Intro to Solidity

Asst. Prof. George Pashev, PhD, MEng.
https://bit.ly/gpashev; https://gpashev.academia.edu

Solidity

Solidity is a contract-oriented, high-level programming language for implementing smart contracts. Solidity is highly influenced by C++, Python and JavaScript and has been designed to target the Ethereum Virtual Machine (EVM).

The Ethereum Virtual Machine (EVM)

The Ethereum Virtual Machine, also known as EVM, is the runtime environment for smart contracts in Ethereum. The Ethereum Virtual Machine focuses on providing security and executing untrusted code by computers all over the world.

The EVM specialised in preventing Denial-of-service attacks and ensures that programs do not have access to each other's state, ensuring communication can be established without any potential interference.

The Ethereum Virtual Machine has been designed to serve as a runtime environment for smart contracts based on Ethereum.

What is Smart Contract?

A smart contract is a computer protocol intended to digitally facilitate, verify, or enforce the negotiation or performance of a contract. Smart contracts allow the performance of credible transactions without third parties. These transactions are trackable and irreversible.

The concept of smart contracts was first proposed by Nick Szabo in 1994. Szabo is a legal scholar and cryptographer known for laying the groundwork for digital currency.

It is fine if you do not understand Smart Contract right now, we will go into more detail later.

Install

Install Node.js

First make sure you have node.js available on your CentOS machine. If it is not available then install it using the following commands –

```
# First install epel-release

$sudo yum install epel-release

# Now install nodejs

$sudo yum install nodejs

# Next install npm (Nodejs Package Manager )

$sudo yum install npm
```

Install

```
# Finally verify installation
$npm --version
```

If everything has been installed then you will see an output something like this –

```
3.10.10
```

Install solc

Once you have Node.js package manager installed then you can proceed to install Solidity compiler as below –

```
$sudonpm install -g solc
```

Install solc

Once you have Node.js package manager installed then you can proceed to install Solidity compiler as below –

```
$sudonpm install -g solc
```

The above command will install solcjs program and will make it available globally through out the system. Now you can test your Solidity compiler by issuing following command –

```
$solcjs-version
```

If everything goes fine, then this will print something as follows –

```
0.5.2+commit.1df8f40c.Emscripten.clang
```

Now you are ready to use solcjs which has fewer features than the standard Solidity compiler but it will give you a good starting point.

Solidity Test APP

```
pragma solidity ^0.5.0;
contract SolidityTest {
   constructor() public{
   }
   function getResult() public view returns(uint){
      uint a = 1;
      uint b = 2;
      uint result = a + b;
      return result;
   }
}
```

RUN

ONLINE GUI: https://remix.ethereum.org/

Step 2 – Under Compile Tab, click Start to Compile button.

Step 3 – Under Run Tab, click Deploy button.

Step 4 – Under Run Tab, Select SolidityTest at 0x... in drop-down.

Step 5 – Click getResult Button to display the result.

Type	Keyword	Values
Boolean	bool	true/false
Integer	int/uint	Signed and unsigned integers of varying sizes.
Integer	int8 to int256	Signed int from 8 bits to 256 bits. int256 is the same as int.
Integer	uint8 to uint256	Unsigned int from 8 bits to 256 bits. uint256 is the same as uint.
Fixed Point Numbers	fixed/unfixed	Signed and unsigned fixed point numbers of varying sizes.
Fixed Point Numbers	fixed/unfixed	Signed and unsigned fixed point numbers of varying sizes.
Fixed Point Numbers	fixedMxN	Signed fixed point number where M represents number of bits taken by type and N represents the decimal points. M should be divisible by 8 and goes from 8 to 256. N can be from 0 to 80. fixed is same as fixed128x18.
Fixed Point Numbers	ufixedMxN	Unsigned fixed point number where M represents number of bits taken by type and N represents the decimal points. M should be divisible by 8 and goes from 8 to 256. N can be from 0 to 80. ufixed is same as ufixed128x18.

address

address holds the 20 byte value representing the size of an Ethereum address. An address can be used to get the balance using .balance method and can be used to transfer balance to another address using .transfer method.

```
address x = 0x212;
address myAddress = this;
if (x.balance < 10 && myAddress.balance >= 10) x.transfer(10);
```

Solidity supports three types of variables.

