**Solidity Smart Contracts**

**Table of Contents**

* **Abstract Contracts**
* **Solidity Interfaces**
* **Solidity Libraries**

**Abstract Contracts**

Contracts that include at least one function but do not include its implementation might be referred to as abstract contracts. It is not possible to generate an instance of an abstract. Abstract contracts are typically utilized as base contracts, allowing child contracts to inherit and make use of the functions of the parent contract. Any derived contract that is inherited from the abstract contract should provide an implementation for the incomplete functions, and if the derived contract is also not implementing the incomplete functions, then that derived contract will also be marked as abstract. The abstract contract defines the structure of the contract, and any derived contract that is inherited from the abstract contract should provide an implementation for the incomplete functions. In Solidity, there is no abstract term that may be used to create an abstract contract. Instead, a contract is considered abstract if it contains functions that have not been implemented.

In the following illustration, an abstract contract is developed, and then it is inherited by a different contract that has completely implemented all of the abstract contract's functionalities. In the calling contract, an instance of an abstract contract is generated, and simultaneously, an object of the child contract is made. All of the abstract functions that are implemented in the child contract can be invoked, and they are accessed through the object of the child contract.

Abstract contract implementation in Solidity

| pragma solidity ^0.8.0;  contract absContract {   // Declaring functions  function getStr (  string \_strIn) public view returns(  string memory);  function setValue(uint \_in1, uint \_in2) public;  function add() public returns(uint); }    contract derived is absContract{   // Declaring private  // variables  uint private num1;  uint private num2;   // Defining functions inherited  // from abstract parent contract  function getStr (  string \_strIn) public view returns(  string memory){  return \_strIn;  }    function setValue (  uint \_in1, uint \_in2) public{  num1 = \_in1;  num2 = \_in2;  }  function add() public returns(uint){  return (num2 + num1);  }   }  // Caller contract contract call{   // Creating an instance of  // an abstract contract  absContract abs;    // Creating an object of  // child contract  function call(){  abs = new derivedContract();  }   // Calling functions inherited  // from abstract contract  function getValues(  ) public returns (uint){  abs.setValue(10, 16);  abs.getStr("Blockchain with Ineuron");  return abs.add();  }   } |
| --- |

**Solidity Interfaces**

The use of an interface keyword results in the creation of an abstract contract, which is also referred to as a pure abstract contract. Interfaces are identical to abstract contracts. The only thing that interfaces include are declarations of functions; this means that the functions contained within interfaces do not have any statements, nor do they have any definitions, nor do they have any state variables, nor do they have any constructors. Only the external type can be used for functions of the Interface. They are allowed to inherit from other interfaces, but not from other contracts. You can access enums and structs that are contained within an interface by using the dot notation that corresponds to the name of the interface.

The following example demonstrates how the contract childContract may be used to implement an interface by demonstrating how it can carry out each of the interface's functions.

Example 1

| pragma solidity 0.8.0;  // A simple interface interface simpleInterface{   // Functions having only  // declaration not definition  function getStr(  ) public view returns(string memory);  function setValue(  uint \_num1, uint \_num2) public;  function add(  ) public view returns(uint); }  // Contract that implements interface contract childContract is simpleInterface{   // Private variables  uint private num1;  uint private num2;   // Function definitions of functions  // declared inside an interface  function getStr(  ) public view returns(string memory){  return "Blockchain";  }    // Function to set the values  // of the private variables  function setValue(  uint \_num1, uint \_num2) public{  num1 = \_num1;  num2 = \_num2;  }    // Function to add 2 numbers  function add(  ) public view returns(uint){  return num1 + num2;  }   } contract call{    //Creating an object  InterfaceExample obj;   function call() public{  obj = new thisContract();  }    // Function to print string  // value and the sum value  function getValue(  ) public returns(uint){  obj.getStr;  obj.setValue(10, 16);  return obj.add();  } } |
| --- |

Example 2

**// SPDX-License-Identifier: MIT**

**pragma solidity ^0.8.0;**

**contract Counter {**

**uint public count;**

**function increment() external {**

**count += 1;**

**}**

**}**

**interface ICounter {**

**function count() external view returns (uint);**

**function increment() external;**

**}**

**contract MyContract {**

**function incrementCounter(address \_counter) external {**

**ICounter(\_counter).increment();**

**}**

**function getCount(address \_counter) external view returns (uint) {**

**return ICounter(\_counter).count();**

**}**

**}**

**Solidity Libraries**

The concept of libraries in Solidity is analogous to that of contracts in that both store reusable code. A library contains functions that are accessible to other contracts and can be called upon by those contracts. The gas cost can be reduced by deploying common code through the creation of a library. Only functions that are considered pure or view can be called from outside the library. This means that direct library function calls are only possible when the functions in question do not affect the state variables. Due to the presumption that it does not belong to any state, it cannot be destroyed. The library does not contain any state variables, it is unable to inherit any element, and it cannot be inherited by anything else.

To define a library contract, rather than using the term "generic contract," one should use the phrase "library." Because libraries do not have any storage, they are unable to keep state variables, fallback functions, or payment functions. Additionally, functions cannot be constructed within a library because the library cannot store ethers. It is only possible to use libraries for carrying out fundamental operations that are based on inputs and outputs. Libraries cannot be used to alter the state of the contract in any way. However, it is able to implement some data types like struct and enums, both of which are user-defined data types, as well as constant variables that are saved in a stack of Ethereum rather than in storage.

For the convenience of its users, Solidity comes with a number of built-in libraries. The following is a list of some of the libraries:

Modular network: This includes a large number of modular libraries that are quite helpful for implementation. Some examples of these libraries are ArrayUtils, Token, CrowdSale, Vesting, StringUtils, LinkedList, and Wallet, among others.

OpenZeppelin : Roles, MerkleProof, ECDSA, Math, Address, SafeERC20, ERC165Checker, SafeMath, and others that guard from overflow.

Dapp-bin: It was developed by Ethereum and provides a number of libraries that are both intriguing and useful. Some examples include DoublyLinkedList, StringUtils, and IterableMapping.

Solidity Library Implementation:

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  library SafeMath {  function add(uint x, uint y) internal pure returns (uint) {  uint z = x + y;  require(z >= x, "uint overflow");   return z;  } }  library Math {  function sqrt(uint y) internal pure returns (uint z) {  if (y > 3) {  z = y;  uint x = y / 2 + 1;  while (x < z) {  z = x;  x = (y / x + x) / 2;  }  } else if (y != 0) {  z = 1;  }  // else z = 0 (default value)  } }  contract TestSafeMath {  using SafeMath for uint;   uint public MAX\_UINT = 2\*\*256 - 1;   function testAdd(uint x, uint y) public pure returns (uint) {  return x.add(y);  }   function testSquareRoot(uint x) public pure returns (uint) {  return Math.sqrt(x);  } }  // Array function to delete element at index and reorganize the array // so that there are no gaps between the elements. library Array {  function remove(uint[] storage arr, uint index) public {  // Move the last element into the place to delete  require(arr.length > 0, "Can't remove from empty array");  arr[index] = arr[arr.length - 1];  arr.pop();  } }  contract TestArray {  using Array for uint[];   uint[] public arr;   function testArrayRemove() public {  for (uint i = 0; i < 3; i++) {  arr.push(i);  }   arr.remove(1);   assert(arr.length == 2);  assert(arr[0] == 0);  assert(arr[1] == 2);  } } |
| --- |