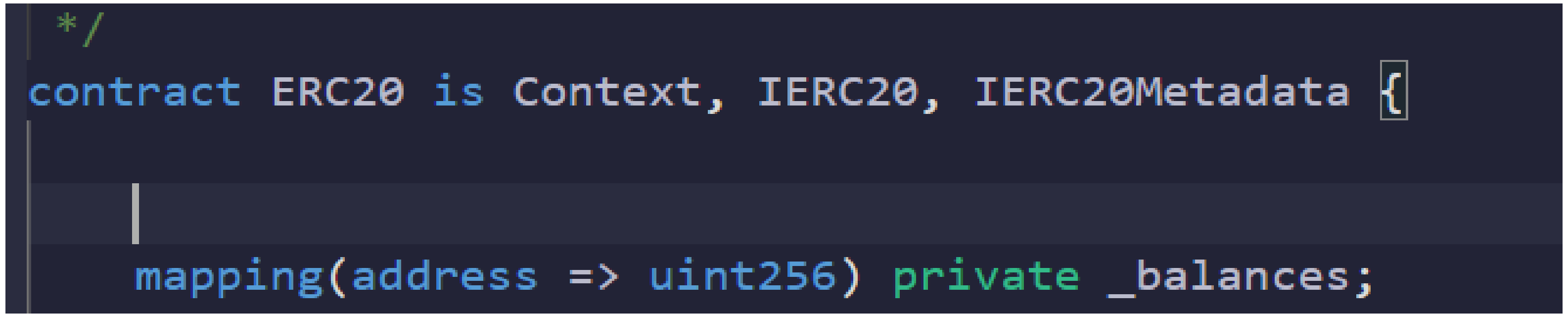
BCDV 1010
Smart Contract Development Essentials

2023 January
week 03 - class 14

Contract Inheritance

Inheritance is one of the most important features of the object-oriented programming language. It is a way of extending a program's functionality, separating the code, reducing the dependency, and increasing the existing code's re-usability. In Solidity, multiple contracts can be inherited into a single contract.

In the following example, the ERC20 contract is inheriting multiple other smart contracts
The “is” keyword to inherit from a contract or multiple contracts



```
 */
contract ERC20 is Context, IERC20, IERC20Metadata {


    mapping(address => uint256) private _balances;
```

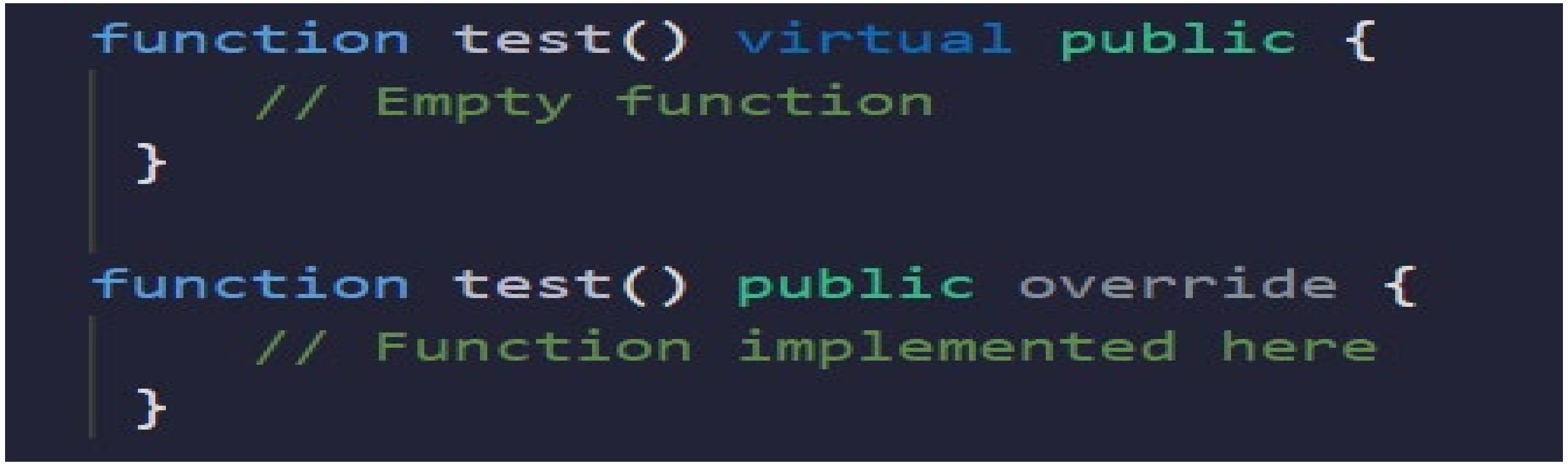
Once you inherit the contract all the non-private functions in the inherited or the parent contract is accessible in the child contract.

