Week 6 – Contract Interaction and Libraries

Dr. Minfeng Qi

04 September 2023

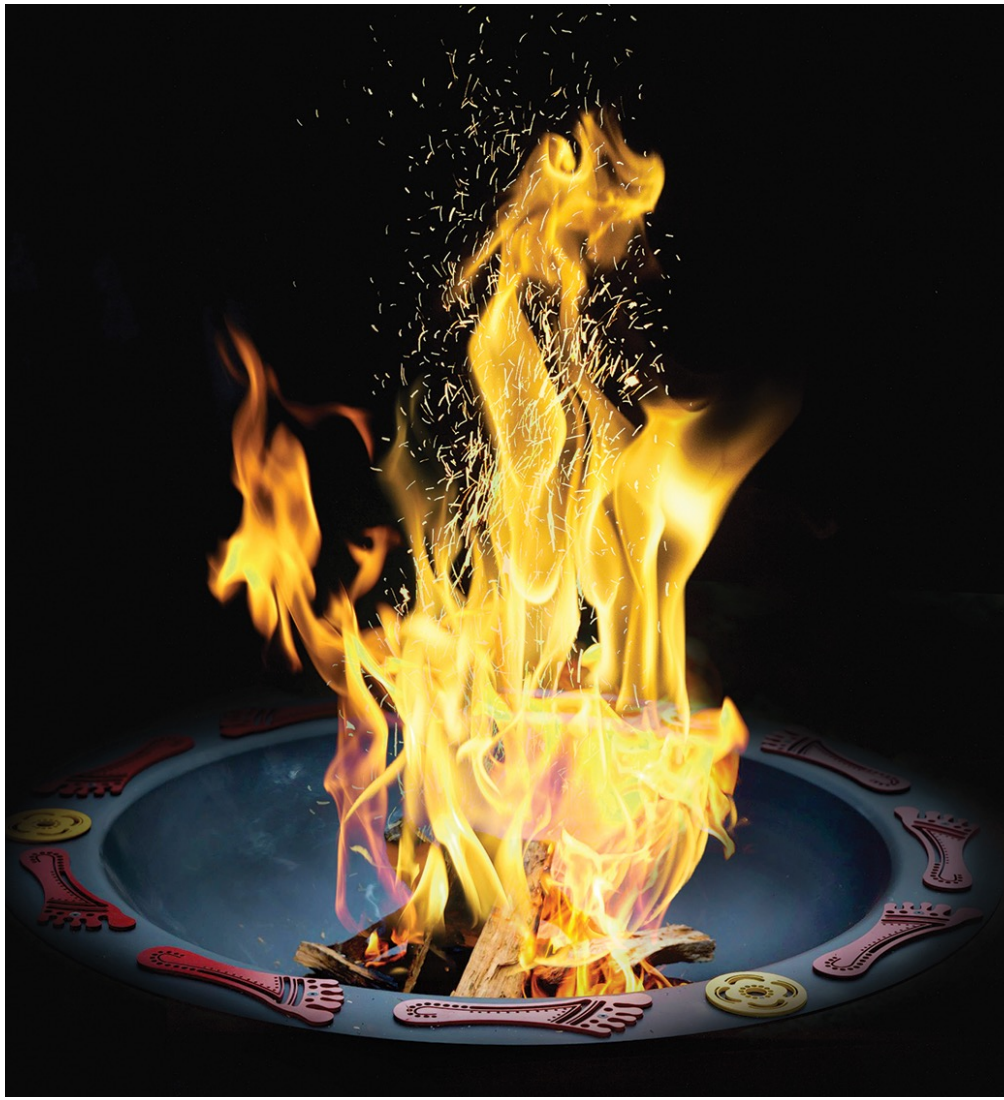
Acknowledgement of Country

We respectfully acknowledge the Wurundjeri People of the Kulin Nation, who are the Traditional Owners of the land on which Swinburne's Australian campuses are located in Melbourne's east and outer-east, and pay our respect to their Elders past, present and emerging.

We are honoured to recognise our connection to Wurundjeri Country, history, culture, and spirituality through these locations, and strive to ensure that we operate in a manner that respects and honours the Elders and Ancestors of these lands.

We also respectfully acknowledge Swinburne's Aboriginal and Torres Strait Islander staff, students, alumni, partners and visitors.

We also acknowledge and respect the Traditional Owners of lands across Australia, their Elders, Ancestors, cultures, and heritage, and recognise the continuing sovereignties of all Aboriginal and Torres Strait Islander Nations.