- **State Variables** − Variables whose values are permanently stored in a contract storage.
- **Local Variables** − Variables whose values are present till function is executing.
- **Global Variables** − Special variables exists in the global namespace used to get information about the blockchain.

Solidity is a statically typed language, which means that the state or local variable type needs to be specified during declaration. Each declared variable always have a default value based on its type. There is no concept of "undefined" or "null".

State Variable

Variables whose values are permanently stored in a contract storage.

```
pragma solidity ^0.5.0;
contract SolidityTest {
   uint storedData;      // State variable
   constructor() public {
      storedData = 10;   // Using State variable
   }
}
```

Local Variable

Variables whose values are available only within a function where it is defined. Function parameters are always local to that function.

```
pragma solidity ^0.5.0;
contract SolidityTest {
   uint storedData; // State variable
   constructor() public {
      storedData = 10;
   }
   function getResult() public view returns(uint){
      uint a = 1; // local variable
      uint b = 2;
      uint result = a + b;
      return result; //access the local variable
   }
}
```

Global Variables

These are special variables which exist in global workspace and provide information about the blockchain and transaction properties.

Name	Returns
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 caller)
msg.sig (bytes4)	First four bytes of the calldata (function identifier)
msg.value (uint)	Number of wei sent with the message
now (uint)	Current block timestamp
tx.gasprice (uint)	Gas price of the transaction

Example

```
pragma solidity ^0.5.0;
contract C {
   uint public data = 30;
   uint internal iData= 10;

   function x() public returns (uint) {
      data = 3; // internal access
      return data;
   }
}
contract Caller {
   C c = new C();
   function f() public view returns (uint) {
      return c.data(); //external access
   }
}
contract D is C {
   function y() public returns (uint) {
      iData = 3; // internal access
      return iData;
   }
   function getResult() public view returns(uint){
      uint a = 1; // local variable
      uint b = 2;
      uint result = a + b;
      return storedData; //access the state variable
   }
```

Bytes to String Conversion

Bytes can be converted to String using string() constructor.

```
bytes memory bstr = new bytes(10);
string message = string(bstr);
```

Example

Initializing Arrays

You can initialize Solidity array elements either one by one or using a single statement as follows −

```
uint balance[3] = [1, 2, 3];
```

The number of values between braces [ ] can not be larger than the number of elements that we declare for the array between square brackets [ ]. Following is an example to assign a single element of the array −

If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write −

```
uint balance[] = [1, 2, 3];
```

You will create exactly the same array as you did in the previous example.

```
balance[2] = 5;
```

```
pragma solidity ^0.5.0;

contract test {
   function testArray() public pure{
      uint len = 7;

      //dynamic array
      uint[] memory a = new uint[](7);

      //bytes is same as byte[]
      bytes memory b = new bytes(len);

      assert(a.length == 7);
      assert(b.length == len);

      //access array variable
      a[6] = 8;

      //test array variable
      assert(a[6] == 8);

      //static array
      uint[3] memory c = [uint(1) , 2, 3];
      assert(c.length == 3);
   }
}
```

Enums

```
pragma solidity ^0.5.0;

contract test {
   enum FreshJuiceSize{ SMALL, MEDIUM, LARGE }
   FreshJuiceSize choice;
   FreshJuiceSize constant defaultChoice = FreshJuiceSize.MEDIUM;

   function setLarge() public {
      choice = FreshJuiceSize.LARGE;
   }
   function getChoice() public view returns (FreshJuiceSize) {
      return choice;
   }
   function getDefaultChoice() public pure returns (uint) {
      return uint(defaultChoice);
   }
}
```

Run the above program using steps provided in Solidity First Application chapter

Defining a Struct

To define a Struct, you must use the **struct** keyword. The struct keyword defines a new data type, with more than one member. The format of the struct statement is as follows −

