

Introduction to Blockchain Development with Ethereum

Coding-Bootcamps.com

Session 6- Structure of Smart Contracts

The Solidity Contract:

Pragma

The pragma key word is used to invoke the certain compilers.

It is similar to the version being used.

Solidity contract writers will be different variations of pragma entries.

The pragma directive is entered as shown:

Using a “greater than or equal to compiler” indicator: `pragma solidity >=0.4.22 <0.6.0;`

This indicates that the contract needs at least compiler version 4.22 and is backward compatible to compiler version 0.6.0.

The contract with the entry below, with the caret, ^, will not compile with a version earlier than 0.5.2

`pragma solidity ^0.5.2;`

Other variations: `pragma solidity >=0.5.0 <0.7.0;`

Comments

Code comments greatly increase the maintainability and understandability of your contract

Single line comments are indicated by “//”

Multiline comments are entered as : “/* */”

```

1  pragma solidity >=0.5.0 <0.7.0;
2
3  contract TotalUpContract {
4
5      // This contract totals passed in numbers
6
7      /* Declare variable num1
8      Declare variable num2
9      Declare sumToal
10     initialize all to 0 */
11
12     int public num1 = 0;
13     int public num2 = 0;
14     int public sumTotal = 0;
15
16     function setNumbers (int _quant1, int _quant2) public {
17         num1 = _quant1;
18         num2 = _quant2;
19     }
20
21
22     function sumIt() public returns(int _sum1) {
23
24         sumTotal = num1 + num2;
25         _sum1 = sumTotal;
26         return _sum1;
27     }
28
29 }
30

```


Booleans

Booleans are either “true” or “false”

Booleans have the list of operators

- ! negation
- \$\$ Logical conjunction, "and"
- || Logical disjunction, "or"
- == equality
- != inequality

In the function below, sumTotal will not be returned.





```
int public num1 = 0;
int public num2 = 0;
int public sumTotal = 0;
bool public bCond1;

function setNumbers (int _quant1, int _quant2) public {
    num1 = _quant1;
    num2 = _quant2;
}

function sumIt() public returns(int _sum1) {
    sumTotal = num1 + num2;
    // _sum1 = sumTotal;
    bCond1 = false;

    if (bCond1 == true){
        return sumTotal;
    }
}
```



Integers

Solidity supports signed and unsigned integers

- int (signed)
- uint (unsigned)

Signed integers can be negative values.

Integers can be different sizes from 8 to 256 bits

For example:

- int8
- int256
- uint

- uint256

The integer operators exhibit the same functionality and behavior as Java, C++ and JavaScript.

Operators:

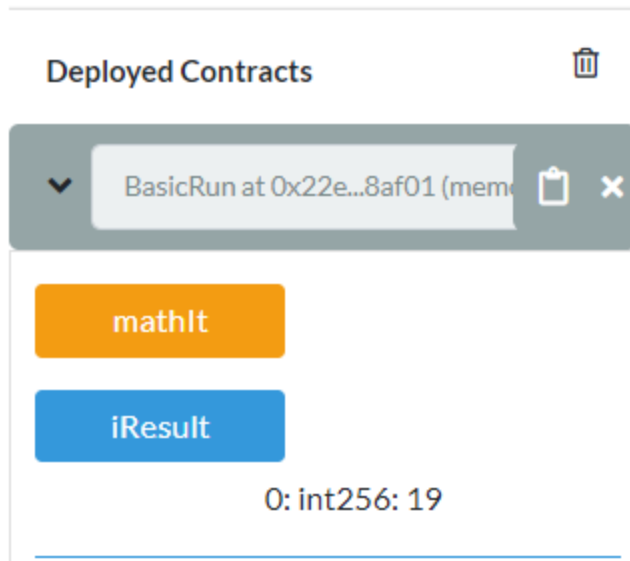
Comparisons

- <=, less than or equal to
- <, less than
- ==, equality
- !=, inequality
- >=, greater than or equal to
- >, greater than

Arithmetic Operations

- +, addition
- -, subtraction
- *, multiplication
- /, division
- %, modulo
- **, exponential

```
1  pragma solidity ^0.5.1;
2
3  contract BasicRun {
4
5
6      int  numA = 20;
7      int  numN = -1;
8      uint numB = 16;
9      uint numC = 100;
10     int public iResult;
11
12
13     function mathIt() public returns ( int _iResult){
14
15         iResult = numA + numN;
16         _iResult = iResult;
17
18     }
```



Strings

Strings are special type of arrays.