Visibility

- **external** - External functions are called by other contracts. To call an external function within the contract, use the *this.function_name()* method.
- **public** - Public functions/variables can be used directly both externally and internally. For public state variables, Solidity automatically creates a getter function for it.
- **internal** - Internal functions/variables can only be used internally or within derived contracts.
- **private** – Private functions/variables can only be used internally and cannot be used in derived contracts.

Constructor and Modifier

Constructor is a special function. Each contract can define one, and it runs automatically once when the contract is deployed. It can be used to initialize some parameters of the contract, such as initializing the contract's owner address.

```
2   pragma solidity ^0.8.0;
3
4       address owner;
5
6       constructor() {
7           owner = msg.sender;
8       }
```

Modifiers in Solidity are a unique syntax, similar to decorators in object-oriented programming. They declare characteristics that a function possesses and help reduce code redundancy. The primary use case for modifiers is to perform checks before running a function, such as checking addresses, variables, balances, etc.

```
11      // define modifier
12      modifier onlyOwner {
13          require(msg.sender == owner);
14          _;
15      }
16
17      function changeOwner(address _newOwner) external onlyOwner{
18          owner = _newOwner;
19      }
```

SWINBURNE UNIVERSITY OF TECHNOLOGY

Declare Events

The events are declared with the event keyword, followed by event name, then the type and name of each parameter to be recorded. Let's take the Transfer event from the ERC20 token contract as an example：

```
event Transfer(address indexed from, address indexed to, uint256 value);
```

Transfer event records three parameters: from, to, and value, which correspond to the address where the tokens are sent, the receiving address, and the number of tokens being transferred. Parameter from and to are marked with indexed keywords, which will be stored at a special data structure known as topics and easily queried by programs.

Emit Events

```
11  contract Owner {
12
13      address private owner;
14
15      // event for EVM logging
16      event OwnerSet(address indexed oldOwner, address indexed newOwner);
17
18      // modifier to check if caller is owner
19      modifier isOwner() {
20          // If the first argument of 'require' evaluates to 'false', execution terminates and all
21          // changes to the state and to Ether balances are reverted.
22          // This used to consume all gas in old EVM versions, but not anymore.
23          // It is often a good idea to use 'require' to check if functions are called correctly.
24          // As a second argument, you can also provide an explanation about what went wrong.
25          require(msg.sender == owner, "Caller is not owner");
26          _;
27      }
28
29      /**
30       * @dev Set contract deployer as owner
31       */
32      constructor() {
33          console.log("Owner contract deployed by:", msg.sender);
34          owner = msg.sender; // 'msg.sender' is sender of current call, contract deployer for a constructor
35          emit OwnerSet(address(0), owner);
36      }
```

We can emit events in functions. In the above example, each time the constructor is called, OwnerSet event will be emitted and corresponding parameters will be recorded.

Inheritance

Inheritance is one of the core concepts in object-oriented programming, which can significantly reduce code redundancy. It is a mechanism where you can derive a class from another class for a hierarchy of classes that share a set of attributes and methods. In solidity, smart contracts can be viewed objects, which supports inheritance.

There are two important keywards for inheritance in Solidity:

- virtual: If the functions in the parent contract are expected to be overridden in its child contracts, they should be declared as virtual.
- Override: If the functions in the child contract override the functions in its parent contract, they should be declared as override.

```
mapping(address => uint256) public override balanceOf;
```

Inheritance Example

```
3   contract Grandfather {
4       event Log(string msg);
5
6       // Apply inheritance to the following 3 functions: hip(), pop(), man(), then log "Grandfather".
7       function hip() public virtual{
8           emit Log("Grandfather");
9       }
10
11      function pop() public virtual{
12          emit Log("Grandfather");
13      }
14
15      function Grandfather() public virtual {
16          emit Log("Grandfather");
17      }
18  }
```

```
20  contract Father is Grandfather{
21      // Apply inheritance to the following 2 functions: hip() and pop(), then change the log value to "Father".
22      function hip() public virtual override{
23          emit Log("Father");
24      }
25
26      function pop() public virtual override{
27          emit Log("Father");
28      }
29
30      function father() public virtual{
31          emit Log("Father");
32      }
33  }
```

Let's define a contract called Father, which inherits the Grandfather contract. The syntax for inheritance is contract Father is Grandfather, which is very intuitive. In the Father contract, we rewrote the functions hip() and pop() with the override keyword, changing their output to "Father". We also added a new function called father, which output a string "Father".

Multiple Inheritance

Solidity contracts can inherit from multiple contracts.