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

Example

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

```
pragma solidity ^0.5.0;

contract test {
   struct Book {
      string title;
      string author;
      uint book_id;
   }
   Book book;

   function setBook() public {
      book = Book('Learn Java', 'TP', 1);
   }
   function getBookId() public view returns (uint) {
      return book.book_id;
   }
}
```

Mappings

Example

Try the following code to understand how the mapping type works in Solidity.

```
pragma solidity ^0.5.0;

contract LedgerBalance {
   mapping(address => uint) public balances;

   function updateBalance(uint newBalance) public {
      balances[msg.sender] = newBalance;
   }
}
contract Updater {
   function updateBalance() public returns (uint) {
      LedgerBalance ledgerBalance = new LedgerBalance();
      ledgerBalance.updateBalance(10);
      return ledgerBalance.balances(address(this));
   }
}
```

Explicit Conversion

We can explicitly convert a data type to another using constructor syntax.

```
int8 y = -3;
uint x = uint(y);
//Now x = 0xfffff..fd == two complement representation of -3 in 256 bit format.
```

Conversion to smaller type costs higher order bits.

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

Conversion to higher type adds padding bits to the left.

```
uint16 a = 0x1234;
uint32 b = uint32(a); // b = 0x00001234
```

Conversion to smaller byte costs higher order data.

```
bytes2 a = 0x1234;
bytes1 b = bytes1(a); // b = 0x12
```

Ether and Time Units

In solidity we can use wei, finney, szabo or ether as a suffix to a literal to be used to convert various ether based denominations. Lowest unit is wei and 1e12 represents 1 x 10^{12}.

```
assert(1 wei == 1);
assert(1 szabo == 1e12);
assert(1 finney == 1e15);
assert(1 ether == 1e18);
assert(2 ether == 2000 fenny);
```

Time Units

Similar to currency, Solidity has time units where lowest unit is second and we can use seconds, minutes, hours, days and weeks as suffix to denote time.

```
assert(1 seconds == 1);
assert(1 minutes == 60 seconds);
assert(1 hours == 60 minutes);
assert(1 day == 24 hours);
assert(1 week == 7 days);
```

Owned.sol

```
pragma solidity ^0.5.0;

// Owned.sol
contract Owned {
    address public owner;
    constructor() public {
        owner = msg.sender;
    }
    modifier onlyOwner {
        //....
    }
    function transferOwnership(address newOwner) public onlyOwner {
        //...
    }
}
```

Syntax

The basic syntax is shown here.

```
function function-name(parameter-list) scope returns() {
   //statements
}
```

Example

Try the following example. It defines a function called getResult that takes no parameters −

```
pragma solidity ^0.5.0;

contract Test {
   function getResult() public view returns(uint){
      uint a = 1; // local variable
      uint b = 2;
      uint result = a + b;
      return result;
   }
}
```

```
pragma solidity ^0.5.0;

contract SolidityTest {
   constructor() public{
   }
   function getResult() public view returns(string memory){
      uint a = 1;
      uint b = 2;
      uint result = a + b;
      return integerToString(result);
   }
   function integerToString(uint _i) internal pure
      returns (string memory) {
```

```
function getResult() public view returns(uint product, uint sum){
    uint a = 1; // local variable
    uint b = 2;
    product = a * b;
    sum = a + b;

    //alternative return statement to return
    //multiple values
    //return(a*b, a+b);
}
```

Modifiers

```
contract Owner {
    address owner;
    constructor() public {
        owner = msg.sender;
    }
    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }
    modifier costs(uint price) {
        if (msg.value >= price) {
            _;
        }
    }
}
```

```
contract Register is Owner {
    mapping (address => bool) registeredAddresses;
    uint price;
    constructor(uint initialPrice) public { price = initialPrice; }

    function register() public payable costs(price) {
        registeredAddresses[msg.sender] = true;
    }
    function changePrice(uint _price) public onlyOwner {
        price = _price;
    }
}
```

View Functions

View functions ensure that they will not modify the state. A function can be declared as view. The following statements if present in the function are considered modifying the state and compiler will throw warning in such cases.

