LAB 16-08-2023

Visibility

Introduction

In Solidity smart contract programming, function visibility plays a crucial role in determining the accessibility of functions within a contract and across different contracts.

By defining the visibility of functions, developers can control how functions can be interacted with, ensuring proper usage, and preventing potential security vulnerabilities.

Additionally, function visibility interacts with contract inheritance, enabling child contracts to inherit and utilize functions from their parent contracts based on their visibility settings.

Understanding Function Visibility

Function visibility in Solidity defines the scope of accessibility for a function.

It specifies whether the function can be accessed within the same contract, inherited contracts, or external contracts.

Solidity provides four visibility modifiers for functions: **public**, **external**, **internal**, and **private**.

These modifiers determine the level of access to the function from various contexts.

The choice of function visibility modifier depends on the desired level of accessibility and security requirements. It is important to carefully consider which visibility modifier to use for each function to ensure proper functionality and prevent unauthorized access or unintended interactions.

Function Visibility and Inheritance

Inheritance is a fundamental feature in Solidity that allows child contracts to inherit properties and functions from their parent contracts. Function visibility plays a significant role in determining which functions are accessible by child contracts.

When a child contract inherits from a parent contract, it inherits the functions defined in the parent contract along with their visibility settings. The visibility of a function in the parent contract affects the visibility of that function in the child contract.

Visibility Modifiers

Solidity provides **four** visibility modifiers as following:

1) Public Visibility

- **Public** functions can be accessed by all parties within and outside the contract. This includes the main contract, its derived contracts, and any other third party contracts.
- The inherited function will retain its public visibility in the child contract.
- If visibility of a **function** has not been declared then **by default** it will be **public.**
- A **state variable** with the public modifier can be accessed by any contract in the application.

```
function function_name (parameter list) public returns (return_type)
```

Example

```
contract Public{

    string public publicmsg = "Public Variable is Called";

    function getpublic() public view returns(string memory){

        return publicmsg;
    }
}
```

Task

- Observe and deploy the above given ‘Public’ **contract**.
- Create a **child contract** of ‘Public’ named ‘Public2’. Deploy it and see which of the **state variables** and **functions** were inherited from the **parent contract.**

Task

- Create a Solidity **contract** named ‘PublicVisibility’. Inside it, declare a **public** state variable ‘publicData’ of type **uint**.
- Create a **child contract** named ‘PublicChild’. In it, implement a **public** function ‘getPublicData()’ that returns the value of ‘publicData’.

2) Private Visibility

- **Private** functions are restricted to the contract in which they are defined in.
- If a function in the parent contract has private visibility, it cannot be accessed or inherited by the child contract.
- If a **state variable** is private, only the main contract in which they were declared can call them.
- Setting the visibility of a function or variable to private does not make it invisible on the blockchain. It simply restricts its access to functions within the contract.

```
function function_name (parameter list) private returns (return_type)
```

Example

```
contract Private{

    string private privatemsg = "Private Variable is Called";

    function getprivate() private view returns(string memory){

        return privatemsg;
    }

    function getprivatevar() public view returns(string memory){

        return privatemsg;
    }

    function getprivatefunc() public view returns(string memory){

        //You are calling 'getprivate()' function here
        //It returns a string value which is then returned by the
        //'getprivatefunc()' function
        return getprivate();
    }
}
```

Task

- Observe and deploy the above given ‘Private’ **contract**. Run each of its **functions** available to you and observe what they do.
- Create a **child contract** of ‘Private’ named ‘Private2’. Deploy it and observe which of the **state variables** and **functions** were inherited from the **parent contract.**
- In the ‘Private2’ **contract**. Create a function that modifies the value of the ‘privatemsg’ variable in the parent contract. Is it possible?
- In the ‘Private’ contract, replace the **private** visibility of the ‘privatemsg’ variable with **public.** Deploy the ‘Private2’ contract again and observe what changes you get.

