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.

view vs pure vs when neither is used

- If you are changing the value of a 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.

```
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.

```
// 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.

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

int

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

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

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

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

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;
        }
    }
```

Bool Data Type

- true
- false

e.g. bool public value = true.

- By default value is false if not initialized.

```
// 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;
    }
}
```

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
    }
}
```

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")

  }
}
```

Modifier

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;
  }
}
```

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**.

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