Virtual functions and overrides

If you wish to implement some of the functions with exact same name as in the parent contract, Then in the parent contract you can have a function of type **virtual** and in the child contract you must override it.

virtual functions are always empty. The function logic is implemented in the **override** function.

override function must also need to have the same parameters.

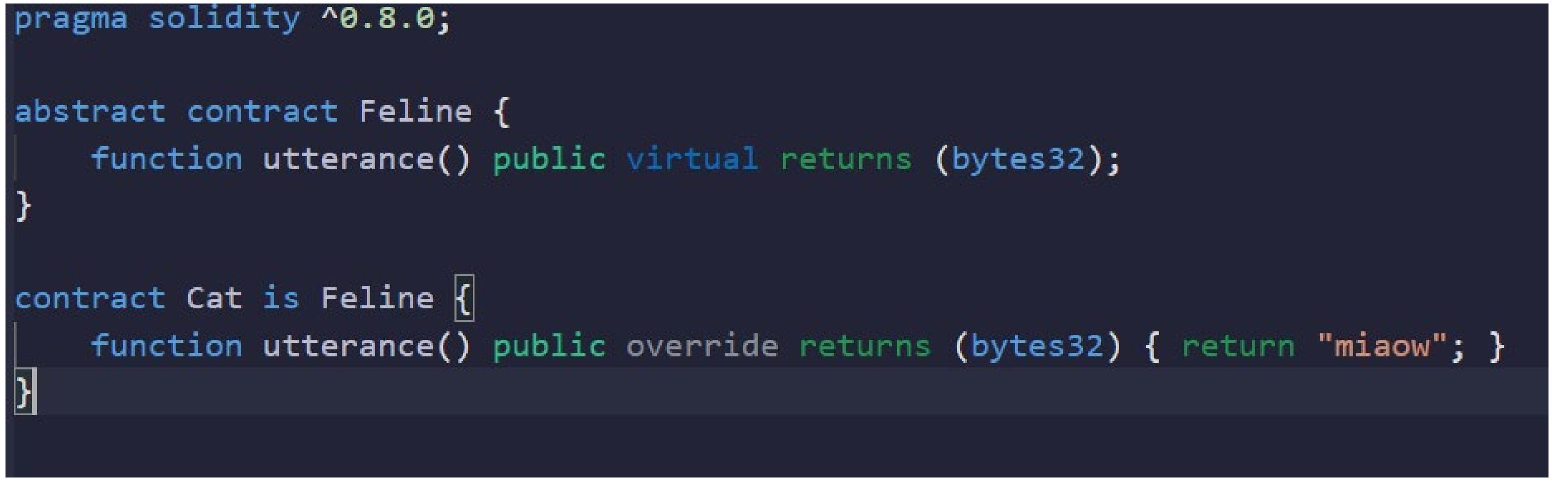


```
function test() virtual public {
    // Empty function
}

function test() public override {
    // Function implemented here
}
```

Abstract Contracts

So, If you are creating any contract which contains at least one or more virtual functions (The function which is not implemented) then it must be marked as an **abstract contract.**

The keyword **virtual** can only be used inside an abstract contract.



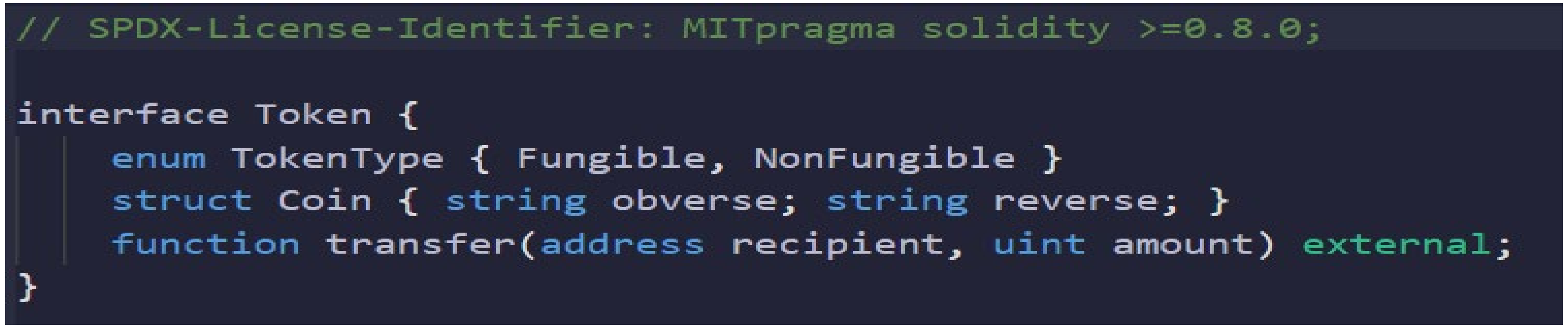
```
pragma solidity ^0.8.0;

abstract contract Feline {
    function utterance() public virtual returns (bytes32);
}

contract Cat is Feline {
    function utterance() public override returns (bytes32) { return "miaow"; }
}
```

If a contract inherits from an abstract contract and does not implement all non-implemented functions by overriding, it needs to be marked as abstract as well.

