**Solidity Smart Contracts**

**Table of Contents**

* **Creating Smart Contracts**
* **Constructor**
* **Scope visibility**
* **State variable visibility**
* **Functions**
* **Function visibility**
* **Getter functions**
* **Setter functions**
* **Function modifiers**
* **Return variables and returning multiple values**
* **Immutable state variables**
* **Payable functions**

**Creating Smart Contracts**

Smart contracts are basically computer programmes that are recorded on a blockchain and are activated when certain criteria are satisfied. A typical application of this technology is to automate the implementation of an agreement so that all participants may be certain of the conclusion instantly, without the involvement of a middleman or the loss of valuable time. When certain conditions are met, they can also automate a workflow, triggering the next action to be performed.

Contracts in Solidity are analogous to classes in object-oriented languages in their functionality. It contains data that is persistent in state variables and functions that can modify the data in these variables. It is possible to execute a function on a separate contract (instance) by executing an EVM function call. This will swap the context, preventing access to state variables in the calling contract. Before anything can happen, it is necessary to invoke the contract and its functions. In Ethereum, there is no idea of a "cron" that can be used to automatically call a function at a specific event.

Smart contracts function by executing basic "if/when...then..." statements that are typed into code and stored on a blockchain. Smart contracts are becoming increasingly popular. When the preset criteria are met and validated, the operations are carried out by a network of computers connected together. These measures could include releasing funds to the proper parties, registering a vehicle, providing notices, or issuing a citation, among others. When the transaction is completed, the blockchain is updated to reflect the new information. Because of this, the transaction cannot be modified, and only those who have been granted access can view the results.

In a smart contract, there can be as many specifications as necessary to ensure that the task is accomplished correctly and that all parties are satisfied with the outcome. The terms of the agreement must be determined by the participants, who must agree on how transactions and their data will be represented on the blockchain, agree on the "if/when...then..." rules that will govern those transactions, explore all possible exceptions, and define a framework for resolving disputes.

Later, a developer can create the smart contract, however increasingly, firms that use blockchain for business are providing templates, web interfaces, and other online tools to make the process of creating smart contracts more convenient for their customers.

Simple smart contract in Solidity:

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  contract HelloWorld {  string public greet = "Hello Neuron"; } |
| --- |

When a contract is created, its constructor (a function declared with the constructor keyword) is executed once.

A constructor is optional. Only one constructor is allowed, which means overloading is not supported.

When to use a constructor?

After the constructor has executed, the final code of the contract is stored on the blockchain. This code includes all public and external functions and all functions that are reachable from there through function calls. The deployed code does not include the constructor code or internal functions only called from the constructor.

Internally, constructor arguments are passed ABI encoded after the code of the contract itself, but you do not have to care about this if you use web3.js.

If a contract wants to create another contract, the source code (and the binary) of the created contract has to be known to the creator. This means that cyclic creation dependencies are impossible.

**Solidity Constructor example:**

| // Solidity program to demonstrate // creating a constructor pragma solidity ^0.8.0;    // Creating a contract contract constructorExample {     // Declaring state variable  string str;     // Creating a constructor  // to set value of 'str'  constructor() public {   str = “Ineuron";   }     // Defining function to  // return the value of 'str'   function getValue(  ) public view returns (  string memory) {   return str;   }  } |
| --- |

**Scope Visibility**

Scope of local variables is limited to the function in which they are defined but State variables can have three types of scopes.

* Public − Public state variables can be accessed internally as well as via messages. For a public state variable, an automatic getter function is generated.
* Internal − Internal state variables can be accessed only internally from the current contract or contract deriving from it without using this.
* Private − Private state variables can be accessed only internally from the current contract; they are defined not in the derived contract from it.

In Solidity, local variables have a scope that is confined to the function in which they are defined, whereas state variables might have three different scopes.

Public - State variables that are accessible both internally and through messages are known as public state variables. An automatic getter method is built for a public state variable when the variable is declared.

Internal - Internal state variables can only be accessible internally from the current contract or a contract originating from it, and they cannot be accessed externally from any other contract.

Private -Private state variables can only be accessed internally from the current contract in which they are defined, not from the contract derived from it. They are not accessible from any other contract.

Global Variables are Special variables that exist in the global namespace used to get certain information about the blockchain.

