Solidity Advance

Part - 5

Contents

Contract and Objects

Inheritance

Abstract Class

Interface

Polymorphism

Library

Payable Address

Payable Function

Error and Revert

Events

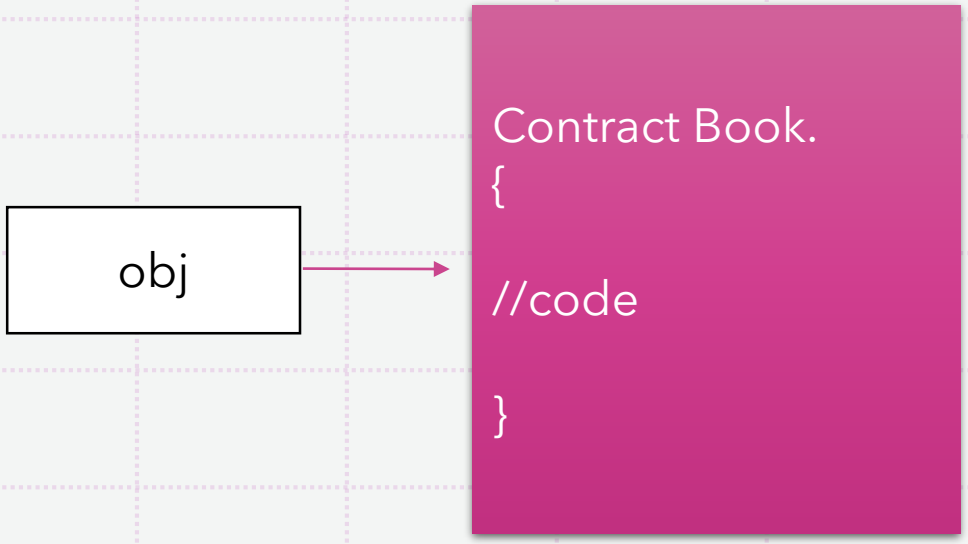
Much more

Contract and Object

- Solidity Contracts are like a class in any other object-oriented programming language.
- A class is a description of an object's property and behavior.
- Object is a real world entity.

Contract and Object

Book obj = new Book( );



Note - Constructor of a contract is called as soon as its contract is created.

```
contract Book{
uint length;
uint breadth;
uint height;

function setDimension(uint _length,uint _breadth,uint _height) public
{length = _length;breadth=_breadth;height=_height;}

function getDimension() public view  returns (uint,uint,uint)
{ return (length,breadth,height);}
}
contract D {
Book obj= new Book();
function getInstance() public view returns(Book){
    return obj}
function writeDimension(uint _length,uint _breadth, uint _height) public {
    obj.setDimension(_length, _breadth, _height);}
function readDimension() public view returns(uint,uint,uint){
    return(obj.getDimension());}
}
```

Importing Contract

- Import "contract name with location";

```
contract Book{
uint length;
uint breadth;
uint height;

function setDimension(uint _length,uint _breadth,uint
_height) public
{length = _length;
breadth=_breadth;
height=_height;
}

function getDimension() public view  returns
(uint,uint,uint)
{ return (length,breadth,height);
}
}
```