- When inheriting, you should follow the order from highest to lowest seniority. For instance, if we write a contract named ”Son" which inherits from contracts ”Grandfather" and ”Father", it should be written as contract Son is *Grandfather, father*. Writing it as contract Son is *Father, Grandfather* would throw an error.
- If a function exists in multiple inherited contracts, such as the hip() and pop() functions in some examples, it must be overridden in the child contract; otherwise, an error will occur.
- When overriding a function that has the same name in multiple parent contracts, the override keyword should be followed by the names of all parent contracts, like override(*Grandfather, father*).

```
24  contract Son is Grandfather, Father{
25      function hip() public virtual override(Grandfather, Father){
26          emit Log("Son");
27      }
28
29      function pop() public virtual override(Grandfather, Father) {
30          emit Log("Son");
31      }
```

Functions from Parent Contracts

In child contracts, there are two ways to invoke functions from parent contracts: direct invocation and using the super keyword.

- **Direct Invocation**: Child contracts can directly call a function from a parent contract using the syntax ParentContractName.functionName(). For instance, using Grandfather.pop():
- **Using the super Keyword**: Child contracts can use super.functionName() to call a function from the immediate parent contract. In Solidity inheritance, the order of declaration is from right to left. So, if a contract is declared as contract Son is Grandfather, Father, then ”Father" is the immediate parent, and super.pop() will call Father.pop() instead of Grandfather.pop():

```
24  function callParent() public {
25      Grandfather.pop();
26  }
27
28  function callParentSuper() public {
29      // This will call the function from the immediate parent contract, i.e., Father.pop()
30      super.pop();
31  }
32
```

SWINBURNE UNIVERSITY OF TECHNOLOGY

Abstract

If a contract contains at least one unimplemented function (no contents in the function body **{}**), it must be labelled as **abstract**; Otherwise, it will not compile. Moreover, the unimplemented function needs to be labelled as **virtual**.

```
pragma solidity ^0.8.0;

// Define an abstract contract
abstract contract AbstractContract {
    // Declare an abstract function
    function doWork() public virtual returns(uint);
}

contract ConcreteContract is AbstractContract {

    // Implement the abstract function in the derived contract
    function doWork() public override returns(uint) {
        // Logic of the function
        uint workDone = 10;
        return workDone;
    }
}
```

In this code, AbstractContract is an abstract contract that declares an abstract function doWork(). The ConcreteContract inherits from AbstractContract and provides an implementation for the abstract function.

Interface

The interface contract is similar to the abstract contract, but it requires no functions are implemented. Rules of the interface:

- Cannot contain state variables.
- Cannot contain constructors.
- Cannot inherit non-interface contracts.
- All functions must be external and cannot have contents in the function body.
- The contract that inherits the interface contract must implement all the functions defined in it.

Although the interface does not implement any functionality, it is the skeleton of smart contracts. Interface defines what the contract does and how to interact with them: if a smart contract implements an interface (like ERC20 or ERC721), other Dapps and smart contracts will know how to interact with it. Because it provides two important pieces of information:

- The bytes4 selector for each function in the contract, and the function signatures function name (parameter type).
- Interface id (see EIP165 for more information)

When to use Interface

If we know that a contract implements the IERC721 interface, we can interact with it without knowing its detailed implementation.

The Bored Ape Yacht Club BAYC is an ERC721 NFT, which implements all functions in the IERC721 interface. We can interact with the BAYC contract with the IERC721 interface and its contract address, without knowing its source code. For example, we can use balanceOf() to query the BAYC balance of an address, or use safeTransferFrom() to transfer a BAYC NFT.

```
contract interactBAYC {
    // Use BAYC address to create interface contract variables (ETH Mainnet)
    IERC721 BAYC = IERC721(0xBC4CA0EdA7647A8aB7C2061c2E118A18a936f13D);

    // Call BAYC's balanceOf() to query the open interest through the interface
    function balanceOfBAYC(address owner) external view returns (uint256 balance){
        return BAYC.balanceOf(owner);
    }

    // Safe transfer by calling BAYC's safeTransferFrom() through the interface
    function safeTransferFromBAYC(address from, address to, uint256 tokenId) external{
        BAYC.safeTransferFrom(from, to, tokenId);
    }
}
```