Important:

Solidity is a statically typed programming language, which means that the type of the state or local variable must be defined at the time of declaration. Every defined variable has a default value that is determined by the type of the variable. There is no such thing as a "undefined" or a "null."

Variables in Solidity:

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  contract Variables {  // State variables are stored on the blockchain.  string public text = "Hello Neuron";  uint public num = 69;   function simpleFunc() public {  // Local variables are not saved to the blockchain.  uint i = 619;   // Here are some global variables  uint timestamp = block.timestamp;//Current block timestamp  address sender = msg.sender; // address of the caller  } } |
| --- |

**Functions**

A function is a collection of reusable code that may be invoked at any point in your program's code. This reduces the need to rewrite the same code over and again in different places. It assists programmers in the creation of modular code. Functions enable a programmer to break down a large software into a number of smaller and more manageable functions.

Solidity, like any other advanced programming language, provides all of the tools required to construct modular code using functions, just like any other advanced programming language would. This section teaches how to create your own functions in Solidity using the Solidity language.

Function Definition

It is necessary to declare a function before it may be used. Defining a function in Solidity is most commonly done by referencing it with the function keyword, followed by a unique function name, a list of parameters (which may or may not be empty), and a statement block enclosed by curly brackets.

Function Syntax

The basic syntax is shown here:

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

Simple Function Example:

| pragma solidity ^0.8.0;  contract Test {  function getResult() public view returns(uint){  uint a = 1; // local variable  uint b = 2;  uint result = a + b;  return result;  }} |
| --- |

**Function Visibility**

Solidity distinguishes between two types of function calls: external function calls that generate an actual EVM message call and internal function calls that do not. Furthermore, internal functions can be rendered inaccessible to derived contracts by modifying the contract's structure. There are four different types of visibility for functions as a result of this.

External

External functions are included in the contract interface, which means that they can be called from other contracts as well as from transactions. An external function f cannot be called from within another function (for example, f() does not work, but this.f() does work).

Public

Public functions are an element of the contract interface and can be called either internally or via message calls. They are called by the contract interface.

Internal

Internal functions can only be accessible from within the current contract or from contracts that are derived from it, and not from outside of it. They are not accessible from the outside world. The fact that they are not available to the outside world through the contract's ABI means that they can take parameters of internal types such as mappings or storage references.

Private

Private functions are similar to internal functions, with the exception that they are not visible in derived contracts.

**Solidity getter and setter functions**

You have to send a transaction if you want to write to or change a state variable.

On the other hand, you don't have to pay anything to read state variables.

A setter is a function in Solidity that changes the value of a variable (modifies the state of the contract). When you declare your function, you must tell it what parameters it will take.

In Solidity, a getter is a function that returns a value.

Simple getter and setter functions in Solidity:

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  contract SimpleStorage {  // State variable to store a number  uint public num;   // You need to send a transaction to write to a state variable.  function set(uint \_num) public {  num = \_num;  }   // You can read from a state variable without sending a transaction.  function get() public view returns (uint) {  return num;  } } |
| --- |

**Function Modifiers**

In Solidity, modifiers say what is happening in a way that is clear and easy to read. They are like the decorator pattern in Object-Oriented Programming.

A modifier is a piece of code that is added to a function to change its behavior.

For example, checking a condition automatically before running the function (this is mainly what they are used for).

Modifiers are helpful because they cut down on duplicate code. You can use the same modifier in more than one function if your smart contract checks for the same condition.

When should we use modifiers?

Modifiers are most often used to check a condition automatically before a function is run. If the function doesn't meet the requirement of the modifier, an exception is thrown, and the function stops running.

The wildcard symbol

A \_; is the merge wildcard symbol. It combines the code for the function with the code for the modifier where the \_; is.

In other words, where the special symbol \_; appears in the modifier's definition, the body of the function (to which the modifier is attached) will be added.

In order to work, a modifier must have the symbol \_; in its body. It must be done

The place where you write the \_; symbol will decide if the function has to be executed before, in between or after the modifier code.

you can place the \_; at the beginning, middle or the end of your modifier body.

In practice, (especially until you understand how modifiers work really well), the safest usage pattern is to place the \_; at the end. In this scenario, the modifier serves for a consistent validation check, so to check a condition upfront and then carry on. The code snippet below show show this as example:

You can put the \_; at the beginning, in the middle, or at the end of the body of your modifier.