Strings are must be inside quotes or double quotes:

"hello ", 'world '

Strings cannot be directly compared for equivalence in Solidity

However, strings' hash values can be compared.

String hashes can be compared by adding a function that returns a Boolean of the status of the passed in strings.

Next call the function with the strings to be compared passed in as parameters.

```

function sumIt() public returns(int _sum1, string memory _str1) {
    sumTotal = num1 + num2;

    bCond1 = true;
    _str1 = instruct;

    bEqual = compareStringsbyBytes(_str1, "test");

    if (bCond1 == true && bEqual == true){
        return (sumTotal, instruct);
    }
}

function compareStringsbyBytes(string memory s1, string memory s2) internal pure returns(bool){
    return keccak256(abi.encodePacked(s1)) == keccak256(abi.encodePacked(s2));
}

```

Strings Literals using escape characters

\<newline>, escapes an actual newline

\\, backslash

\', single quote

\", double quote

\b, backspace

\f, form feed

\n, newline

\r, (carriage return)

\t, (tab)

\v, (vertical tab)

```

1 pragma solidity ^0.5.1;
2
3 contract BasicRun {
4
5     string public liters = "\n'\\solidity is fun !\\'\t \"\treally fun\"";
6
7
8 }

```

Address

The address data type is meant for Ethereum addresses.

There are 2 versions of the address data type: address and address payable.

address: manages a 20 byte data type for an Ethereum address

address payable: same as address, but also including methods: transfer and send.

address payable supports sending Ether (Ethereum's native currency) to another address.

address does not send Ether.

Operators:

Comparisons

- <=, less than or equal to
- <, less than
- ==, equality
- !=, inequality
- >=, greater than or equal to
- >, greater than

```
1  pragma solidity 0.5.1;
2
3  contract GetAddress {
4
5      address payable public wallet1;
6
7      function getMyAddress() public returns (address _addr1){
8
9          wallet1 = msg.sender;
10         return wallet1;
11     }
12 }
13
14
15 }
```

Currently select address is: 0xCA35b7d915458EF540aDe6068dFe2F44E8fa733c

Environment

JavaScript VM

3
4
5
6
7
8

Account

0xca3...a733c (96.99999999999999 ether)

0xca3...a733c (96.99999999999999 ether)

0x147...c160c (99.99999999999999 ether)

0x4b0...4d2db (102.99999999999999 ether)

0x583...40225 (100 ether)

0xdd8...92148 (100 ether)

Gas limit

Value

GetAddress

The Contract returns the current address as wallet1

Deployed Contracts

GetAddress at 0xf1f...a14f4 (memory)

getMyAddress

wallet1

0: address: 0xCA35b7d915458EF540aDe6068dFe2F44E8fa733c

Enums

Enums are an example of a user defined, or custom data type.

Enums are given a specific name and contain a list of data.

The data is 0 indexed.

When an enum is declared, it is declared with its name and it is typed as the enum.

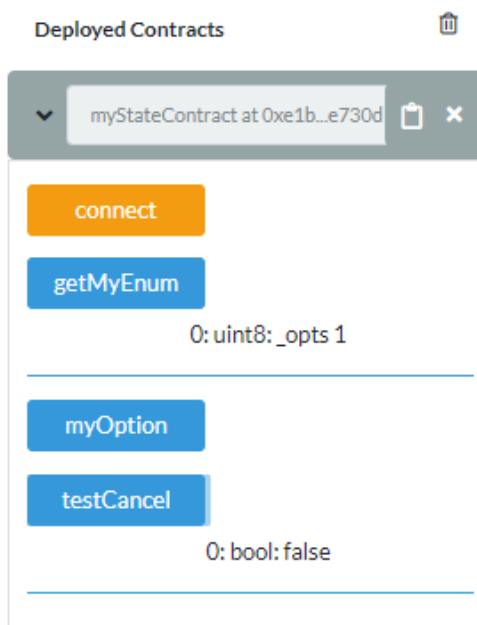
The contract below sets the enum to connect, which is index 1,