Task

- Create a contract **"Bank"** with a private state variable **"balance"** and a public function **"deposit"** that adds funds to the balance.
- Implement another contract **"BankUser"** (child contract of **“Bank”**) that interacts with the **"Bank"** contract by calling the **"deposit"** function.
- Attempt to create a ‘get()’ function for **"balance**" state variable in the **"BankUser"** contract. Is it possible? If not why?

3) Internal Visibility

- Functions declared with the **internal** keyword are only accessible within the contract in which they were declared.
- If a function in the parent contract has internal visibility, it can only be accessed by the main contract and any of the child contracts.
- The inherited function will retain its internal visibility, allowing the child contract to access and use it.
- If visibility of a **state variable** has not been declared then **by default** it will be **internal.**

- **State variables** declared with the internal modifier can only be accessed within the contract in which they were defined and by a derived contract.

```
function function_name (parameter list) internal returns (return_type)
```

Example

```
contract Internal{

    string internal internalmsg = "Internal Variable is Called";

    function getinternal() internal view returns(string memory){

        return internalmsg;
    }

    function getinternalvar() public view returns(string memory){

        return internalmsg;
    }

    function getinternalfunc() public view returns(string memory){

        return getinternal();
    }
}
```

Task

- Observe and deploy the above given ‘Internal’ **contract**. Run each of its **functions** available to you and observe what they do.
- Create a **child contract** of ‘Internal’ named ‘Internal2’. Deploy it and observe which of the **state variables** and **functions** were inherited from the **parent contract.**
- In the ‘Internal2’ **contract**. Create a function that modifies the value of the ‘internalmsg’ variable in the parent contract. Is it possible?
- In the ‘Internal’ contract, replace the **internal** visibility of the ‘internalmsg’ variable with **public.** Deploy the ‘Internal2’ contract again and observe what changes you get.

Task

- Design a Solidity **contract** named ‘FamilyFortune’. Declare an **internal** state variable ‘inheritance’ of type **uint** to represent a family inheritance.
- Implement an **internal** function ‘addInheritance(**uint** amount)’ that allows family members to contribute to the inheritance pool.
- Create a **child contract** named ‘Heir’ that **inherits** from ‘FamilyFortune’.
- Inside ‘Heir’, implement a **public** function that **calls** the inherited ‘addInheritance()’ function, allowing an heir to add to the family fortune.

Hint:

The **public** function in ‘Heir’ contract:

```
function function_name(uint amount) visibility{

    Name_of_function_to_be_called(uint amount);
}
```

4) External Visibility

- **External** functions can only be called from outside the contract in which they were declared.
- If a function in the parent contract has **external** visibility, it can be accessed only by external contracts (Third-Party contracts), not by the child contract itself.
- The inherited function in the child contract will also have external visibility.
- **State variables cannot** have external visibility.

```
function function_name (parameter list) external returns (return_type)
```

Example

```
contract External{

    // string external externalmsg = "External Variable is Called";

    function getexternal() external pure returns(string memory){
        return "External Function";
    }

    // function getexternalfunc() public view returns(string memory){

    //     return getexternal();
    // }
}
```

Task

- Observe and deploy the above given ‘External’ **contract**. Run each of its **functions** available to you and observe what they do.
- Uncomment the ‘externalmsg’ **variable** and ‘getexternalfunc()’ **function**. You get errors, why is that?
- Create a **child contract** of ‘External’ named ‘External2’. Deploy it and observe what was inherited from the **parent contract.**
- Copy the ‘getexternalfunc()’ **function** in the parent contract and paste it in the ‘External2’ contract. Uncomment it and deploy the contract. Will it run now?

Task

Design a Solidity **contract** named ‘Apartment’. Declare an **internal** variable ‘tenantcount’ of **uint** type to keep track of the number of tenants. Implement an **external** ‘joinApartment()’ **function** that allows new tenants to join. Create a **child contract** ‘Tenant’, run the ‘joinApartment()’ function in the ‘Tenant’ contract. Attempt to make a ‘getter()’ function for ‘tenantcount’ in the ‘Tenant’ contract as well. Is it possible?