Interfaces

Interfaces are similar to abstract contracts, but they cannot have any functions implemented. There are further restrictions:

- They cannot inherit from other contracts, but they can inherit from other interfaces.
- All declared functions must be external.
- They cannot declare a constructor.
- They cannot declare state variables.



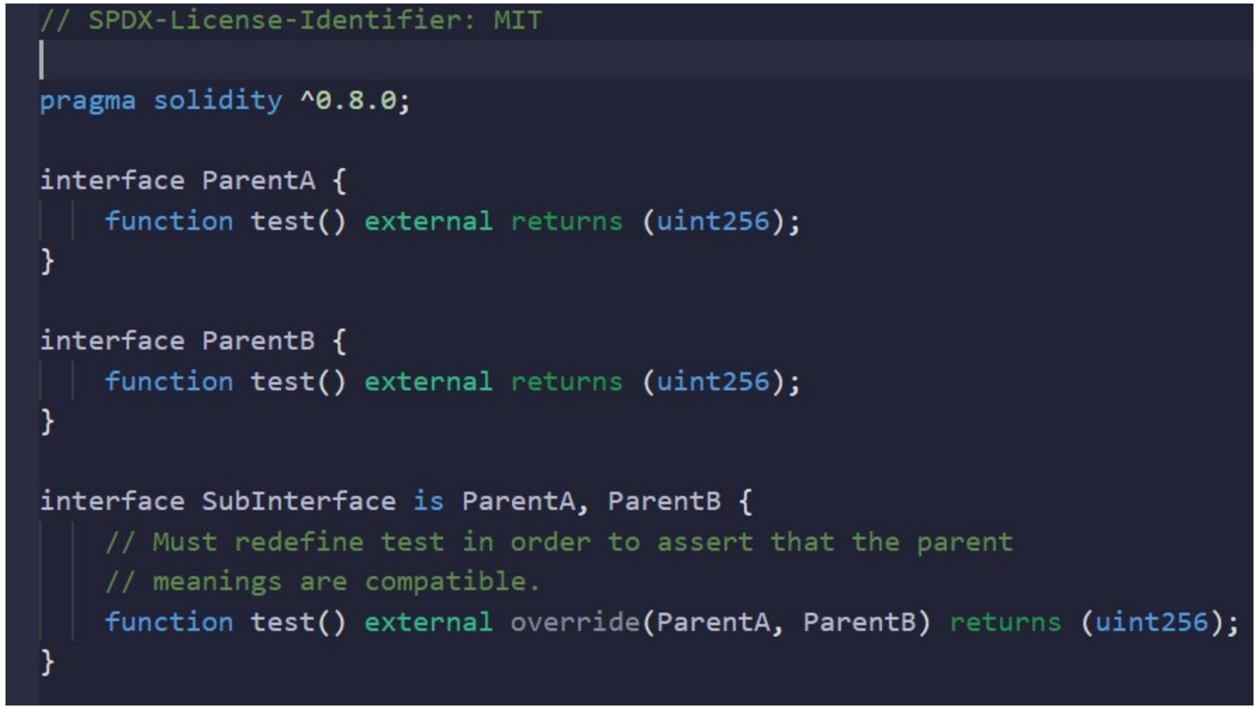
```
// SPDX-License-Identifier: MITpragma solidity >=0.8.0;

interface Token {
    enum TokenType { Fungible, NonFungible }
    struct Coin { string obverse; string reverse; }
    function transfer(address recipient, uint amount) external;
}
```

Some of these restrictions might be lifted in the future.

Interfaces are basically limited to what the Contract ABI can represent, and the conversion between the ABI and an interface should be possible without any information loss.

Interfaces are denoted by their own keyword.