The enum is initialized in the constructor as cancel, index 2, but the testCancel returns false

```

1  pragma solidity 0.5.1;
2
3  contract myStateContract {
4      enum UserOptions {listen, connect, cancel}
5      UserOptions public myOption;
6
7      constructor() public {
8          myOption = UserOptions.cancel;
9      }
10
11     function connect() public {
12         myOption = UserOptions.connect;
13     }
14
15     function testCancel() public view returns(bool){
16         return myOption == UserOptions.cancel;
17     }
18
19     function getMyEnum() public view returns (UserOptions _opts){
20
21         return myOption;
22     }
23 }
24

```



Structs

Another custom data type is Structs

Structs contain a list of attribute datatypes that describe it.

```

struct Funder {
    address addr;
    uint amount;
}

```

Structs model a model a real-world entity that can be managed with a contract.

Structs are containers of multiple data members of its type.

Structs are used by other data structures such as Arrays and Mappings.


```
1 pragma solidity ^0.5.2;
2
3 contract myStructContract {
4
5     //array to keep track of several people
6     //Person[] public people;
7
8     uint256 public studentCounter = 0;
9     //mapping, key value pair unit is key person is value
10    mapping(uint => Student) public students;
11
12
13
14
15    struct Student{
16        uint _id1;
17        string _fname;
18        string _lname;
19    }
20
21
22
23    function addPerson(string memory _fname, string memory _lname) public {
24        studentCounter++;
25        //people.push(Person(_fname, _lname));
26        //peopleCounter += 1;
27        students[studentCounter] = Student(studentCounter, _fname, _lname);
28    }
29
30
31
32 }
```

Global Variables

Ether Units

```
assert(1 wei == 1);  
assert(1 szabo == 1e12);  
assert(1 finney == 1e15);  
assert(1 ether == 1e18);
```

Time Units

- 1 == 1 seconds
- 1 minutes == 60 seconds
- 1 hours == 60 minutes
- 1 days == 24 hours
- 1 weeks == 7 days

Block and Transaction Properties

- `blockhash(uint blockNumber)` returns (bytes32): hash of the given block - only works for 256 most recent, excluding current, blocks
- `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 payable): 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
- `now` (uint): current block timestamp (alias for `block.timestamp`)
- `tx.gasprice` (uint): gas price of the transaction
- `tx.origin` (address payable): sender of the transaction (full call chain)

Arrays

Arrays are data structures where one variable of a certain data type, contains multiple values, in different sections of memory.