Interface Example

We take **IERC721** contract, the interface for the **ERC721** token standard, as an example. It consists of 3 events and 9 functions, which all **ERC721** contracts need to implement. In interface, each function ends with **;** instead of the function body **{ }**. Moreover, every function in interface contract is by default **virtual**, so you do not need to label function as **virtual** explicitly.

```
35  interface IERC721 is IERC165 {
36      event Transfer(address indexed from, address indexed to, uint256 indexed tokenId);
37      event Approval(address indexed owner, address indexed approved, uint256 indexed tokenId);
38      event ApprovalForAll(address indexed owner, address indexed operator, bool approved);
39
40      function balanceOf(address owner) external view returns (uint256 balance);
41
42      function ownerOf(uint256 tokenId) external view returns (address owner);
43
44      function safeTransferFrom(address from, address to, uint256 tokenId) external;
45
46      function transferFrom(address from, address to, uint256 tokenId) external;
47
48      function approve(address to, uint256 tokenId) external;
49
50      function getApproved(uint256 tokenId) external view returns (address operator);
51
52      function setApprovalForAll(address operator, bool _approved) external;
53
54      function isApprovedForAll(address owner, address operator) external view returns (bool);
55
56      function safeTransferFrom( address from, address to, uint256 tokenId, bytes calldata data) external;
57  }
```

Libraries

Library is a special type of contract, designed to enhance the reusability of Solidity code and reduce gas consumption. Library contracts typically consist of a collection of useful functions.

For example, the String library contract is a code library used for converting uint256 types into their corresponding string types.

```
library Strings {
    bytes16 private constant _HEX_SYMBOLS = "0123456789abcdef";

    /**
     * @dev Converts a `uint256` to its ASCII `string` decimal representation.
     */
    function toString(uint256 value) public pure returns (string memory) {
        // Inspired by OraclizeAPI's implementation - MIT licence
        // https://github.com/oraclize/ethereum-api/blob/b42146b063c7d6ee1358846c198246239e9360e8/oraclizeAPI_0.4.25.sol

        if (value == 0) {
            return "0";
        }
        uint256 temp = value;
        uint256 digits;
        while (temp != 0) {
            digits++;
            temp /= 10;
        }
        bytes memory buffer = new bytes(digits);
        while (value != 0) {
            digits -= 1;
            buffer[digits] = bytes1(uint8(48 + uint256(value % 10)));
            value /= 10;
        }
        return string(buffer);
    }
    ...
}
```

SWINBURNE UNIVERSITY OF TECHNOLOGY

How to Use Library Contracts

- Using the *'using for'* directive

  The directive 'using A for B;' is used to attach library functions (from library A) to any type (B). After the directive is added, the functions in library A are automatically added as members of the B type variable and can be called directly. Note: when called, this variable will be passed as the first argument to the function:

```
using Strings for uint256;
function getString1(uint256 _number) public pure returns(string memory){
        return _number.toHexString();
}
```

- Calling library functions through the library contract's name

  You can also call a library function directly using the name of the library contract. In this case, you would use the name of the library, followed by the function you want to call."

```
function getString2(uint256 _number) public pure returns(string memory){
        return Strings.toHexString(_number);
}
```

Import

Solidity supports the use of the **import** keyword to import contracts from other source codes, making development more modular。

- import by using the relative position of the source file

  import ‘./father.sol’;

- import contracts from the internet by using the source file's URL

  import 'https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/Address.sol’;

- import via the npm directory

  import '@openzeppelin/contracts/access/Ownable.sol';

- import specific contracts via global symbols

  import {father} from ‘./father.sol';

Error Handling

- error statement is a new feature in solidity 0.8. It saves gas and informs users why the operation failed. It is the recommended way to throw error in solidity. Custom errors are defined using the error statement, which can be used inside and outside of contracts.

  In functions, error must be used together with revert statement.

```
error TransferNotOwner(); // custom error
revert TransferNotOwner();
```

- require statement was the most commonly used method for error handling prior to solidity 0.8. It is still popular among developers.

```
require(condition, "error message");
```

- The assert statement is generally used for debugging purposes, because it does not include error message to inform the user.

```
assert(condition);
```

Overloading

In Solidity, functions can be overloaded, meaning that functions with the same name but different input parameter types can coexist. They are considered as distinct functions. However, note that Solidity does not allow overloading of modifiers.

Function Overloading: For example, we can define two functions named saySomething(). One doesn't take any parameters and returns "Nothing"; the other takes a string parameter and returns the provided string.

```
pragma solidity ^0.8.4;

contract OverloadingExample {
    // First saySomething function with no parameters
    function saySomething() public pure returns(string memory) {
        return "Nothing";
    }

    // Overloaded saySomething function that takes a string parameter
    function saySomething(string memory _message) public pure returns(string memory) {
        return _message;
    }
}
```

Thank you

04 Sep 2023