Guide to important features of Ethereum’s main programming language.

Introduction:

Solidity is considered the main programming language for writing Ethereum smart contracts, a very important building block of the Ethereum ecosystem. A smart contract is a program that runs on the Ethereum Virtual Machine (EVM) which consists of some code such as functions, and some data which presents the current state of the contract on the blockchain. The data consists of variables that hold some data type (Integer, String, Array, structs, …) Smart contracts are also considered a type of Ethereum account. This means that they don’t only encapsulate some logic, but they also can send transactions to other accounts and hold a balance using the main virtual currency Ether defined by Ethereum. Once a smart contract is deployed to the blockchain it can be interacted with via transactions. A transaction originating from a user or another smart contract can execute some function written in the called smart contract. Contracts define a set of rules (code) that enforces how the world interacts with them. Once a contract is deployed to the Ethereum blockchain it will exist there permanently, the same goes for modifying the state of contracts which is also irreversible.

1. Version pragma

Every contract written in solidity must declare the version of the solidity compiler. This declaration allows for future proof code which will prevent newer versions of solidity from breaking the code.

pragma solidity >=0.5.0 <0.6.0;  
   
contract myContract {  
  
  
  
}

we can provide a range of compiler versions. For example, we could compile the code to be compatible with any version between 0.5.0 (inclusive) and 0.6.0 (exclusive).

1. State variables and datatypes

State variables present the current state of a contract which is permanently stored in the blockchain. we can think of this as a database for the contact. Each state variable defined in the contract must have a data type associated with it. For example, "uint" is a datatype that presents unsigned integers of a maximum size of 256bit

uint stateVar = 0;

below is a shortlist of some of the datatypes used in Solidity

Structs:

This is a more complex data type that allows to create blueprints with multiple properties.

struct Person {  
 uint age;  
 string name;  
}

We can create an instance of the struct using the following syntax

Person user = Person(25,**"alex"**);

Arrays:

two types of arrays in solidity: fixed-size arrays and dynamic arrays. Fixed-size arrays will be defined with a size limit and dynamically sized arrays have no fixed size and they can keep growing.

// dynamic array of type Person  
Person[] public users;  
  
// fixed size array of type unit  
uint[100] fixedSizeArray;

Arrays can be accessed via indexing we can also add to an array using push() method as follows:

string temp = users[0];  
  
users.push(**"alex"**);

Int:

Int is a datatype that hold singed integers of maximum size of 256-bit

// singed integer

int neg = -999;

strings:

string str = **"This is a string"**;

1. Functions in Solidity.

Functions in solidity are written in a very similar fashion as written in other high-level programming leagues. We provide the name of the function, parameters that it can take, and a body. The parameters must be declared with a datatype. For parameters that are reference types such as arrays, structs, mappings, and strings, it is required to provide a special keyword to tell the compiler where they should be stored. When passing by value memory keyword is used which means that the compiler will make a new copy of the passed parameter and pass it to the function. This allows the function to modify the parameter without changing the original variable. Passing by reference uses the keyword storage which allows a function to change the origin of the variable. We also should provide an access modifier keyword for the function. In solidity functions public by default can be accessed inside and outside of the contract also we can make functions only accessible from the inside of the contract by using the keyword Private.

function passByVal (string memory \_name) public {  
  
  
}  
  
function passByRef (string storage \_name) private {  
  
}

functions in solidity can return some value that we can specify. Return values must also have a type.

function returnValFunc (string storage \_name) public returns (string memory) {  
  
  
 return str;  
}

solidity offers extra keywords that add more functionality to the functions. View and pure keywords can be used to specify how a function behaves in terms of accessing the data in the contract. View functions are intended to only return a view of the data and not modify it in any way. Pure functions are stricter and they are not intended to view or modify the state variables of the contracts.

function viewFunc (string storage \_name) public returns (string memory) {  
  
  
 return str;  
}  
  
function pureFunc () private pure returns (uint) {  
  
  
 return 10 % 60;  
}

Other than Private and Public visibility modifiers, there are two other special modifiers that solidity offers. Those are internal and external modifiers. Internal is like private, with the exception that internal functions can be accessed by child contracts. External functions are like public functions, although they can only be called externally by an account and not by some function inside a contract.

