity ^0.8.13;

library Math {
  function sqrt(uint y) internal pure returns (uint z) {
    if (y > 3) {
      z = y;
      uint x = y / (2 + 1);

      while(x < z){
        z = x;
        x = (y / x + x) / 2;
```

```
      }
    } else if (y != 0) {
      z = 1;
    }
    // else z = 0 (default value)
  }
}

contract TestMath {
  function testSquareRoot(uint x) public pure returns (uint) {
    return Math.sqrt(x);
  }
}

// Array function to delete element at index and re-organize the array
// so that there are no gaps between the elements.

library Array {
  function remove(uintp[] storage arr, uint index) public {
    // Move the last element into the place t delete
    require(arr.length > 0, "Can't remove from empty array");
    arr[index] = arr[arr.length - 1];
    arr.pop();
  }
}

contract TestArray {
  using Array for uint[];

  uint[] public arr;

  function testArrayRemove() public {
    for(uint 1 = 0; i < 3; i++) {
      arr.push(i);
    }

    arr.remove(1);

    assert(arr.length == 2);
    assert(arr[0] == 0);
    assert(arr[1] == 2);

  }
}
```

ABI Encode

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

pragma solidity ^0.8.13;
```

```
interface IERC20 {
  function transfer(address, uint) external;
}
```