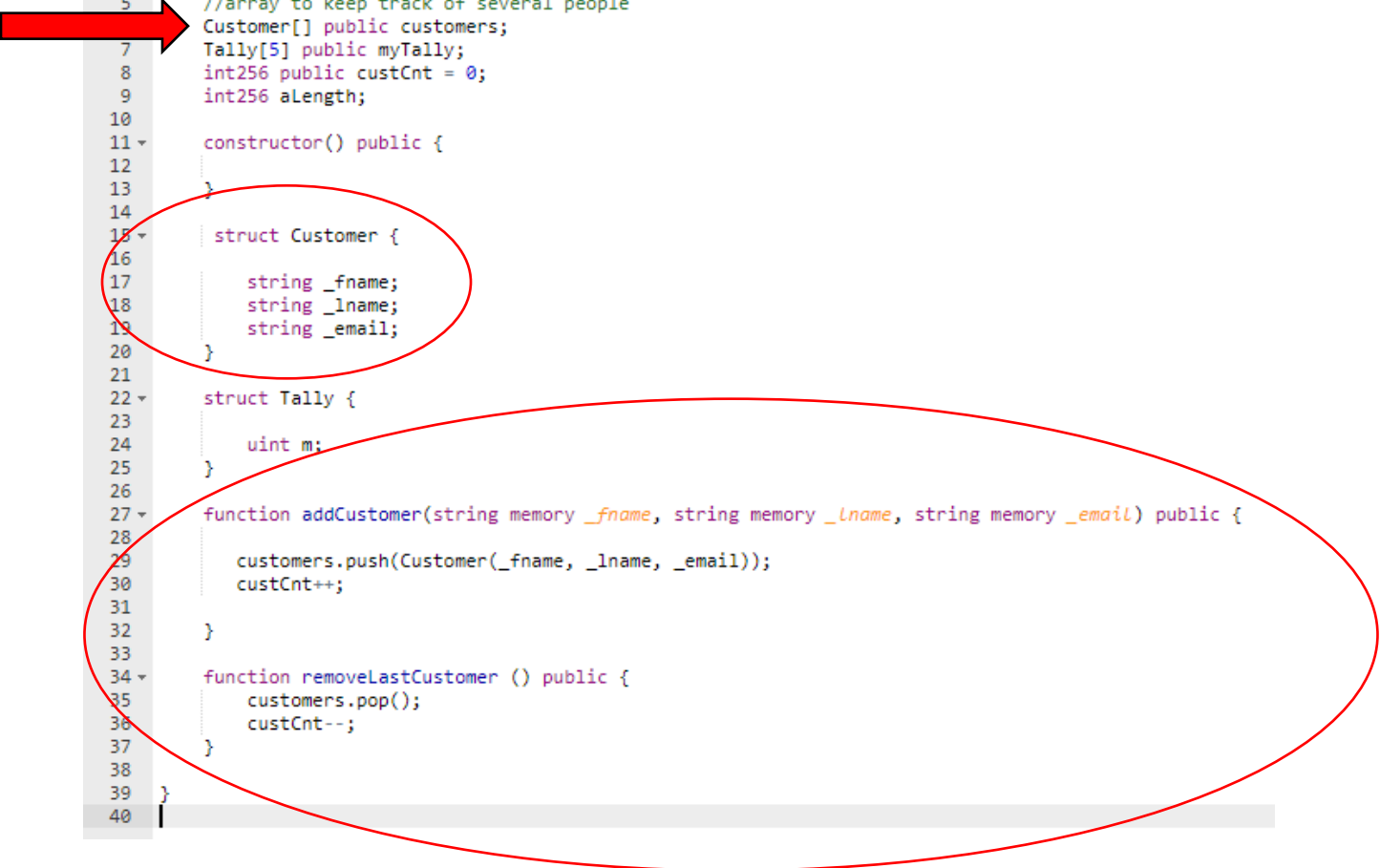
Arrays can be fixed size, or dynamic.

Fixed size arrays can be declared at compile time.

Values can also be added or removed from the Array using member methods push and pop.

The array length is used to add length to the array.

Arrays are often built from structs.



```
1  pragma solidity 0.5.1;
2
3  contract ArrayContract {
4
5      //array to keep track of several people
6      Customer[] public customers;
7      Tally[5] public myTally;
8      int256 public custCnt = 0;
9      int256 aLength;
10
11     constructor() public {
12
13     }
14
15     struct Customer {
16         string _fname;
17         string _lname;
18         string _email;
19     }
20
21
22     struct Tally {
23
24         uint m;
25     }
26
27     function addCustomer(string memory _fname, string memory _lname, string memory _email) public {
28
29         customers.push(Customer(_fname, _lname, _email));
30         custCnt++;
31     }
32
33
34     function removeLastCustomer () public {
35         customers.pop();
36         custCnt--;
37     }
38
39 }
40
```

Deployed Contracts

MyMappingContract at 0xfc7...d

addCustomer

_fname:

"Art"

_lname:

"Mitchel"

_email:

"art@yahoo.com"

transact

custCounter

0: uint256: 4

customers

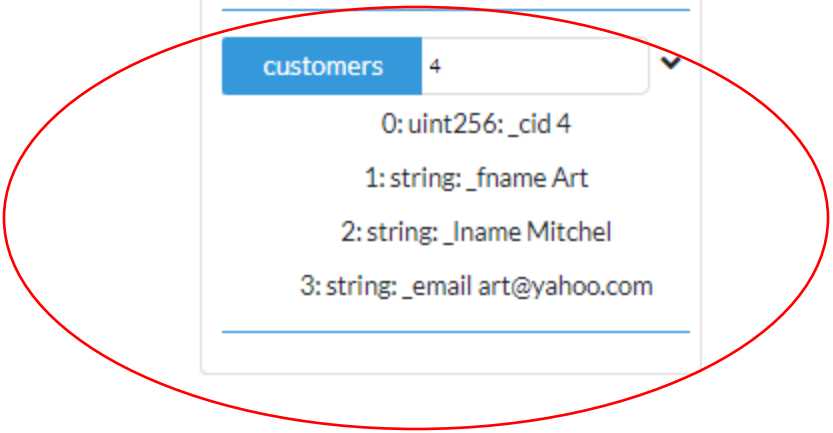
4

0: uint256: _cid 4

1: string: _fname Art

2: string: _lname Mitchel

3: string: _email art@yahoo.com



Control Structures

if, else, while, do, for, break, continue, return with the same behavior, rules and functionality as Java, C++, and JavaScript.