Visibility Specifier	**Function**	**State variable**
Public	All contracts can call the function with public visibility	The State variable declared with the public visibility Specifier can be accessed

	specifier.	by all the contracts.
Private	Only the contract containing the function can call the function with visibility Specifier private.	The State variables declared with private visibility Specifier can only be used by the same contract in which it is declared.
Internal	The contract containing the function and its child contracts can call the function with internal visibility Specifier.	The State variable declared with internal visibility Specifier can be used by the same contract or its child contracts.
External	The external contracts can call the function with external visibility Specifier.	The state variable does not have external visibility Specifier.
Default visibility Specifier	The default visibility Specifier for functions is public in Solidity.	The default visibility Specifier for the State variable is internal in Solidity.

Reference: https://www.geeksforgeeks.org/function-visibility-specifier-in-solidity/

Note:

State variable visibility modifiers (public, private, and internal) control the visibility of variables within a contract, while function visibility modifiers control the accessibility of functions.

INHERITANCE

1 Introduction

Inheritance is a fundamental concept in **object-oriented programming** that allows for code reuse and the creation of hierarchical relationships between classes.

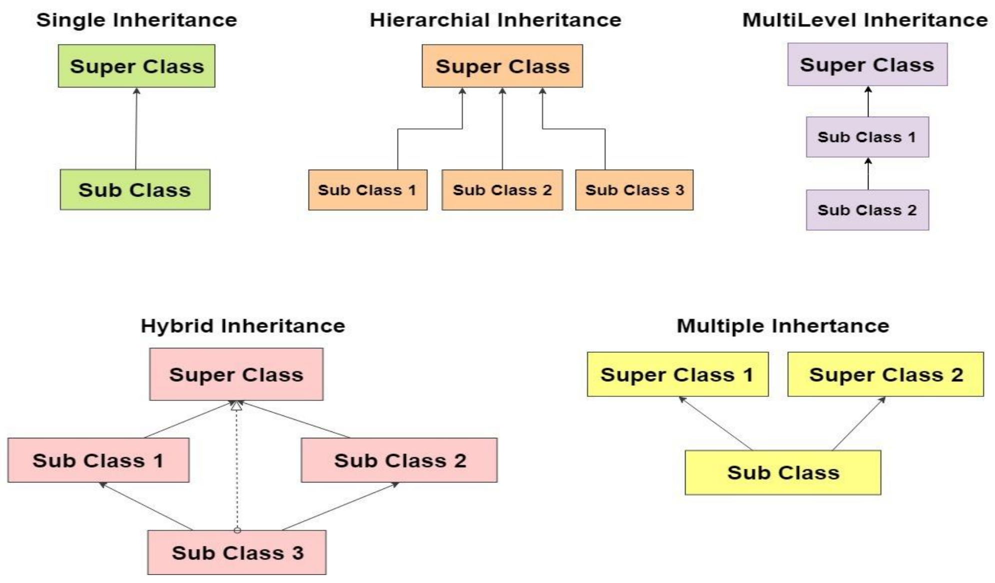
In Solidity, the programming language for Ethereum smart contracts, inheritance is also supported, enabling the reuse of code and the establishment of relationships between contracts.

In Solidity, the inheritance keyword **'is'** used to denote inheritance between contracts.

2 virtual and override Keywords

There are two important keywords related to inheritance:

1) The **'virtual'** keyword is used to declare functions in the parent contract that are **expected to be overridden** in its child contracts. This keyword indicates that the function can be extended or modified in the child contracts.
2) The **'override'** keyword is used in child contracts to indicate that the functions they define are **overriding the functions** in the parent contract. This keyword ensures that the child contracts provide their own implementation of the function and prevents accidental overriding.



3 Types of Inheritance

Here are the commonly used types of inheritance in Solidity:

1) Single Inheritance:

➔ Single inheritance refers to the scenario where a derived contract **inherits from a single** base contract.

➔ The derived contract inherits all the members (state variables, functions, and modifiers) of the base contract. State variables are usually supported with getters and setters.

