Blockchain Technology Module 4:Solidity

Introducing solidity

- Solidity is a domain-specific language of choice for programming contracts in Ethereum.
- Its syntax is closer to JavaScript and C.
- It is a statically typed language, which means that variable type checking in solidity is carried out at compile time.
- Other features of the language include inheritance, libraries, and the ability to define composite data types.
- Solidity is also a called contract-oriented language.
- In solidity, contracts are equivalent to the concept of classes in other object-oriented programming languages.

Types

- Solidity has two categories of data types: value types and reference types.
- VALUE TYPES
- Boolean
  - This data type has two possible values, true or false, for example: bool v = true; This statement assigns the value true to v.
- Integers
  - This data type represents integers. A table is shown here, which shows various keywords used to declare integer data types.

Keyword	Types	Details
`int`	Signed integer	`int8` to `int256`, which means that keywords are available from `int8` up to `int256` in increments of 8, for example, `int8`, `int16`, `int24`.
`uint`	Unsigned integer	`uint8`, `uint16`, ... to `uint256`, unsigned integer from 8 bits to 256 bits. The usage is dependent on the requirements that how many bits are required to be stored in the variable.

- For example, in this code, note that uint is an alias for uint256:
- uint256 x;
- uint y;
- int256 z;
- These types can also be declared with the constant keyword, which means that no storage slot will be reserved by the compiler for these variables. In this case, each occurrence will be replaced with the actual value:
- uint constant z=10+10;

Address

- This data type holds a 160-bit long (20 byte) value. This type has several members that can be used to interact with and query the contracts. These members are described here:
- Balance
  - The balance member returns the balance of the address in Wei.
- Send
  - This member is used to send an amount of ether to an address (Ethereum's 160-bit address) and returns true or false depending on the result of the transaction
  - Example:

```
address to = 0x6414cc08d148dce9ebf5a2d0b7c220ed2d3203da;
address from = this;
if (to.balance < 10 && from.balance > 50) to.send(20);
```

- Call functions
  - The call, callcode, and delegatecall are provided in order to interact with functions that do not have Application Binary Interface (ABI)

- Array value types (fixed size and dynamically sized byte arrays):
- Solidity has fixed size and dynamically sized byte arrays.
- Fixed size keywords range from bytes1 to bytes32, whereas dynamically sized keywords include bytes and string.
- The bytes keyword is used for raw byte data and string is used for strings encoded in UTF-8.
- As these arrays are returned by the value, calling them will incur gas cost. length is a member of array value types and returns the length of the byte array.
- An example of a static (fixed size) array is as follows:
  - bytes32[10] bankAccounts;
- An example of a dynamically sized array is as follows:
  - bytes32[] trades;
- Get length of trades by using the following code:
  - trades.length;

LITERALS

- These are used to represent a fixed value.
- Integer literals
  - Integer literals are a sequence of decimal numbers in the range of 0-9. An example is shown as follows:
  - uint8 x = 2;
- String literals
  - String literals specify a set of characters written with double or single quotes. An example is shown as follows:
  - 'packt' "packt“
- Hexadecimal literals
  - Hexadecimal literals are prefixed with the keyword hex and specified within double or single quotation marks.
  - An example is shown as follows: (hex'AABBCC');

ENUMS

- This allows the creation of user-defined types. An example is shown as follows:

```
enum Order{Filled, Placed, Expired };
Order private ord;
ord=Order.Filled;
```

- Explicit conversion to and from all integer types is allowed with enums

FUNCTION TYPES

- There are two function types: internal and external functions.
- Internal functions
  - These can be used only within the context of the current contract.
- External functions
  - External functions can be called via external function calls.
  - A function in solidity can be marked as a constant.
  - Constant functions cannot change anything in the contract; they only return values when they are invoked and do not cost any gas.