C.sol

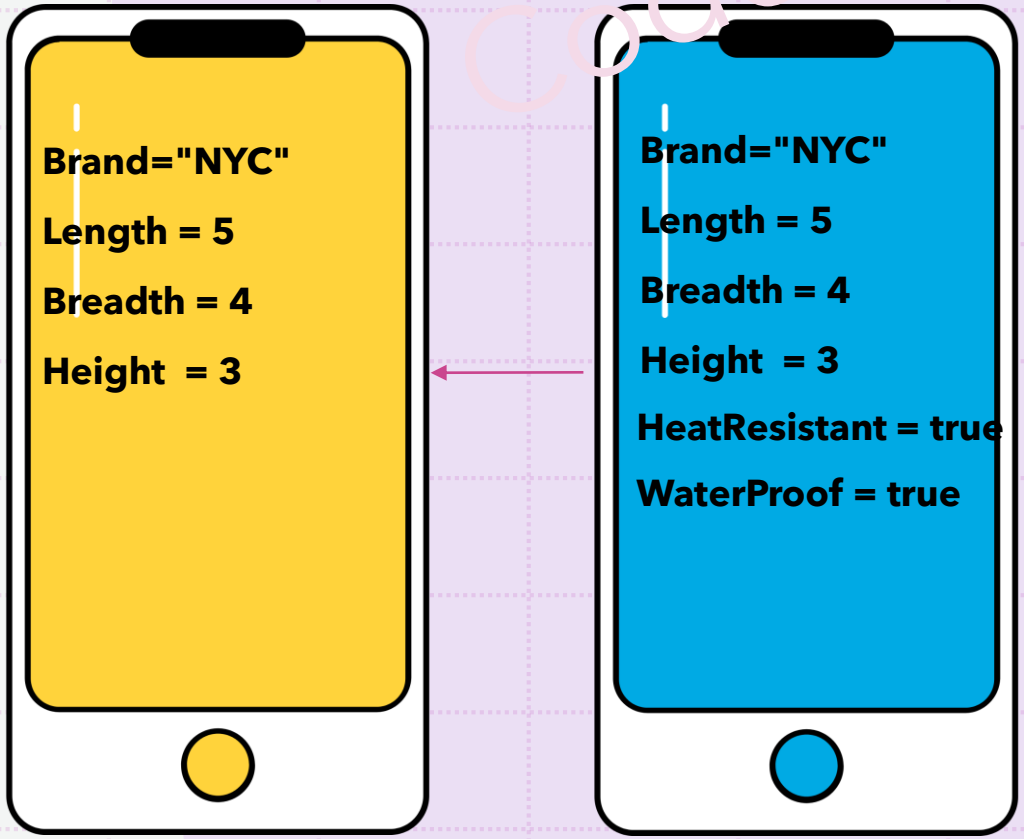
```
import "./C.sol";
contract D {
Book obj= new Book();

function getInstance() public view returns(Book){
    return obj;
}
function writeDimension(uint _length,uint _breadth, uint _height) public
{
    obj.setDimension(_length, _breadth, _height);
}
function readDimension() public view returns(uint,uint,uint){
    return(obj.getDimension());
}
}
```

D.sol

Inheritance

1. Inheritance in Solidity is **the procedure in which one contract inherits the attributes and methods of another contract**.



```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Car {
 uint public wheels=4;
 uint public doors=4;
 string public brandName="CTE";
 uint public headlight=2;
 bool public safetyBag=true;
}
contract superCar is Car{
   uint public speed=400;
   uint public modelNumber=121;
   string public modelName="Texxo";
}
```

Abstract Contracts

- **Abstract contracts** are contracts that can have functions without its implementation.
- To make a contract **abstract** you have to use abstract keyword.
- Function without implementation must contain **virtual** keyword.
- The abstract contract defines the structure of the contract and any derived contract inherited from it should provide an implementation for the incomplete functions. And that functions should contain **override** keyword.
- And if the **derived contract** is also not implementing the incomplete functions then that derived contract will also be **marked as abstract**.
- Contract marked as **abstract** contract cannot be deployed.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

abstract contract Parent {
 string public str;
 address public manager;
 constructor(){
    str="Hello World";
    manager=msg.sender;
 }
 function setter(string memory _str) public virtual;
}
contract Child is Parent{
   uint public x;
   function setter(string memory _str) public override {
      str=_str;
   }
}
```

Interface

- An interface is an agreement or a contract between itself and any contract that implements it.
- **Interfaces restrictions** -
  - They can only inherit from other interfaces but not from other contracts.
  - They cannot declare state variables.
  - They cannot declare constructor.
  - Functions can be declared but not implemented. All declared functions must be **external**.

Interface

```
interface Calculator {
   function getResult() external view returns(uint);
}

contract A is Calculator{
    function getResult() external view returns(uint){

    }
}
```

Polymorphism

- Polymorphism is one of the core concepts of object-oriented programming (OOP) and **describes situations in which something occurs in several different forms**.