➔ This type of inheritance provides a **simple and straightforward** way to reuse code and extend functionality.

```
contract Base {
    uint public x;
    function setX(uint _x) public {
        x = _x;                    }
}
contract Derived is Base {
    function getX() public view returns (uint) {
        return x;                                }
}
```

TASK

1) Deploy and observe the above smart contract.
2) Change Parent function's **(setX)** visibility to Private, and observe the change.

TASK

Create a base contract **'Parent'** with an uint variable **'parentValue'**. Derive a contract **'Child'** from **'Parent'** with an additional uint variable **'childValue'**.

Deploy **'Child'** and demonstrate inheritance by accessing both **'parentValue'** and **'childValue'**.

2) Multi-level Inheritance:

- ➔ Multi-level inheritance occurs when a derived contract inherits from a base contract that itself inherits from another base contract.
- ➔ In other words, there is a **chain of inheritance**, where each derived contract further acts as a base contract for subsequent derived contracts.
- ➔ This allows for the creation of a hierarchical structure of contracts, with each **level building** upon the functionality of the previous level.

```
contract Base {

    string public base = "Base Contract";

    function foo() public view virtual returns (string memory) {

        return base;         }

}

contract Middle is Base {

    string public middle = "Middle Contract";

    function foo() public view virtual override returns (string memory) {

        return middle;       }

}

contract Derived is Middle {

    string public derived = "Derived Contract";

    function foo() public view virtual override returns (string memory) {

        return derived;      }

}
```

TASK

1) Deploy and observe each of the above smart contracts.
2) Does the number of variables in each contract remain the same, as we move down the chain?

TASK

Create three Solidity contracts: **‘BaseUser’**, **‘Moderator’**, and **‘Admin’**, with Moderator inheriting from **‘BaseUser’**, and **‘Admin’** inheriting from **‘Moderator’**.

Model a content management system on the blockchain using **multi-level inheritance** to manage user roles and permissions.

Deploy **‘Admin’** and showcase the inherited functions and attributes.

Hint:

1) **‘BaseUser’** would have just an address, **‘Moderator’** would have address and name (string) and **‘Admin’** would have address, name (string) and PIN (uint) as state variables.
2) A virtual function named **Print** would be used to emit state variables in each contract.

Direct and Indirect Initialization of Constructor

1) Deploy, observe and explain each contract separately.

2) Attach a javascript log for each contract and observe how inheritance is ordered.

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

contract Base {
    uint data;
    constructor(uint _data) {
        data = _data;       }

    function Print() public virtual returns (string memory) {
        return "Base Initialized";
    }
}

contract Derived is Base(2) {
    constructor() {}

    function getData() external view returns (uint) {
        uint result = data ** 2;
        return result;                              }

    function Print() public virtual override returns (string memory) {
        return "Direct Initialization";
    }
}

contract Derived2 is Base {
    constructor(uint _temp) Base(_temp) {}

     function getData() external view returns (uint) {
        uint result = data ** 4;
        return result;
    }
    function Print() public virtual override returns (string memory) {
        return "Indirect Initialization";
   }
}
contract Caller {
    Derived c = new Derived();

    function getResult() public returns(uint){
        c.Print();
        return c.getData();
    }
}
```

3) Hierarchical Inheritance:

- ➔ Hierarchical inheritance involves a base contract that serves as a parent to multiple derived contracts.
- ➔ Each derived contract **inherits from the same base** contract, sharing its properties and behavior.
- ➔ This type of inheritance is useful when multiple contracts need to access common functionality defined in a single base contract.

```
contract Base {
    uint public x;
    function setX(uint _x) public {
        x = _x;                    }           }
contract DerivedA is Base {
    function getX() public view returns (uint) {
        return x;          }                   }
contract DerivedB is Base {
    function getXTimesTwo() public view returns (uint) {
        return x * 2;     }                    }
```

TASK

1) Deploy and observe each of the above smart contracts.
2) Create **DerivedC** involving a function that returns a cube of **x**.

TASK

Design a set of Solidity contracts representing **'Car'**, **'ElectricCar'**, and **'HybridCar'**, with **'ElectricCar'** and **'HybridCar'** inheriting from **'Car'**.

Implement basic attributes like **'model'**, **'make'**, and specific attributes like **'batteryCapacity'** for **'ElectricCar'** and **'gasTankCapacity'** for **'HybridCar'**. Deploy instances of each type of car and demonstrate the inherited attributes.

4) Multiple Inheritance:

➔ Multiple inheritance refers to the ability of a derived contract to **inherit from multiple base** contracts simultaneously.
➔ This allows the derived contract to combine and inherit the features and behavior of multiple contracts.
➔ Solidity supports multiple inheritance, but it imposes some rules to ensure unambiguous function resolution when conflicts arise due to shared function names or signatures.