In Solidity, Parentheses cannot be omitted for conditionals, but curly braces can be omitted around single-statement bodies.

If ($x < 0$)

 sum = a + b;

if (expression 1) {

 Statement(s) to be executed if expression 1 is true

} else if (expression 2) {

 Statement(s) to be executed if expression 2 is true

} else if (expression 3) {

 Statement(s) to be executed if expression 3 is true

} else {

 Statement(s) to be executed if no expression is true

}

- return statement ends the function and returns control to where the function was called
- break just breaks the loop & return gets control back to the caller method
- continue statement ends program execution of the current iteration of a loop statement but does not stop execution of the loop statement.

If ... else if ... else example:

```

1  pragma solidity 0.5.1;
2
3  contract basicContract1{
4
5      uint iArt;
6      uint myIntVal;
7      uint myInput;
8      event tstVal(string value1);
9
10
11     function setVal(uint _iVal) public {
12
13         string memory str;
14
15
16         myIntVal = _iVal;
17         str=testVal(myIntVal);
18         emit tstVal(str);
19     }
20
21
22     function testVal(uint _iTst) internal pure returns (string memory){
23
24         uint inum;
25         string memory myString;
26
27         inum = _iTst;
28         if(inum == 0){
29
30             myString = "The number is 0";
31
32         }
33         if(inum > 0 && inum < 10){
34             myString = "The number is small";
35         }
36         else if(inum >= 10 && inum < 100) {
37
38             myString = "The number is big";
39
40         }
41         else if(inum >= 100 && inum < 1000) {
42
43             myString = "The number is way big";
44
45         }
46         else if(inum >= 1000 && inum < 999999) {
47
48             myString = "The number is huge";
49
50         }
51         else{
52             myString = "The number is way huge and out of sight";
53         }
54         return myString;
55     }
56 }
57
58 }
```

Mapping Types

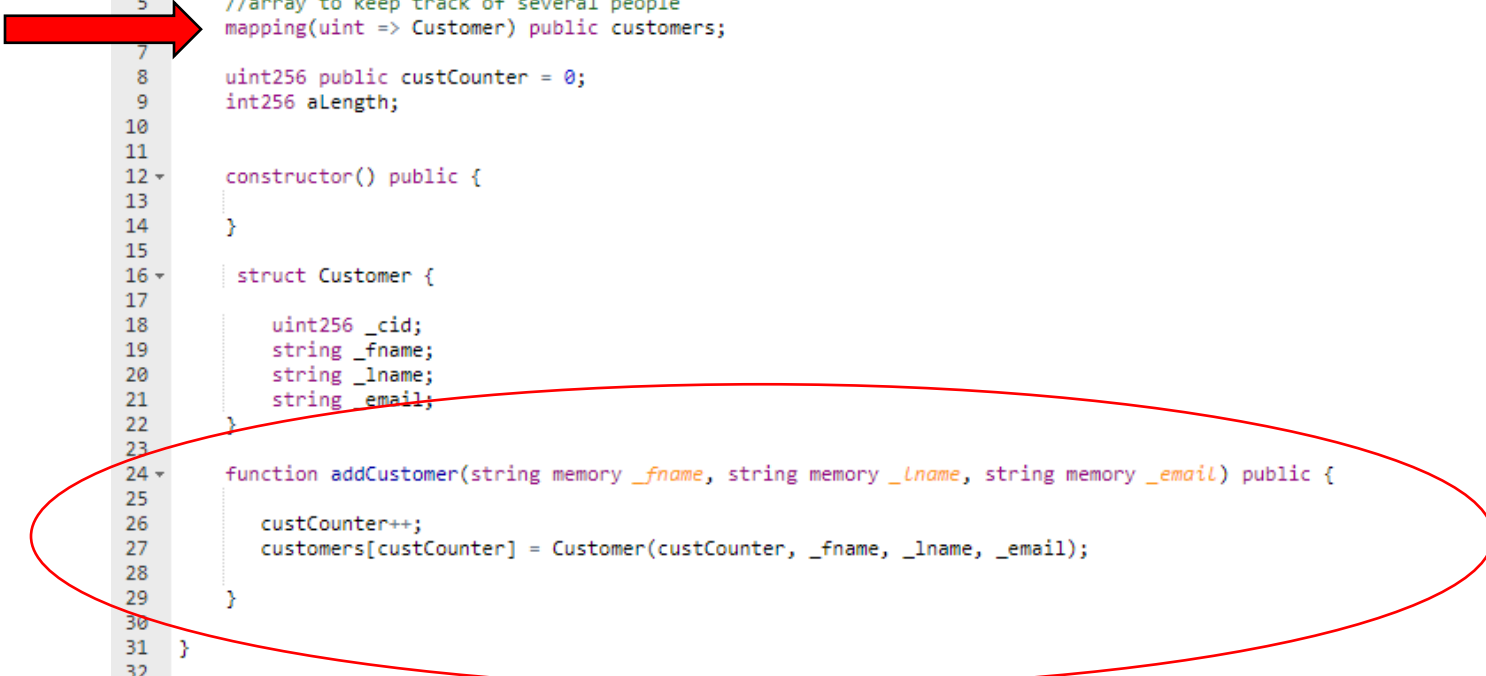
Mapping is similar to hash tables, in that data can be accessed with a key


