**Solidity Smart Contracts**

**Table of Contents**

* **Fallback functions**
* **View functions**
* **Pure functions**
* **Function overloading**
* **Function overriding**
* **Overload resolution and Argument matching**
* **Block and Transaction details**
* **Solidity Inheritance**
* **Single Inheritance**
* **Multilevel Inheritance**
* **Hierarchical Inheritance**
* **Multiple Inheritance**

**Fallback Functions**

If none of the other functions match the function identifier, the solidity fallback function is called. It is run if the call didn't get any data. Most of the time, a contract has only one unnamed task. It is run whenever the contract gets just Ether, with no data. To receive Ether and add it to the contract's total balance, the fallback function must be set to "payable."

Fallback is a function that takes no arguments and returns nothing back.

Fallback function is executed when:

* If a function that doesn't exist is called
* Ether is sent directly to a contract but receive() doesn't exist or msg.data isn't empty, fallback has a 2300 gas limit when called by transfer or send.

There can only be one fallback function in a contract. It is declared with fallback () external [payable]. No arguments can be passed to this function. It can't give anything back and can't be hidden from the outside world. If none of the functions in the opposite direction match the given function signature, it is run when the contract is called. It can be run even if there was no data at all and there is no receive Ether function. The fallback function is still used to get information. But if you want to also get Ether, you have to mark it as payable.

Like any other function, the fallback function can do complicated things as long as it is given enough gas. If a payable fallback function is also used instead of a receive function, it can only think that 2300 gas is available in the worst case.

Features of fallback function:

* It is called when the contract names a function that doesn't exist.
* Must be marked as external.
* It has no name.
* It has no arguments
* It can't return anything back.
* It's usually defined together with contracts.
* If the contract is not marked as payable, it will throw an exception if it gets just plain ether without any data.
* If a contract only gets plain Ether and no other information about the transaction, fallback functions are called.
* This choice for the default design is smart and helps protect users.
* But for our use case, it will be very important that our smart contract has a fallback function that lets it receive plain Ether.

Solidity Implementation of a Fallback Function:

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  contract Fallback {  event Log(uint gas);   // Fallback function must be declared as external.  fallback() external payable {  // send / transfer (forwards 2300 gas to this fallback function)  // call (forwards all of the gas)  emit Log(gasleft());  }   // Helper function to check the balance of this contract  function getBalance() public view returns (uint) {  return address(this).balance;  } }  contract SendToFallback {  function transferToFallback(address payable \_to) public payable {  \_to.transfer(msg.value);  }   function callFallback(address payable \_to) public payable {  (bool sent, ) = \_to.call{value: msg.value}("");  require(sent, "Failed to send Ether");  } } |
| --- |

**View functions**

The view functions are read-only functions, which ensures that state variables cannot be updated after they have been invoked by the user. The compiler will issue a warning if any of the following statements are found in view functions: modifying state variables, emitting events, creating other contracts, using the selfdestruct method, transferring ethers via calls, calling a function that is not 'view or pure', using low-level calls, and so on. The view function is used as the default get method.

View functions assure that they will not cause any changes to the current state. It is possible to declare a function as a view.

If any of the following statements are included in the function, they are deemed to be changing the state, and the compiler will issue a warning in such circumstances:

* Changing the values of state variables.
* Emitting events.
* Other contracts are being created.
* Selfdestruct is being used.
* Sending Ether over function calls.
* Calling any function that is not designated as view or pure.
* Making use of low-level calls.
* Using inline assembly with specific opcodes to accomplish this.