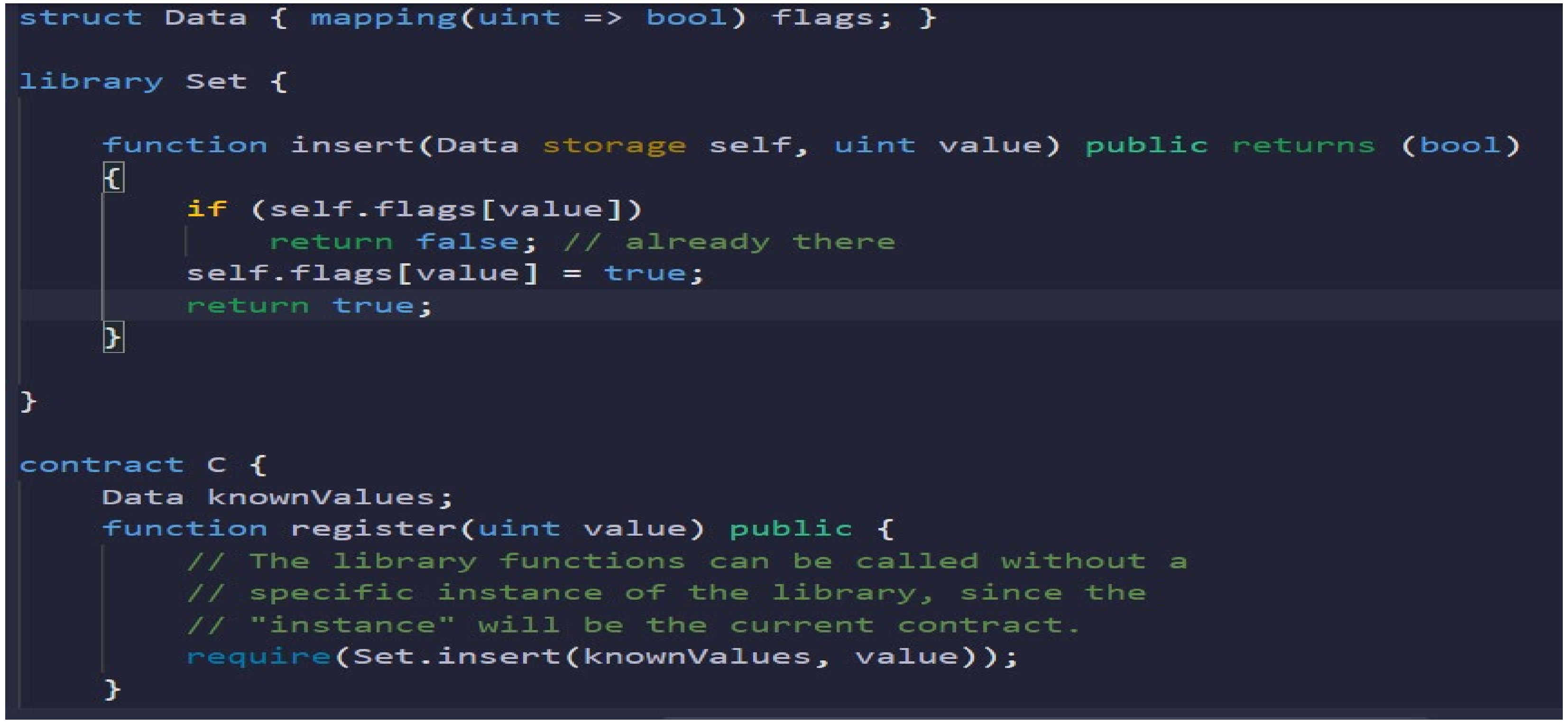
Interfaces can inherit from other interfaces. This has the same rules as a normal inheritance.



```solidity
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

interface ParentA {
    function test() external returns (uint256);
}

interface ParentB {
    function test() external returns (uint256);
}

interface SubInterface is ParentA, ParentB {
    // Must redefine test in order to assert that the parent
    // meanings are compatible.
    function test() external override(ParentA, ParentB) returns (uint256);
}
```

Libraries

Libraries are similar to contracts, but their purpose is that they are deployed only once at a specific address and their code is reused using the DELEGATECALL feature of the EVM.



```
struct Data { mapping(uint => bool) flags; }

library Set {

    function insert(Data storage self, uint value) public returns (bool)
    {
        if (self.flags[value])
            return false; // already there
        self.flags[value] = true;
        return true;
    }

}

contract C {
    Data knownValues;
    function register(uint value) public {
        // The library functions can be called without a
        // specific instance of the library, since the
        // "instance" will be the current contract.
        require(Set.insert(knownValues, value));
    }
```

This means that if library functions are called, their code is executed in the context of the calling contract, i.e. `this` points to the calling contract, and especially the storage from the calling contract can be accessed

Note: Storage means the storage space can be addressed from the caller contract so, we don't need to use the memory, but it does not mean that the data variables in the smart contract can be accessed.

In comparison to contracts, libraries are restricted in the following ways:

- they cannot have state variables
- they cannot inherit nor be inherited
- they cannot receive Ether
- they cannot be destroyed

The directive `using A for B;` can be used to attach library functions (from the library `A`) to any type (`B`) in the context of a contract. These functions will receive the object they are called on as their first parameter (like the `self` variable in Python)