1. Keccak256 and Typecasting

Solidity offers access to an important built Ethereum hash function named keccak256. This hash function is a version of SHA3. Just like any hash function this function takes an input and maps it to a random 256-bit hexadecimal number. keccak256 will only take an input of type Bytes this means we must pack the parameters using another helper function such as abi.encodePacked

keccak256(abi.encodePacked(**"ABCD00232"**));

Solidity offers an important feature which is typecasting. This allows for conversion between data types. For example, we can typecast the hexadecimal value returned from keccak256 to an unsigned integer.

uint rand = uint(keccak256(abi.encodePacked(**"ABCD00232"**)));

1. Events.

Events are a way to make a contract communicate with the front-end that something happened on the blockchain specifically on the smart contract. The front-end can set up a listener for the event while the smart contract can declare and fire this event programmatically. Below is an event trigging function that fires the event after adding a user to the users array.

// event declaration  
event newPersonAdded(uint Id, string name);

function onUserAdded(string memory \_name) private{  
  
  
 Person user = Person(\_ID,\_name);  
  
 uint id = users.push(user);  
  
 emit newPersonAdded(id,\_name);  
  
}

JavaScript front-end event handler function

myContract.newPersonAdded(

// callback function  
 function(error, result) {  
  
 console.log(result.Id);  
 console.log(result.name);  
 }  
)

1. addresses and mappings.

Each account on the Ethereum blockchain has a unique address be that a contract or an externally owned account. We can address accounts or contracts using this address. In solidity there is a datatype which can be used to store the address of an account or a contract. A use case of addresses can be seen with mappings. Mappings is a collection datatype. It is like the concept of a lookup dictionary with key-value store. This provides a more organized and structured way to store data on the blockchain. We define a map by giving it a name and key, value types. The syntax is written as follows.

// mapping declaration  
  
mapping (address => uint) public accountBalance;

Solidity has a useful global variable that stores the address of the contract or user account that called the target contract. This variable can be accessed in any part of the contract by calling msg.sender this call will return the address of the caller which is of type Address. The code bellow demonstrate a use case of mappings and msg.sender.

function setMappingValue(uint \_bal) private{  
  
accountBalance[msg.sender] = \_bal;  
}

function getMappingValue() public view returns(uint){  
  
 return accountBalance[msg.sender];  
}

1. Inheritance in solidity.

Solidity supports contract inheritance. A child contract can inherit from a parent contract. This gives the advantage of breaking the logic into multiple contracts.

contract Person{  
  
string name;  
uint age;  
  
 constructor(string memory \_name,uint \_age) public{  
  
 name = \_name;  
 age = \_age;  
  
 }  
  
}  
  
contract employee is Person{  
  
 uint id;  
  
 constructor(uint \_id, string memory \_name,uint age ) Person(\_name,\_age) public{  
  
 id = \_id;  
 }  
}

1. The difference between Storage and Memory in solidity.

Variables that use Storge are stored permanently on the blockchain. For example, all the state variables defined in the contract are in Storage. Memory variables are stored in memory, and they only live for the execution time of a function, and they are discarded after the function returns and the program finishes. By default, all variables declared inside a function are memory, so there is no need to use the memory keyword. The code below explains this idea using an example:

contract CarFactory {  
  
 struct Car {  
 string name;  
 string make;  
 string model;  
 string state;  
 }  
  
 Car[] cars;  
  
 function useCar(uint \_index) public {  
 // solidity compiler will be telling you that you should explicitly declare `storage` or `memory` here.  
 //Car myCar = cars[\_index];  
  
  
 // So instead, you should declare with the `storage` keyword, like:  
 Car storage myCar = cars[\_index];  
 // ...in which case `myCar` is a pointer to `cars[\_index]`  
 // in storage, and...  
 myCar.state = **"running"**;  
 // ...this will permanently change `cars[\_index]` on the blockchain.  
  