**Solidity view functions example 1**

| // SPDX-License-Identifier: MIT  pragma solidity ^0.8.0;   // Defining a contract contract Test {    // Declaring state   // variables  uint num1 = 2;   uint num2 = 4;    // Defining view function to   // calculate product and sum   // of 2 numbers  function getResult(  ) public view returns(  uint product, uint sum){  uint num1 = 10;  uint num2 = 16;  product = num1 \* num2;  sum = num1 + num2;   } } |
| --- |

**Solidity view functions example 2**

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.13;  contract ViewAndPure {  uint public x = 1;   // Promise not to modify the state.  function addToX(uint y) public view returns (uint) {  return x + y;  }  } |
| --- |

**Pure functions**

Pure functions do not read or modify state variables; instead, they return values that are derived solely from the parameters supplied to the function or from local variables that are contained in it. The compiler issues a warning in such circumstances if any of the statements that read the state variables, access the address or balance, access any global variable block or msg, call a function that is not pure, and so on, are present in pure functions.

Pure functions are those that do not read or modify the state of the system. It is possible to declare a function to be pure.

If any of the following statements are present in the function, they are regarded as reading the state, and the compiler will issue a warning in such circumstances:

* State variables are being read.
* Using the address(this).balance or <address>.balance functions.
* Using any of the special variables of block, tx, or msg (msg.sig and msg.data can be read).
* Calling any function that has not been declared as pure.
* Using inline assembly that contains specific opcodes.
* If an error occurs, pure functions can utilise the reverse() and require() procedures to undo any potential state changes that may have occurred.

**Solidity pure functions example 1**

| //SPDX-License-Identifier: MIT pragma solidity ^0.5.0;   // Defining a contract contract Test {    // Defining pure function to   // calculate product and sum  // of 2 numbers  function getResult(  ) public pure returns(  uint product, uint sum){  uint num1 = 2;   uint num2 = 4;  product = num1 \* num2;  sum = num1 + num2;   } } |
| --- |

**Solidity pure functions example 2**

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  contract ViewAndPure {  uint public x = 1;   // Promise not to modify the state.  function addToX(uint y) public view returns (uint) {  return x + y;  }   // Promise not to modify or read from the state.  function add(uint i, uint j) public pure returns (uint) {  return i + j;  } } |
| --- |

**Function overloading**

You can have numerous definitions for the same function name in the same scope if the function name is different. The types of arguments in the argument list, as well as the number of arguments in the argument list, must be different for each definition of the function. It is not possible to overload function declarations that differ simply in the type of return they return.

In Solidity, the idea of function overloading is demonstrated in the following example.

| pragma solidity ^0.8.0;  contract Test {  function getSum(uint x, uint y) public pure returns(uint){   return a + b;  }  function getSum(uint x, uint y, uint z) 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);  } } |
| --- |

**Function overriding**

A virtual function is one that allows an inherited contract to override the operation of a function that is defined as such.

The function that overrides the base function should be labeled as an override in Solidity in order to distinguish it from the base function.

| pragma solidity ^0.8.0;  contract Base1 {  function foo() virtual public {} }  contract Base2 {  function foo() virtual public {} }  contract Inherited is Base1, Base2 {  // Derives from multiple bases defining foo(), so we must explicitly  // override it  function foo() public override(Base1, Base2) {} |
| --- |

**Solidity Events**

In Solidity, when an event is emitted, it becomes an inheritable part of the contract and keeps the arguments that were supplied in the transaction logs. For as long as the contract is present on the blockchain, these logs are stored on the blockchain and can be accessed by using the contract's address to navigate to them.

Event generation is not accessible from within contracts, including the contract that generated and emitted the event in the first place. Generally, events are used to tell the caller program about the current state of the contract, which is accomplished through the usage of the logging capability provided by the Ethereum Virtual Machine.

Apps that may be used to execute the dependent logic are notified of changes to the contracts and applications that can be used to notify applications of changes to the contracts.

The **event** keyword can be used to declare a specific event.

Events are defined in the contracts as global events that can be called from within the contract's functionalities. Using the event keyword, followed by an identifier and the parameter list, and ending with a semicolon, you can declare events in your application.

The parameter values are used to log information or to execute conditional logic depending on the situation. The information and values associated with it are recorded as part of the transactions that take place within the block. There is no requirement to provide variables; only data types are required instead.