Modifying state variables.
Emitting events.
Creating other contracts.
Using selfdestruct.
Sending Ether via calls.
Calling any function which is not marked view or pure.
Using low-level calls.
Using inline assembly containing certain opcodes.

Getter method are by default view functions.

Example

```
pragma solidity ^0.5.0;

contract Test {
   function getResult() public view returns(uint product, uint sum){
      uint a = 1; // local variable
      uint b = 2;
      product = a * b;
      sum = a + b;
   }
}
```

Run the above program using steps provided in Solidity First Application chapter.

Output

```
0: uint256: product 2
1: uint256: sum 3
```

Pure functions ensure that they not read or modify the state. A function can be declared as pure. The following statements if present in the function are considered reading the state and compiler will throw warning in such cases.

Reading state variables.

Accessing address(this).balance or <address>.balance.

Accessing any of the special variable of block, tx, msg (msg.sig and msg.data can be read).

Calling any function not marked pure.

Using inline assembly that contains certain opcodes.

Pure functions can use the revert() and require() functions to revert potential state changes if an error occurs.

Example

```
pragma solidity ^0.5.0;

contract Test {
   function getResult() public pure returns(uint product, uint sum){
      uint a = 1;
      uint b = 2;
      product = a * b;
      sum = a + b;
   }
}
```

Run the above program using steps provided in Solidity First Application chapter.

Output

```
0: uint256: product 2
1: uint256: sum 3
```

Fallback function is a special function available to a contract. It has following features –

- It is called when a non-existent function is called on the contract.
- It is required to be marked external.
- It has no name.
- It has no arguments
- It can not return any thing.
- It can be defined one per contract.
- If not marked payable, it will throw exception if contract receives plain ether without data.

Following example shows the concept of a fallback function per contract.

Example

```
pragma solidity ^0.5.0;

contract Test {
   uint public x ;
   function() external { x = 1; }
}
contract Sink {
   function() external payable { }
}
contract Caller {
   function callTest(Test test) public returns (bool) {
      (bool success,) = address(test).call(abi.encodeWithSignature("nonExistingFunction()"));
      require(success);
      // test.x is now 1
```

Function overloading

Example

```
pragma solidity ^0.5.0;

contract Test {
   function getSum(uint a, uint b) public pure returns(uint){
      return a + b;
   }
   function getSum(uint a, uint b, uint c) public pure returns(uint){
      return a + b + c;
   }
   function callSumWithTwoArguments() public pure returns(uint){
      return getSum(1,2);
   }
   function callSumWithThreeArguments() public pure returns(uint){
      return getSum(1,2,3);
   }
}
```

Cryptographic Functions

Solidity provides inbuilt cryptographic functions as well. Following are important methods –

keccak256(bytes memory) returns (bytes32) – computes the Keccak-256 hash of the input.

ripemd160(bytes memory) returns (bytes20) – compute RIPEMD-160 hash of the input.

sha256(bytes memory) returns (bytes32) – computes the SHA-256 hash of the input.

ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address) – recover the address associated with the public key from elliptic curve signature or return zero on error. The function parameters correspond to ECDSA values of the signature: r - first 32 bytes of signature; s: second 32 bytes of signature; v: final 1 byte of signature. This method returns an address.

Visibility Quantifiers

Following are various visibility quantifiers for functions/state variables of a contract.

external – External functions are meant to be called by other contracts. They cannot be used for internal call. To call external function within contract this.function_name() call is required. State variables cannot be marked as external.

public – Public functions/ Variables can be used both externally and internally. For public state variable, Solidity automatically creates a getter function.

internal – Internal functions/ Variables can only be used internally or by derived contracts.

private – Private functions/ Variables can only be used internally and not even by derived contracts.

Inheritance is a way to extend functionality of a contract. Solidity supports both single as well as multiple inheritance. Following are the key highlighsts.