Mappings are data structures stored with key, value pairs.



Mappings syntax: mapping(key => value)


customers(1,"Mark", "Smith", mark@yahoo.com)

```
1 pragma solidity 0.5.1;
2
3 contract MyMappingContract {
4
5     //array to keep track of several people
6     mapping(uint => Customer) public customers;
7
8     uint256 public custCounter = 0;
9     int256 aLength;
10
11
12     constructor() public {
13
14     }
15
16     struct Customer {
17
18         uint256 _cid;
19         string _fname;
20         string _lname;
21         string _email;
22     }
23
24     function addCustomer(string memory _fname, string memory _lname, string memory _email) public {
25
26         custCounter++;
27         customers[custCounter] = Customer(custCounter, _fname, _lname, _email);
28     }
29
30 }
31
32
```



Deployed Contracts 



▼ MyMappingContract at 0x52b...  


addCustomer 

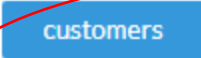

_fname:

_lname:

_email:


0: uint256: 3

 3 

0: uint256: _cid 3
1: string: _fname Marty
2: string: _lname Jones
3: string: _email marty@yahoo.com

Conversion between native Types

There are two types of conversion: Implicit and Explicit

Implicit conversions

When an operator is used on different data types the compiler will make type conversion based on no data loss.

If data will be lost, a compiler error is returned.

The example below shows the conversion of different integer types.

```

6  contract dataCon {
7
8      uint i1;
9      uint8 i2;
10     uint256 sum;
11     string s1= "7";
12
13
14
15  function addIT() public {
16
17      sum = i1 + i2;
18  }
19
20
21
22 }
```

Explicit Conversions

Explicit data conversions are possible but may not always return the desired result.

Explicit conversions may sometimes produce truncated data.

Conversion of a signed integer to an unsigned integer is a reliable explicit conversion

```

15  function addIT() public {
16
17      sum = i1 + i2;
18
19      int y = -3;
20      uint x = uint(y);
21
22      x++;
23  }
```

If an integer is explicitly converted to a smaller type bits are cut off

```

uint32 a = 0x12345678;
uint16 b = uint16(a); // b will be 0x5678 now
```

Events

Events are members of contracts

When events are called the event, arguments are stored in the transaction's log

The logged information can be seen when the transaction is mined.

The log is part of the blockchain and remain with the block.

It is possible to subscribe to an event.