An event can be called from any method by referencing it by name and passing it the necessary parameters to complete the task.

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  contract Event {  // Event declaration  // Up to 3 parameters can be indexed.  // Indexed parameters helps you filter the logs by the indexed parameter  event Log(address indexed sender, string message);  event AnotherLog();   function test() public {  emit Log(msg.sender, "Hello Neuron!");  emit Log(msg.sender, "Welcome to Tech Neuron!");  emit AnotherLog();  } } |
| --- |

**Solidity Inheritance**

One of the most fundamental characteristics of the object-oriented programming language is the concept of inheritance. Using it to enhance the functionality of a programme, it helps to isolate the code, eliminates dependencies, and increases the re-usability of the already existing code. Solidity allows for the inheritance of smart contracts between themselves, with many contracts being able to be inherited into a single contract. The contract from which other contracts inherit features is referred to as the base contract, and the contract from which the features are inherited is referred to as the derived contract. They are referred to as "parent-child contracts" to put it simply. Unlike other programming languages, inheritance in Solidity is restricted to public and internal modifiers only. Some of the most notable aspects of Solidity are as follows:

* In addition to state variables and internal methods, a derived contract has access to all other non-private members of the contract. However, it is not permitted to do so.
* Overriding of functions is permitted as long as the function signature remains unchanged.
* The compilation will fail if the output parameters are different from the input values.
* The function of a super contract can be accessed through the use of a super keyword or the name of the super contract.
* When there are numerous inheritances, function calls made with the super keyword give preference to the contracts that are the most derived.

**Single Inheritance**

In single or single level inheritance, the functions and variables of a base contract are passed on to only one derived contract, which is referred to as a single level inheritance.

In the following example, the contract parent is inherited by the contract child in order to show Single Inheritance of Contracts.

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  // Defining contract contract parent{   // Declaring internal  // state variable  uint internal sum;    // Defining external function  // to set value of internal  // state variable sum  function setValue() external {  uint a = 5;  uint b = 10;  sum = a + b;  } }  // Defining child contract contract child is parent{    // Defining external function  // to return value of  // internal state variable sum  function getValue(  ) external view returns(uint) {  return sum;  } }  // Defining calling contract contract caller {   // Creating child contract object  child cc = new child();    // Defining function to call  // setValue and getValue functions  function testInheritance(  ) public returns (uint) {  cc.setValue();  return cc.getValue();  } } |
| --- |

**Multilevel Inheritance**

It is quite similar to single inheritance, with the exception that there are different levels of the relationship between the parent and the kid in the case of multiple inheritance. The child contract that is generated from a parent contract likewise serves as a parent for the contract that is derived from the parent contract.

Multi-level inheritance is demonstrated in the following example, in which contract A is inherited by contract B, and contract B is inherited by contract C.

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  // Defining parent contract A contract A {   // Declaring state variables  string internal x;  string a = "Oneneuron" ;  string b = "Kids Neuron";   // Defining external function  // to return concatenated string  function getA() external{  x = string(abi.encodePacked(a, b));  } }  // Defining child contract B // inheriting parent contract A contract B is A {   // Declaring state variables  // of child contract B  string public y;  string c = "Tech Neuron";   // Defining external function to  // return concatenated string  function getB() external payable returns(  string memory){  y = string(abi.encodePacked(x, c));  } }  // Defining child contract C // inheriting parent contract A contract C is B {    // Defining external function  // returning concatenated string  // generated in child contract B  function getC() external view returns(  string memory){  return y;  } }  // Defining calling contract contract caller {   // Creating object of child C  C cc = new C();   // Defining public function to  // return final concatenated string  function testInheritance(  ) public returns (  string memory) {  cc.getA();  cc.getB();  return cc.getC();  } } |
| --- |

**Hierarchical Inheritance**

An inheritance structure in which a parent contract has more than one child contract is known as hierarchical inheritance. It is most commonly used when a similar functionality needs to be used in multiple locations at the same time.