- The syntax to declare a function is shown as follows:

```
function <nameofthefunction> (<parameter types> <name of the variable>)
{internal|external} [constant] [payable] [returns (<return types> <name of the variable>)]
```

```
function function_name(parameter_list) scope returns(return_type)
{
      // block of code
}
```

- Example:

```
function add() public pure returns(uint)
{
    uint num1 = 10;
    uint num2 = 16;
    uint sum = num1 + num2;
    return sum;
}
```

```
• contract SquareFinder
• {
•     // Function to find the square of a number
•     function findSquare(uint num) public pure returns (uint) {
•         return num * num;
•     }
• }
```

```
function add(uint a, uint b) public pure returns (uint)
{
    return a + b;  // only uses local data
}
```

Function Calling

```
function sqrt(uint num) public pure returns(uint)
{
    num = num*num;
    return  num;
}
// Defining a public pure function to demonstrate
// calling of sqrt function
function add() public pure returns(uint)
{
    uint num1 = 10;
    uint num2 = 16;
    uint sum = num1 + num2;
    return sqrt(sum); // calling function
}
```

Return Statements

- In Solidity, more than one value can be returned from a function.
- To return values from the function, the data types of the return values should be defined at function declaration.

```
function returnex() public pure returns(uint, uint, uint, string memory)
{
    uint num1 = 10;
    uint num2 = 16;
    uint sum = num1 + num2;
    uint prod = num1 * num2;
    uint diff = num2 - num1;
    string memory message = "Multiple return values";
    return (sum, prod, diff, message);
}
```

REFERENCE TYPES

- Arrays
- Arrays represent a contiguous set of elements of the same size and type laid out at a memory location.
- The concept is the same as any other programming language.
- Arrays have two members named length and push:
- uint[] OrderIds;

```
contract Example {
  uint[] public numbers;  // dynamic array

  function demoLength() public returns (uint) {
    numbers.push(10);
    numbers.push(20);
    return numbers.length; // returns 2
  }
}
```

- Structs
- These constructs can be used to group a set of dissimilar data types under a logical group.
- These can be used to define new types, as shown in the following
- example:

```
Struct Trade
{
    uint tradeid;
    uint quantity;
    uint price;
    string trader;
}
```

Data location

- Data location specifies where a particular complex data type will be stored.
- Depending on the default or annotation specified, the location can be storage or memory. This is applicable to arrays and structs and can be specified using the storage or memory keywords.
- As copying between memory and storage can be quite expensive, specifying a location can be helpful to control the gas expenditure at times.
- Calldata is another memory location that is used to store function arguments.
- Parameters of external functions use calldata memory.
- By default, parameters of functions are stored in memory, whereas all other local variables make use of storage.
- S.tate variables, on the other hand, are required to use storage

```
function sumValues(uint[] calldata nums) external pure returns
(uint total)
{
      for (uint i = 0; i < nums.length; i++)
    {
        total += nums[i]; // can read nums, but cannot modify it
      }
   }
```

Mappings

- Mappings are used for a key to value mapping.
- This is a way to associate a value with a key.
- All values in this map are already initialized with all zeroes, for example, the following:
  - mapping (address => uint) offers;
- This example shows that offers is declared as a mapping.
- Another example makes this clearer:
  - mapping (string => uint) bids;
  - bids["packt"] = 10;

```
bids = {
  "packt": 10,
  "book": 0, // default value, since not set yet
  "code": 0    // default value
}
```

- This is basically a dictionary or a hash table where string values are mapped to integer values. The mapping named bids has string packt mapped to value 10.

Global variables

- Solidity provides a number of global variables that are always available in the global namespace.
- These variables provide information about blocks and transactions.
- Additionally, cryptographic functions and address related variables are available as well.
- A subset of available functions and variables is shown as follows:
  - keccak256(...) returns (bytes32)
- This function is used to compute the Keccak-256 hash of the argument provided to the function:
  - ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)
- This function returns the associated address of the public key from the elliptic curve signature:
  - block.number
- This returns the current block number.