Events are used with emit to manage writing to the log.

```

11
12 event Sent(address from, address to, uint amount);
13
14
15 constructor() public {
16     issuer = msg.sender;
17 }
18
19
20 function issue (address receiver, uint amount) public {
21     //uint256 c = a + b;
22     //require(c >= a);
23     //return c;
24
25     require(msg.sender == issuer);
26     require(amount < 1000);
27     balance[receiver] += amount;
28 }
29
30 // Sends an amount of existing coins
31 // from any caller to an address
32 function send(address receiver, uint amount) public {
33     require(amount <= balance[msg.sender], "Insufficient balance.");
34     balance[msg.sender] -= amount;
35     balance[receiver] += amount;
36     emit Sent(msg.sender, receiver, amount);
37     payee = receiver;
38     payeeBalance = balance[receiver];
39     issuerBal = balance[msg.sender];
40
41 }
42
logs

```

```

{
  "from": "0x0971b5d216af52c411c9016bbc63665b4e6f2542",
  "topic": "0x3990db2d31862302a685e8086b5755072a6e2b5b780af1ee81ece35ee3cd3345",
  "event": "Sent",
  "args": {
    "0": "0xCA35b7d915458EF540aDe6068dFe2F44E8fa733c",
    "1": "0x4B0897b0513fdC7C541B6d9D7E929C4e5364D2dB",
    "2": "60",
    "from": "0xCA35b7d915458EF540aDe6068dFe2F44E8fa733c",
    "to": "0x4B0897b0513fdC7C541B6d9D7E929C4e5364D2dB",
    "amount": "60",
    "length": 3
  }
}

```

For Loop

For loops have a construction like Java, C++, JavaScript etc.

The For loop header consists of the initialization, the condition and the iterator

The expression in the loop will execute as long as the header condition is true.

```

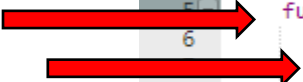
function getStudentCnt() public view returns(uint count) {
    return studentArray.length
}

```

```

function studentLoop() public {
    for (uint i=0; i<studentArray.length; i++) {
        emit LogStudentGrade(studentArray[i], studentStructs[studentArray[i]].grade,
                             studentStructs[studentArray[i]].name);
    }
}

```



```

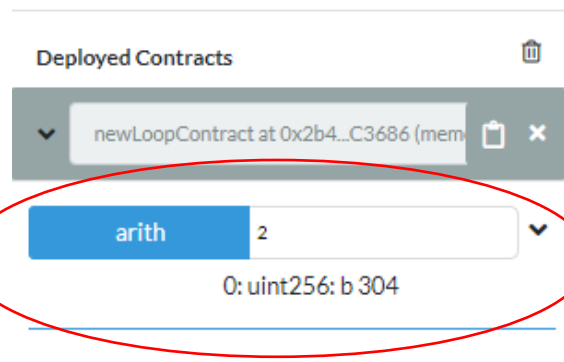
25 function studentLoop() public {
26
27     for (uint i=0; i<studentArray.length; i++) {
28         emit LogStudentGrade(studentArray[i], studentStructs[studentArray[i]].grade, studentStructs[studentArray[i]].name);
29     }
30 }
31

```

```

1 pragma solidity ^0.5.2;
2
3
4 contract newLoopContract {
5     function arith(uint a) public pure returns (uint b) {
6         b = 1;
7         for (uint i = 0; i < a; i++)
8             b = 2 * b + 100;
9     }
10 }

```



While Loop

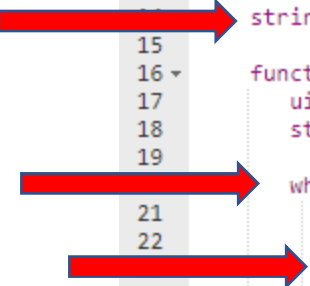
In Solidity, the while loop resembles the same behavior and functionality as Java, C++, JavaScript etc.

The While Loop has a header with a condition

```
while (k < toolsArray.length)
```

The expression in the while loop's braces executes as long as the condition in the header is true.

```
1  pragma solidity ^0.5.1;
2
3  contract Loop {
4
5
6      struct Student {
7          uint grade;
8          string name;
9      }
10
11     mapping(address => Student) public studentStructs;
12     address [] public studentArray;
13     int256 public nmCnt=0;
14     string [] public studentNameArray;
15
16     function studentName() public {
17         uint256 j;
18         string memory nm;
19
20         while (j < studentArray.length){
21
22             nm = studentStructs[studentArray[j]].name;
23             studentNameArray.push(nm);
24             j++;
25             nmCnt++;
26         }
27
28     }
```



Do While Loop

Do while loops again work like Java, C++, and JavaScript

Do While loops are similar to while loops except the condition is evaluated at the end of the loop