 // If you just want a copy, you can use `memory`:  
 Car memory myCarCopy = cars[\_index + 1];  
 // ...in which case `myCarCopy` will simply be a copy of the  
 // data in memory, and...  
 myCarCopy.state = **"stopped!"**;  
 // ...will just modify the temporary variable and have no effect  
 // on `cars[\_index + 1]`. But you can do this:  
 cars[\_index + 1] = myCarCopy;  
 // ...if you want to copy the changes back into blockchain storage.  
 }  
}

1. Solidity interface definition.

Solidity supports the creation of interfaces. This is useful when we want to call some function in an already deployed smart contract. We first need to define an interface for it. Below is a simple example of what an interface contract looks like.

// Interface definition  
contract NumberInterface {  
 function getNum(address \_myAddress) public view returns (uint);  
}

After defining an interface, we can make a call to the function of the external contract. One important piece of data that we need is the address of the contract that we want to call. Bellow a simple example of calling getNum which exists externally.

// Interface definition  
contract NumberInterface {  
 function getNum(address \_myAddress) public view returns (uint);  
}  
  
  
contract MyContract {  
  
 address NumberInterfaceAddress = 0xab38;  
 // ^ The address of the external contract on Ethereum  
 NumberInterface numberContract = NumberInterface(NumberInterfaceAddress);  
 // Now `numberContract` is pointing to the other contract  
  
 function someFunction() public {  
 // Now we can call `getNum` from that contract:  
 uint num = numberContract.getNum(msg.sender);  
 // ...and do something with `num` here  
 }  
}

1. Contract ownership

Solidity offers a special type of contract, that is ownable contracts. This type of smart contract allows a user to own it by setting a link between a contract and the user that deployed it. Setting ownership of a contract happens at deployment time when a contract. During the contract's deployment, the special function constructor is called once and only. This sets the owner variable to the address that was first called the constructor. Ownable contracts offer the “onlyOwner” function modifier, which restricts the execution of functions to only the owner. Below is a full starter ownable contract from OpenZeppelin library

\*\*  
 \* ***@title*** Ownable  
 \* ***@dev*** The Ownable contract has an owner address, and provides basic authorization control  
 \* functions, this simplifies the implementation of "user permissions".  
 \*/  
contract Ownable {  
 address private \_owner;  
  
 event OwnershipTransferred(  
 address indexed previousOwner,  
 address indexed newOwner  
 );  
  
 /\*\*  
 \* ***@dev*** The Ownable constructor sets the original `owner` of the contract to the sender  
 \* account.  
 \*/  
 constructor() internal {  
 \_owner = msg.sender;  
 emit OwnershipTransferred(address(0), \_owner);  
 }  
  
 /\*\*  
 \* ***@return*** the address of the owner.  
 \*/  
 function owner() public view returns(address) {  
 return \_owner;  
 }  
  
 /\*\*  
 \* ***@dev*** Throws if called by any account other than the owner.  
 \*/  
 modifier onlyOwner() {  
 require(isOwner());  
 \_;  
 }  
  
 /\*\*  
 \* ***@return*** true if `msg.sender` is the owner of the contract.  
 \*/  
 function isOwner() public view returns(bool) {  
 return msg.sender == \_owner;  
 }  
  
 /\*\*  
 \* ***@dev*** Allows the current owner to relinquish control of the contract.  
 \* ***@notice*** Renouncing to ownership will leave the contract without an owner.  
 \* It will not be possible to call the functions with the `onlyOwner`  
 \* modifier anymore.  
 \*/  
 function renounceOwnership() public onlyOwner {  
 emit OwnershipTransferred(\_owner, address(0));  
 \_owner = address(0);  
 }  
  
 /\*\*  
 \* ***@dev*** Allows the current owner to transfer control of the contract to a newOwner.  
 \* ***@param*** newOwner The address to transfer ownership to.  
 \*/  
 function transferOwnership(address newOwner) public onlyOwner {  
 \_transferOwnership(newOwner);  
 }  
  
 /\*\*  
 \* ***@dev*** Transfers control of the contract to a newOwner.  
 \* ***@param*** newOwner The address to transfer ownership to.  
 \*/  
 function \_transferOwnership(address newOwner) internal {  
 require(newOwner != address(0));  
 emit OwnershipTransferred(\_owner, newOwner);  
 \_owner = newOwner;  
 }  
}

Sources:

<https://cryptozombies.io/>

<https://docs.openzeppelin.com/contracts/4.x/>