In the following example, contract A is inherited by contract B, and contract A is inherited by contract C, exhibiting Hierarchical Inheritance in the process of inheritance.

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  // Defining parent contract A contract A {   // Declaring internal  // state variable  string internal x;    // Defining external function  // to set value of  // internal state variable  function getA() external {  x = “Ineuron";  }    // Declaring internal  // state variable  uint internal sum;    // Defining external function  // to set the value of  // internal state variable sum  function setA() external {  uint a = 10;  uint b = 20;  sum = a + b;  } }  // Defining child contract B // inheriting parent contract A contract B is A {   // Defining external function to  // return state variable x  function getAstring(  ) external view returns(string memory){  return x;  }   } // Defining child contract C // inheriting parent contract A contract C is A {   // Defining external function to  // return state variable sum  function getAValue(  ) external view returns(uint){  return sum;  } }  // Defining calling contract contract caller {   // Creating object of contract B   B contractB = new B();   // Creating object of contract C  C contractC = new C();    // Defining public function to  // return values of state variables  // x and sum  function testInheritance(  ) public returns (  string memory, uint) {  contractB.getA();  contractC.setA();  return (  contractB.getAstring(), contractC.getAValue());  } } |
| --- |
|  |

**Multiple Inheritance**

One contract can be inherited from multiple contracts under Multiple Inheritance. A parent contract may include more than one child, whereas a child contract may include more than one parent.

A contract A is inherited by another contract B, and contract C is inherited by both contracts A and B, proving Multiple Inheritance in this case.

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  // Defining contract A contract A {   // Declaring internal  // state variable  string internal x;   // Defining external function  // to set value of  // internal state variable x  function setA() external {  x = "Blockchain";  } }  // Defining contract B contract B {  // Declaring internal // state variable  uint internal mul;   // Defining external function  // to set value of internal  // state variable pow  function setB() external {  uint a = 2;  uint b = 20;  mul = a \*\* b;    } }  // Defining child contract C // inheriting parent contract // A and B contract C is A, B {  // Defining external function // to return state variable x function getStr( ) external returns(string memory) {  return x;  }   // Defining external function  // to return state variable pow  function getMul(  ) external returns(uint) {  return mul;  } }  // Defining calling contract contract caller {   // Creating object of contract C  C contractC = new C();   // Defining public function to  // return values from functions // getStr and getPow  function testInheritance(  ) public returns(string memory, uint) {  contractC.setA();  contractC.setB();  return (  contractC.getStr(), contractC.getMul());  } } |
| --- |

**Inheritance with function overriding**

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  /\* Graph of inheritance  A  / \  B C  / \ / F D,E  \*/  contract A {  function foo() public pure virtual returns (string memory) {  return "A";  } }  // Contracts inherit other contracts by using the keyword 'is'. contract B is A {  // Override A.foo()  function foo() public pure virtual override returns (string memory) {  return "B";  } }  contract C is A {  // Override A.foo()  function foo() public pure virtual override returns (string memory) {  return "C";  } }  // Contracts can inherit from multiple parent contracts. // When a function is called that is defined multiple times in // different contracts, parent contracts are searched from // right to left, and in depth-first manner.  contract D is B, C {  // D.foo() returns "C"  // since C is the right most parent contract with function foo()  function foo() public pure override(B, C) returns (string memory) {  return super.foo();  } }  contract E is C, B {  // E.foo() returns "B"  // since B is the right most parent contract with function foo()  function foo() public pure override(C, B) returns (string memory) {  return super.foo();  } }  // Inheritance must be ordered from "most base-like" to "most derived". // Swapping the order of A and B will throw a compilation error. contract F is A, B {  function foo() public pure override(A, B) returns (string memory) {  return super.foo();  } } |
| --- |