Therefore the loop expression will be executed at least once, even if the condition is false.

```

1  pragma solidity ^0.5.2;
2
3
4  contract newLoopContract {
5
6      uint256 public theArea;
7      uint256 public rArea;
8
9      function arith(uint a) public pure returns (uint b) {
10         b = 1;
11         for (uint i = 0; i < a; i++)
12             b = 2 * b + 100;
13     }
14
15
16     function myArea(uint w, uint h) public returns (uint myarea) {
17
18         uint j = 5;
19         theArea = 0;
20
21         do {
22             myarea = w * h;
23             rArea = myarea;
24
25             theArea = theArea + j;
26
27             j--;
28
29         } while (j > 0) ;
30
31     }
32 }
33
34 }

```

Switch Case

Just as in Java, C++, and JavaScript, the switch case is a simplified version of If .. else. A value is compared to another value, or constant. The case equal to the value is return and evaluation ends. The default value is returned when there is no match

```

1  pragma solidity ^0.5.2;
2
3
4  contract newLoopContract {
5
6      uint256 public theArea;
7      uint256 public rArea;
8
9      function searchfor(uint256 myVal) public pure returns (uint256 cVal) {
10         assembly {
11             switch myVal
12             case 0 { cVal := 0 }
13             case 1 { cVal := 1 }
14             case 2 { cVal := 2 }
15             case 3 { cVal := 3 }
16             case 4 { cVal := 4 }
17             case 5 { cVal := 5 }
18             case 6 { cVal := 6 }
19             case 7 { cVal := 7 }
20             case 8 { cVal := 8 }
21
22             default { cVal := 118 }
23         }
24     }
25 }

```

Handling Errors

Given that Solidity is transaction based it uses a state revert model when errors occur.

This prevents wasteful transactions from occurring.

Solidity does not currently support error catching.

Structures such as “try catch” may be added in subsequent versions.

Assert

The assert function is somewhat limited.

Assert is used for testing.

Example: Assert will return an exception if an invalid array index is requested.

Require

The require function is used to ensure only valid values are returned at runtime.

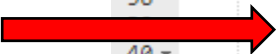
The valid values cover inputs, or contract state variables.

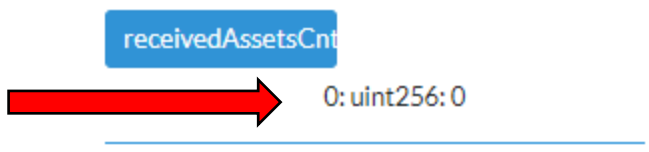
Example: require returns an error if a parameter return false when it should be true.

```

36  function issue (address receiver, uint amount) public {
37
38      uint256 j;
39      require(receivedAssetsCnt > 0, "Received Asset Count must be great than 0");
40      while (j < receivedAssetsCnt){
41          require(msg.sender == issuer);
42          require(amount < 1000);
43          balance[receiver] += amount;
44
45          receivedAssetsCnt++;
46      }
47
48  }

```





```
[vm] from:0xca3...a733c to:MyContract.issue(address,uint256) 0xcbb...04d39 value:0 wei data:0x867...00064 logs:0 hash:0xdf1...64c5a
transact to MyContract.issue errored: VM error: revert.
revert The transaction has been reverted to the initial state.
Reason provided by the contract: "Received Asset Count must be great than 0". Debug the transaction to get more information.
```

Revert

The revert function also triggers exceptions and returns an error.

Revert will flag an error and reverse the call.

An optional message can be passed into Revert to return error details.

More Resources

To read more on blockchain and understand it in depth, the reading the following articles are highly recommended:

- History and Evolution of Blockchain Technology from Bitcoin
- Overview of Blockchain evolution and phases from Ethereum to Hyperledger
- Comprehensive overview and analysis of blockchain use cases in many industries
- Overview of blockchain technology and blockchain development

Also, the following are more tutorials and resources on Ethereum blockchain development.

- How to Write Ethereum Smart Contracts with Solidity in 1 hour
- How to Build Auction DApp with Ethereum and Solidity Programming Language
- How to Work with Ethereum Blockchain Applications through Remix IDE
- Certified Solidity Professional Certification exam
- Learn Ethereum: Build your own DApps with Ethereum and smart contracts book by Brian Wu