```
contract Poly{
    function add(uint a,uint b) public pure returns(uint){
        return a+b;
    }
    function add(uint a,uint b,uint c) public pure returns(uint){
        return a+b+c;
    }
     function add(string memory a,uint b) public pure returns(uint){

    }
}
```

Error and Revert

- Errors allow you to provide more information to the caller about why a condition or operation failed.

- Errors are used together with the revert statement.

- The revert statement unconditionally aborts and reverts all changes similar to the require function, but it also allows you to provide the name of an error and additional data which will be supplied to the caller (and eventually to the front-end application or block explorer) so that a failure can more easily be debugged or reacted upon.

Error and Revert

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.17;

contract A {
    error InsufficientBalance(uint balance, uint withdrawAmount);

    function test(uint _withdrawAmount) public view {
        uint bal = address(this).balance;
        if (bal < _withdrawAmount) {
            revert InsufficientBalance(bal,_withdrawAmount);
        }
    }
}
```

Events

Amazon frontend

Pay Button

Event

Smart Contract

Events

- Events are inheritable members of contracts.
- When you call them, they cause the arguments to be stored in the transaction's log - a special data structure in the blockchain.
- These logs are associated with the address of the contract, are incorporated into the blockchain, and stay there as long as a block is accessible.
- The Log and its event data is not accessible from within contracts (not even from the contract that created them).

Where events get stored ?

Logs file

Event Advantages

- User get notify
- Owner of the SC get notify
- Low cost storage

```
contract sample{

  event etherTransfer(address receiver,uint balance);
  function sendEther(address _user) public payable{
    require(msg.value>=2 ether,"Value is less than 2 ether");
     payable(_user).transfer(msg.value);
     emit etherTransfer(_user,msg.value);
  }
}

```

Library

- Libraries in solidity are similar to contracts that contain reusable codes.
- Functions of the library can be called directly when they do not modify the state variables i.e. only pure and view functions can be called from outside of the library.
- **It cannot be destroyed** because it is assumed as stateless.
- The library does not have state variables, it cannot inherit any element and cannot be inherited.

```
//SPDX-License-Identifier: MIT
pragma solidity >=0.5.0 <0.9.0;


library Addition{

    function add(uint a,uint b) public pure returns(uint){
        return a+b;
    }
}


contract demo {

  function addition(uint a,uint b) public pure returns(uint){
      return Addition.add(a,b);
  }



}
```

Notes

Contract and Object

Solidity is a programming language used for developing smart contracts on the Ethereum blockchain. In Solidity, a contract is a collection of code (functions) and data that is stored on the Ethereum blockchain. Contracts can be used to represent digital assets, such as tokens or certificates, or to execute a predefined set of rules.

In Solidity, objects are instances of contract types. They are created when a contract is deployed to the Ethereum blockchain. Each object has its own set of data and can execute the functions defined within the contract. When an object executes a function, it can modify its own data and/or the data of other objects on the blockchain.

Solidity contracts and objects work together to define the behavior of decentralized applications (DApps) on the Ethereum blockchain. Contracts define the rules and functions of the DApp, while objects represent the actual instances of the DApp that are executed on the blockchain.

Instance of contract

In Solidity, you can create instances of contracts just like you would create objects in an object-oriented programming language.

When you deploy a contract to the Ethereum blockchain, it creates a new instance of that contract, which has its own state and behavior. You can interact with the contract instance by sending transactions or making calls to its public functions.