Control structures

- Control structures available in solidity language are if...else, do, while, for, break, continue, and return.
- They work exactly the same as other languages such as C-language or JavaScript.
- Some examples are shown here:
- if: If x is equal to 0 then assign value 0 to y else assign 1 to z:

```
if (x == 0)
y = 0;
else
z = 1;
```

- do: Increment x while z is greater than 1:

```
do{
x++;
} (while z>1);
```

- while: Increment z while x is greater than 0:

```
while(x > 0)
{ z++; }
```

- for, break, and continue: Perform some work until x is less than or equal to 10.
- This for loop will run 10 times, if z is 5 then break the for loop:

```
for(uint8 x=0; x<=10; x++) {
//perform some work z++
if(z == 5)
break; }
```

- It will continue the work similarly, but when the condition is met, the loop will start again.
- return: Return is used to stop the execution of a function and returns an optional value. For example:

```
return 0;
```

- It will stop the execution and return value of 0.

Events

- Events in Solidity can be used to log certain events in EVM logs.
- These are quite useful when external interfaces are required to be notified of any change or event in the contract.
- These logs are stored on the blockchain in transaction logs.
- Logs cannot be accessed from the contracts but are used as a mechanism to notify change of state or the occurrence of an event (meeting a condition) in the contract.
- In a simple example here, the valueEvent event will return true if the x parameter passed to function Matcher is equal to or greater than 10:

```
pragma solidity ^0.4.0;
contract valueChecker {
uint8 price=10;
event valueEvent(bool returnValue);
function Matcher(uint8 x) public returns (bool) {
if (x>=price) {
valueEvent(true);
return true; } } }
```

Inheritance

- Inheritance is supported in Solidity.
- The is keyword is used to derive a contract from another contract.
- In the following example, valueChecker2 is derived from the valueChecker contract.
- The derived contract has access to all non-private members of the parent contract:

```
pragma solidity ^0.4.0;
contract valueChecker {
uint8 price = 20;
event valueEvent(bool returnValue);
function Matcher(uint8 x) public returns (bool) {
if (x>=price) {
valueEvent(true);
return true; } } }

contract valueChecker2 is valueChecker {
function Matcher2() public view returns (
uint) { return price+10; } }
```

- In the preceding example, if the uint8 price = 20 is changed to uint8 private price = 20, then it will not be accessible by the valueChecker2 contract. This is because now the member is declared as private, it is not allowed to be accessed by any other contract.
- The error message that you will see in Remix is

```
browser/valuechecker.sol:20:8: DeclarationError: Undeclared identifier.
```

Libraries

- Libraries are deployed only once at a specific address and their code is called via CALLCODE or DELEGATECALL opcode of the EVM.
- The key idea behind libraries is code reusability.
- They are similar to contracts and act as base contracts to the calling contracts.
- A library can be declared as shown in the following example:

```
library Addition {
function Add(uint x,uint y) returns (uint z) {
return x + y; } }
```

- This library can then be called in the contract, as shown here.
- First, it needs to be imported and then it can be used anywhere in the code. A simple example is shown as follows:

```
import "Addition.sol"
function Addtwovalues() returns(uint) {
return Addition.Add(100,100); }
```

- There are a few limitations with libraries; for example, they cannot have state variables and cannot inherit or be inherited. Moreover, they cannot receive Ether either; this is in contrast to contracts

Functions

- Functions in Solidity are modules of code that are associated with a contract. Functions are declared with a name, optional parameters, access modifier, optional constant keyword, and optional return type. This is shown in the following example:

  function orderMatcher (uint x)

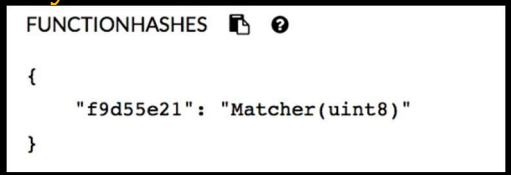
  private constant returns(bool return value)

- In the preceding example, function is the keyword used to declare the function. orderMatcher is the function name, uint x is an optional parameter, private is the access modifier or specifier that controls access to the function from external contracts, constant is an optional keyword used to specify that this function does not change anything in the contract but is used only to retrieve values from the contract and returns (bool return value) is the optional return type of the function.