In practice, the safest way to use the \_; is at the end. This is especially true until you really understand how modifiers work. In this case, the modifier is used as a consistent validation check, so that a condition can be checked up front and then the process continues. This is shown as an example in the code below:

Modifiers can be used to:

* Restrict access to a function
* Validate inputs of a function
* Guard against the reentrancy hack

Solidity implementation of function modifiers:

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  contract FunctionModifier {  // We will use these variables to demonstrate how to use  // modifiers.  address public owner;  uint public x = 10;  bool public locked;   constructor() {  // Set the transaction sender as the owner of the contract.  owner = msg.sender;  }   // Modifier to check that the caller is the owner of  // the contract.  modifier onlyOwner() {  require(msg.sender == owner, "Not owner");  // Underscore is a special character only used inside  // a function modifier and it tells Solidity to  // execute the rest of the code.  \_;  }   // Modifiers can take inputs. This modifier checks that the  // address passed in is not the zero address.  modifier validAddress(address \_addr) {  require(\_addr != address(0), "Not valid address");  \_;  }   function changeOwner(address \_newOwner) public onlyOwner validAddress(\_newOwner) {  owner = \_newOwner;  } |
| --- |

Returning multiple variables from functions:

There are several ways to return outputs from a function.The following code implements the various permutations and combinations while returning function outputs.

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  contract Function {  // Functions can return multiple values.  function returnMany()  public  pure  returns (  uint,  bool,  uint  )  {  return (1, true, 2);  }   // Return values can be named.  function named()  public  pure  returns (  uint x,  bool b,  uint y  )  {  return (1, true, 2);  }   // Return values can be assigned to their name.  // In this case the return statement can be omitted.  function assigned()  public  pure  returns (  uint x,  bool b,  uint y  )  {  x = 1;  b = true;  y = 2;  }   // Use destructuring assignment when calling another  // function that returns multiple values.  function destructuringAssignments()  public  pure  returns (  uint,  bool,  uint,  uint,  uint  )  {  (uint i, bool b, uint j) = returnMany();   // Values can be left out.  (uint x, , uint y) = (4, 5, 6);   return (i, b, j, x, y);  }   // Cannot use map for either input or output   // Can use array for input  function arrayInput(uint[] memory \_arr) public {}   // Can use array for output  uint[] public arr;   function arrayOutput() public view returns (uint[] memory) {  return arr;  } } |
| --- |

**Immutable state variables**

State variables can be marked immutable which causes them to be read-only, but assignable in the constructor. The value will be stored directly in the code. Once values are assigned to immutable variables in the constructor , they cannot be modified later on.

The following code will throw an error , since we are trying to modify the value of the variable ‘a’ using a setter function, which is not allowed since ‘a’ is an immutable variable.

**pragma solidity ^0.8.0;**

**contract TestImmutable {**

**uint256 public immutable a;**

**// This is a valid constructor**

**constructor (uint256 \_a) public {**

**a = \_a;**

**}**

**// This is invalid and will not compile**

**function setA (uint256 \_a) public {**

**a = \_a;**

**}**

**}**

It is important to know that the code that the compiler uses to create a contract will change the runtime code of the contract before it is returned. It will do this by replacing all references to immutables with the values that have been assigned to them. This is important if you want to compare the code made by the compiler and the code stored in the blockchain.

**Payable Functions**

When you write a smart contract, you have to make sure that money goes into and out of the contract. Payable takes care of this for you. Any Solidity function with a modifier Payable makes sure that the function can send and receive Ether. It can handle transactions where the Ether value is not zero, but it can't handle transactions where the Ether value is zero. Also, if you want a function to process transactions but don't include the payable keyword in them, the transaction will be automatically rejected. As an example, let's say you have a receive() function with the payable modifier. This means that this function can receive money in the contract. Now, let's say you have a send() function without the payable modifier. When you try to send money out of the contract, the transaction will be rejected.

In Solidity, fallback payable functions are also a big help. If someone sends money to your contract through a function that doesn't have the payable modifier, you can define that function to have a fallback payable function that makes sure the transaction will still go through. Because of this, it is a good idea to use some version of a function with the noname and payable modifiers. In this case, the name of the function is "noname," not "payable." "Payable" is the word that describes the function.