Here's an example of how to create a contract instance in Solidity:

```
pragma solidity ^0.8.0;

contract MyContract {
    uint256 public myValue;

    constructor(uint256 initialValue) {
        myValue = initialValue;
    }

    function setValue(uint256 newValue) public {
        myValue = newValue;
    }
}

contract MyContractFactory {
    function createMyContract(uint256 initialValue) public returns (MyContract) {
        return new MyContract(initialValue);
    }
}
```

Instance of contract

In this example, we have two contracts: `MyContract` and `MyContractFactory`. `MyContract` has a public state variable `myValue`, and a function `setValue` that allows you to update the value of `myValue`. The `constructor` function sets the initial value of `myValue` when the contract is created.

`MyContractFactory` has a function `createMyContract` that creates a new instance of `MyContract` and returns it. When you call `createMyContract` on `MyContractFactory`, it deploys a new instance of `MyContract` to the blockchain with the initial value specified in the function call.

You can interact with the contract instance returned by `createMyContract` just like you would with any other contract instance. For example:

```
MyContractFactory factory = new MyContractFactory();
MyContract myContract = factory.createMyContract(42);
uint256 value = myContract.myValue(); // returns 42
myContract.setValue(100);
value = myContract.myValue(); // returns 100
```

In this example, we create an instance of `MyContractFactory`, then use it to create a new instance of `MyContract` with an initial value of 42. We then call `myValue` to retrieve the value of `myValue`, which should be 42. We then call `setValue` to update the value of `myValue` to 100, and call `myValue` again to verify that the value was updated.

Inheritance

Inheritance is a mechanism in Solidity that allows a new contract to be based on an existing contract, inheriting all of its properties and methods. This can help to make Solidity code more modular, as it allows for common functionality to be defined in a base contract and then reused in multiple derived contracts.

To define a new contract that inherits from an existing contract, the `is` keyword is used, followed by the name of the base contract. For example:

```
pragma solidity ^0.8.0;

contract Animal {
    string public species;

    constructor(string memory _species) {
        species = _species;
    }

    function speak() public pure virtual returns (string memory) {
        return "Animal speaks!";
    }
}

contract Dog is Animal {
    constructor() Animal("Canine") {}

    function speak() public pure override returns (string memory) {
        return "Woof!";
    }
}
```

Inheritance

In this example, the `Dog` contract inherits from the `Animal` contract using the `is` keyword. The `Dog` contract adds its own constructor that calls the `Animal` constructor with the argument "Canine". It also overrides the `speak` function defined in the `Animal` contract to return "Woof!" instead of the generic animal sound.

Inheritance can be used to create hierarchies of contracts, where more specific contracts inherit from more general contracts. This can help to reduce code duplication and increase code reuse, as well as make the code easier to understand and maintain.

Why Interface?

The main use of interfaces in Solidity is to provide a way for contracts to interact with each other in a standardized and secure manner.

Interfaces are used to define the functions that can be called on a contract, but they do not provide any implementation details for those functions. This allows other contracts to interact with the contract without knowing the implementation details, making the contract more secure and less prone to errors.

In addition, interfaces can be used to abstract the implementation details of a contract, making it easier to upgrade or replace the implementation without affecting the contracts that rely on it.

Another use of interfaces is to provide a way for contracts to interact with external systems, such as other blockchains or off-chain data sources. By defining a standardized interface for these systems, contracts can interact with them in a secure and consistent way.

Overall, interfaces provide a way to create modular and interoperable contracts, which can help to reduce the complexity and increase the security of decentralized applications.

Abstract Contract

Solidity is a programming language used for developing smart contracts on the Ethereum blockchain. In Solidity, a contract is a collection of code (functions) and data that is stored on the Ethereum blockchain. Contracts can be used to represent digital assets, such as tokens or certificates, or to execute a predefined set of rules.

In Solidity, objects are instances of contract types. They are created when a contract is deployed to the Ethereum blockchain. Each object has its own set of data and can execute the functions defined within the contract. When an object executes a function, it can modify its own data and/or the data of other objects on the blockchain.

Solidity contracts and objects work together to define the behavior of decentralized applications (DApps) on the Ethereum blockchain. Contracts define the rules and functions of the DApp, while objects represent the actual instances of the DApp that are executed on the blockchain.