```
contract BaseA {
    uint public x;
    function setX(uint _x) public {
        x = _x;                   }
}
contract BaseB {
    string public name;
    function setName(string memory _name) public {
        name = _name;                            }
}
contract Derived is BaseA, BaseB {
    function getData() public view returns (uint, string memory) {
        return (x, name);                          }
}
```

TASK

Develop three Solidity contracts: **'Engine'**, **'Transmission'**, and **'Car'**, where **'Car'** inherits from both **'Engine'** and **'Transmission'**.

Implement attributes like **'horsepower'** in 'Engine' and **'gearType'** in 'Transmission'.

Create a **'Car'** instance that inherits features from both **'Engine'** and **'Transmission'** and demonstrate accessing and displaying these combined attributes.

TASK

Create three Solidity contracts: **‘WorkerRole’**, **‘ClientRole’**, and **‘Freelancer’**, with **‘Freelancer’** inheriting from both **‘WorkerRole**’ and **‘ClientRole’**.

Incorporate necessary state variables in each contract and some functions like **submitWork** in ‘WorkerRole’ and **createProject** in ‘ClientRole’.

Deploy a **‘Freelancer’** instance inheriting attributes from both roles, and demonstrate invoking these combined actions.

5) Hybrid Inheritance:

➔ Hybrid inheritance is a **combination** of multiple inheritance and hierarchical inheritance.
➔ It involves a derived contract that inherits from multiple base contracts, some of which may also serve as base contracts for other derived contracts.
➔ This type of inheritance can be beneficial when creating complex contract relationships and reusing code across different contract hierarchies.

```
contract BaseA { … }

contract BaseB { … }

contract DerivedA is BaseA, BaseB { … }

contract DerivedB is BaseA, BaseB { … }

contract HybridDerived is DerivedA, DerivedB { … }
```

TASK

Develop Solidity contracts for **‘Participant’, ‘Supplier’**, **‘Manufacturer’**, and **‘Product’**, where **‘Product’** inherits from **‘Supplier’** and **‘Manufacturer’**.

Incorporate necessary state variables in each contract and some functions like **‘placeOrder’** in ‘Supplier’, **‘manufactureProduct’** in ‘Manufacturer’, and **‘trackProduct’** in ‘Product’.

Deploy an instance of **‘Product’** inheriting features from all roles, and demonstrate invoking these combined actions for efficient supply chain management.

TASK

1) Deploy, observe and explain each contract separately.
2) Attach a javascript log for each contract and observe how inheritance is ordered.

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

/* Inheritance tree :
    Car
  /     \
TOYOTA  BMW
  \     /
   Supra
*/
contract Car {
    event Log(string message);

    function demo() public virtual {
        emit Log("Base Car demo Called");
    }
    function temp() public virtual {
        emit Log("Base Car temp Called");
    }
}

contract TOYOTA is Car {
    function demo() public virtual override {
        emit Log("TOYOTA demo called");
    }
    function temp() public virtual override {
        emit Log("TOYOTA temp called");
        super.temp();
    }
}

contract BMW is Car {
    function demo() public virtual override {
        emit Log("BMW demo called");
    }
    function temp() public virtual override {
        emit Log("BMW temp called");
        super.temp();
    }
}

contract Supra is TOYOTA, BMW {
    function demo() public override(TOYOTA, BMW) {
        super.demo();
    }
    function temp() public override(TOYOTA, BMW) {
        super.temp();
    }
}
```