- How to define a function:
- The syntax of defining a function is shown as follows:

```
function <name of the function>(<parameters>) <visibility specifier> returns
(<return data type> <name of the variable>)
{
    <function body>
}
```

- Function signature:
- Functions in Solidity are identified by its signature, which is the first four bytes of the Keccak-256 hash of its full signature string. This is also visible in Remix IDE, as shown in the following screenshot.
- f9d55e21 is the first four bytes of 32-byte Keccak-256 hash of the function named Matcher.



- Input parameters of a function:
- Input parameters of a function are declared in the form of
- <data type> <parameter name> . This example clarifies the concept where uint x and uint y are input parameters of the checkValues function:

```
contract myContract
{
    function checkValues(uint x, uint y)
    {
    }
}
```

- Output parameters of a function: Output parameters of a function are declared in the form of <data type> <parameter name>. This example shows a simple function returning a uint value:

```
contract myContract
{
    function getValue() returns (uint z)
    {
        z=x+y;
    }
}
```

- Internal function calls:
  - Functions within the context of the current contract can be called internally in a direct manner.
  - These calls are made to call the functions that exist within the same contract. These calls result in simple JUMP calls at the EVM bytecode level.
- External function calls:
  - External function calls are made via message calls from a contract to another contract.
  - In this case, all function parameters are copied to the memory.
  - If a call to an internal function is made using the this keyword, it is also considered an external call.
  - The this variable is a pointer that refers to the current contract. It is explicitly convertible to an address and all members for a contract are inherited from the address.

- Fallback functions:
  - This is an unnamed function in a contract with no arguments and return data. This function executes every time Ether is received.
  - It is required to be implemented within a contract if the contract is intended to receive Ether; otherwise, an exception will be thrown and Ether will be returned.
  - This function also executes if no other function signatures match in the contract. If the contract is expected to receive Ether, then the fallback function should be declared with the payable modifier.
  - The payable is required; otherwise, this function will not be able to receive any Ether. This function can be called using the address.call() method as, for example, in the following

```
function ()
{
    throw;
}
```

- Modifier functions:
  - These functions are used to change the behavior of a function and can be called before other functions.
  - _ (underscore) is used in the modifier functions that will be replaced with the actual body of the function when the modifier is called. Basically, it symbolizes the function that needs to be guarded.
- Constructor function:
  - This is an optional function that has the same name as the contract and is executed once a contract is created. Constructor functions cannot be called later on by users, and there is only one constructor allowed in a contract. This implies that no overloading functionality is available.

- Function visibility specifiers (access modifiers):
  - Functions can be defined with four access specifiers as follows:
  - External: These functions are accessible from other contracts and transactions. They cannot be called internally unless the this keyword is used.
  - Public: By default, functions are public. They can be called either internally or using messages.
  - Internal: Internal functions are visible to other derived contracts from the parent contract.
  - Private: Private functions are only visible to the same contract they are declared in.
- Function Modifiers:
  - pure: This modifier prohibits access or modification to state
  - view: This modifier disables any modification to state
  - payable: This modifier allows payment of ether with a call constant: This modifier disallows access or modification to state
- Other important keywords/functions throw:
  - throw is used to stop execution. As a result, all state changes are reverted. In this case, no gas is returned to the transaction originator because all the remaining gas is consumed.

Import

- Import in Solidity allows the importing of symbols from the existing Solidity files into the current global scope. This is similar to import statements available in JavaScript, as for example, in the following:
- import "module-name";

Comments, Version pragma, Layout of a Solidity source code file

```
1  pragma solidity ^0.4.0; //specify the compiler version
2  /*
3  This is a simple value checker contract that checks the value
4  provided and returns boolean value based on the condition
5  expression evaluation.
6  */
7  import "dev.oraclize.it/api.sol";
8  contract valuechecker {
9      uint price=10;
10     //This is price variable declare and initialized with value 10
11     event valueEvent(bool returnValue);
12     function Matcher (uint8 x) returns (bool)
13     {
14         if ( x >= price)
15         {
16             valueEvent(true);
17             return true;
18         }
19     }
20 }
```

Sample Solidity program as shown in Remix IDE