Events

In Solidity, an event is a way for a contract to notify the outside world when something happens. Events are essentially messages that the contract emits, and they can be subscribed to by external parties such as user interfaces, other contracts, or even off-chain systems.

Here's an example of an event in Solidity:

```
pragma solidity ^0.8.0;

contract MyContract {
    event NewValueSet(uint256 value);

    function setValue(uint256 newValue) public {
        emit NewValueSet(newValue);
    }
}
```

Events

In this example, `NewValueSet` is an event that is emitted by the `MyContract` contract when the `setValue` function is called. The event takes a single argument, `value`, which is an unsigned integer of 256 bits.

To listen for this event, an external party could use the Web3.js library or a similar tool to subscribe to events emitted by the `MyContract` contract. For example:

```
myContractInstance.events.NewValueSet(function(error, result) {
    if (!error) {
        console.log("New value set:", result.returnValues.value);
    }
});
```

This code would listen for `NewValueSet` events emitted by `myContractInstance`, and log a message to the console whenever a new value is set.

Events are commonly used in Solidity to notify external parties of important state changes in a contract, such as the creation of a new user account, the completion of a transaction, or the occurrence of an error. By emitting events, a contract can provide transparency and visibility into its internal state, and allow external parties to react to those changes in a timely and effective manner.

Events

In this example, `NewValueSet` is an event that is emitted by the `MyContract` contract when the `setValue` function is called. The event takes a single argument, `value`, which is an unsigned integer of 256 bits.

To listen for this event, an external party could use the Web3.js library or a similar tool to subscribe to events emitted by the `MyContract` contract. For example:

```
myContractInstance.events.NewValueSet(function(error, result) {
    if (!error) {
        console.log("New value set:", result.returnValues.value);
    }
});
```

This code would listen for `NewValueSet` events emitted by `myContractInstance`, and log a message to the console whenever a new value is set.

Events are commonly used in Solidity to notify external parties of important state changes in a contract, such as the creation of a new user account, the completion of a transaction, or the occurrence of an error. By emitting events, a contract can provide transparency and visibility into its internal state, and allow external parties to react to those changes in a timely and effective manner.

Polymorphism

Solidity is a programming language used for developing smart contracts on the Ethereum blockchain. In Solidity, a contract is a collection of code (functions) and data that is stored on the Ethereum blockchain. Contracts can be used to represent digital assets, such as tokens or certificates, or to execute a predefined set of rules.

In Solidity, objects are instances of contract types. They are created when a contract is deployed to the Ethereum blockchain. Each object has its own set of data and can execute the functions defined within the contract. When an object executes a function, it can modify its own data and/or the data of other objects on the blockchain.

Solidity contracts and objects work together to define the behavior of decentralized applications (DApps) on the Ethereum blockchain. Contracts define the rules and functions of the DApp, while objects represent the actual instances of the DApp that are executed on the blockchain.

Library are stateless?

Libraries in Solidity are typically stateless because they are designed to provide a collection of reusable functions that can be called by other contracts without the need to modify the state of the library or the calling contract.

When a library is called by a contract, the library's code is executed in the context of the calling contract. This means that the library has access to the state of the calling contract, but it cannot modify that state directly. Instead, any modifications to the state must be performed by the calling contract itself.

By making libraries stateless, Solidity ensures that they can be reused by multiple contracts without the risk of introducing unwanted side effects or unexpected interactions between contracts. Stateless libraries are also more efficient and easier to test, because they do not rely on external state or dependencies that can change over time.

That being said, it is possible to create stateful libraries in Solidity by using global variables or storage pointers. However, this is generally discouraged, because it can make the library less predictable and more difficult to use and test. In general, it's best to keep Solidity libraries stateless and focused on providing a collection of reusable functions that can be called by other contracts.

Thank you

Instagram - @codeeater21

LinkedIn - www.linkedin.com/in/kshitijWeb3

Twitter - @KshitijWeb3