A derived contract can access all non-private members including internal methods and state variables. But using this is not allowed.

Function overriding is allowed provided function signature remains same. In case of difference of output parameters, compilation will fail.

We can call a super contract's function using super keyword or using super contract name.

In case of multiple inheritance, function call using super gives preference to most derived contract.

Example

```
//Derived Contract
contract E is C {
   uint private result;
   C private c;
   constructor() public {
      c = new C();
   }
   function getComputedResult() public {
      result = compute(3, 5);
   }
   function getResult() public view returns(uint) { return result; }
   function getData() public view returns(uint) { return c.info(); }
}
```

Constructors

```
pragma solidity ^0.5.0;

contract Base {
   uint data;
   constructor(uint _data) public {
      data = _data;
   }
}
contract Derived is Base {
   constructor(uint _info) Base(_info * _info) public {}
}
```

Abstract contract

Example

Try the following code to understand how the abstract contracts works in Solidity.

```
pragma solidity ^0.5.0;

contract Calculator {
   function getResult() public view returns(uint);
}
contract Test is Calculator {
   function getResult() public view returns(uint) {
      uint a = 1;
      uint b = 2;
      uint result = a + b;
      return result;
   }
}
```

Interfaces

Interfaces are similar to abstract contracts and are created using interface keyword. Following are the key characteristics of an interface.

- Interface can not have any function with implementation.
- Functions of an interface can be only of type external.
- Interface can not have constructor.
- Interface can not have state variables.
- Interface can have enum, structs which can be accessed using interface name dot notation.

Libraries

Libraries are similar to Contracts but are mainly intended for reuse. A Library contains functions which other contracts can call. Solidity have certain restrictions on use of a Library. Following are the key characteristics of a Solidity Library.

Library functions can be called directly if they do not modify the state. That means pure or view functions only can be called from outside the library.

Library can not be destroyed as it is assumed to be stateless.

A Library cannot have state variables.

A Library cannot inherit any element.

A Library cannot be inherited.

```
pragma solidity ^0.5.0;

library Search {
   function indexOf(uint[] storage self, uint value) public view returns (uint) {
      for (uint i = 0; i < self.length; i++) if (self[i] == value) return i;
      return uint(-1);
   }
}
contract Test {
   uint[] data;
   constructor() public {
      data.push(1);
      data.push(2);
      data.push(3);
      data.push(4);
      data.push(5);
   }
   function isValuePresent() external view returns(uint){
      uint value = 4;

      //search if value is present in the array using Library function
      uint index = Search.indexOf(data, value);
      return index;
   }
}
```

Events

First Create a contract and emit an event.

```
pragma solidity ^0.5.0;

contract Test {
   event Deposit(address indexed _from, bytes32 indexed _id, uint _value);
   function deposit(bytes32 _id) public payable {
      emit Deposit(msg.sender, _id, msg.value);
   }
}
```

Then access the contract's event in JavaScript code.

```
var abi = /* abi as generated using compiler */;
var ClientReceipt = web3.eth.contract(abi);
var clientReceiptContract = ClientReceipt.at("0x1234...ab67" /* address */);

var event = clientReceiptContract.Deposit(function(error, result) {
   if (!error)console.log(result);
});
```

Error Handling

Solidity provides various functions for error handling. Generally when an error occurs, the state is reverted back to its original state. Other checks are to prevent unauthorized code access. Following are some of the important methods used in error handling –

assert(bool condition) – In case condition is not met, this method call causes an invalid opcode and any changes done to state got reverted. This method is to be used for internal errors.

require(bool condition) – In case condition is not met, this method call reverts to original state. - This method is to be used for errors in inputs or external components.

require(bool condition, string memory message) – In case condition is not met, this method call reverts to original state. - This method is to be used for errors in inputs or external components. It provides an option to provide a custom message.

revert() – This method aborts the execution and revert any changes done to the state.

revert(string memory reason) – This method aborts the execution and revert any changes done to the state. It provides an option to provide a custom message.