Function declaration with no name

| function () public payable {} |
| --- |

You can define a payable function using the following syntax:

| function receive() payable {} function send() payable {} |
| --- |

As you can see, the keyword payable is not a function; instead, it is a modifier. People sometimes mistake it for a function, which changes the meaning of the whole function and makes the code not work. As we said before about the "noname" function, if someone tries to call another function without the payable modifier, it acts as a fallback and sends the ether being sent to this "noname" function.

Solidity supports several methods of transferring ether between the contracts.

# 

# address.send(amount)

# Send was the first way that ether could be sent from contract to another (). There are two things you should think about while using send.

The first and most important feature is that there is a 2300 gas limit for a contract's fallback function of receiving ether. Also, this amount won't be enough to make more than one event.

The second important thing to keep in mind is that if send() fails (for example, because you ran out of gas), it returns false but does not throw an exception. So, every time send() is used, it should be inside of require. If you don't pay for gas, you won't be able to submit a transaction to the blockchain. Instead, all the changes will be rolled back.

1. address.transfer(amount)

The transfer method has the same limit of 2300 gas. During the process of making these features, however, the developers talked about adding a.gas() modifier, which changes the limit of the given gas.

Second, unlike the send() method, the transfer() method throws an exception when it doesn't work. So, you find out right when the transaction is tried to be executed that it failed. This will show up in your Ethereum wallet or metamask:

1. address.call.value(amount)()

The last and most personalized way to send ether is by using the call function.

If there is an error, the given function will still return false, so keep in mind how to use require().

Its main difference from the two functions that came before it is that you can set the gas limit with the .gas(gasLimit) modifier. It is necessary if the function that pays out the ether in the contract has a complicated logic that needs a lot of gas

Solidity Implementation of payable functions:

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract Payable {

// Payable address can receive Ether

address payable public owner;

// Payable constructor can receive Ether

constructor() payable {

owner = payable(msg.sender);

}

// Function to deposit Ether into this contract.

// Call this function along with some Ether.

// The balance of this contract will be automatically updated.

function deposit() public payable {}

// Call this function along with some Ether.

// The function will throw an error since this function is not payable.

function notPayable() public {}

// Function to withdraw all Ether from this contract.

function withdraw() public {

// get the amount of Ether stored in this contract

uint amount = address(this).balance;

// send all Ether to owner

// Owner can receive Ether since the address of owner is payable

(bool success, ) = owner.call{value: amount}("");

require(success, "Failed to send Ether");

}

function getBalance() public view returns(uint){

return address(this).balance;

}

// Function to transfer Ether from this contract to address from input

function deposit\_transfer(address payable \_to, uint \_amount) public payable{

// Note that "to" is declared as payable

\_to.transfer(\_amount);//2300 gas

}

function deposit\_send(address payable \_to, uint \_amount) public payable{

// Note that "to" is declared as payable

bool sent=\_to.send(\_amount);//2300 gas

require(sent,"failure to send ether");

}

function deposit\_call(address payable \_to, uint \_amount) public payable{

// Note that "to" is declared as payable

(bool sent,bytes memory data)=\_to.call{gas: 25000, value:\_amount}("");//set max gas limit as 25000 gas

require(sent,"failure to send ether");

}

}

Example 2:

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  contract ReceiveEther {  /\*  Which function is called, fallback() or receive()?   send Ether  |  msg.data is empty?  / \  yes no  / \ receive() exists? fallback()  / \  yes no  / \  receive() fallback()  \*/   // Function to receive Ether. msg.data must be empty  receive() external payable {}   // Fallback function is called when msg.data is not empty  fallback() external payable {}   function getBalance() public view returns (uint) {  return address(this).balance;  } }  contract SendEther {  function sendViaTransfer(address payable \_to) public payable {  // This function is no longer recommended for sending Ether.  \_to.transfer(msg.value);  }   function sendViaSend(address payable \_to) public payable {  // Send returns a boolean value indicating success or failure.  // This function is not recommended for sending Ether.  bool sent = \_to.send(msg.value);  require(sent, "Failed to send Ether");  }   function sendViaCall(address payable \_to) public payable {  // Call returns a boolean value indicating success or failure.  // This is the current recommended method to use.  (bool sent, bytes memory data) = \_to.call{value: msg.value}("");  require(sent, "Failed to send Ether");  } } |
| --- |