Solidity

▼ Concepts

Interacting with other contracts

- **Interfaces *(main use case - interact with a deployed, “non-accessible” contract)* :** you declare functions that are in the remote smart contract inside the `interface` keyword. Then you use the interface’s name with the remote contract’s address as the parameter to call the contract. With that you can call the function of the remote contract. E.g : `ICounter(0x00000).increment();` (see bellow).

▼ *Technical specs of interfaces*

- cannot have any functions implemented.
- can inherit from other interfaces.
- all declared functions must be external.
- cannot declare a constructor.
- cannot declare state variables.
- functions are marked as `virtual` implicitly.

> 💡 Use interfaces when you can’t import the remote smart contract in your code (basically when you can’t have access to the remote sc’s code). That’s the main difference with the second (next) way for calling contracts.

```
// we assume the Counter smart contract is already deployed on the blockchain and not in that code
contract Counter {
    uint public count;

    function increment() external {
        count += 1;
    }
}

interface ICounter {
    function count() external view returns (uint);

    function increment() external;
}

contract MyContract {
    function incrementCounter(address _counter) external {
        ICounter(_counter).increment();
    }

    function getCount(address _counter) external view returns (uint) {
        return ICounter(_counter).count();
```

```
    }
}
```

- **Calling Other Contract *(main use case - interact with a deployed, “code-included”, accessible contract)* :** like interfaces but you should have imported the wanted contract in your code. You also don’t need to declare remote sc’s functions (like in `interface`).

```
// if the contract B isn't in the same file as contract A, you should import it like :
// import "./contractB.sol";

contract A {
    address addressB;

    modifier(address _addressB) {
        addressB = _addressB;
    }

    function callHelloWorld() external view returns (string memory) {
        B b = B(addressB);
        return b.HelloWorld();
    }
}

contract B {
    function HelloWorld() external pure returns (string memory) {
        return "Hello world";
    }
}
```

- **Inheritance *(main use case - blueprint of a contract)* :** Access the functions of another contract (in the same code) in order to override it. Basically, you copy paste a contract, and this pasted contract have the same functions as the parent contract + it can modify the functions.

To modify a function, you have to add the `virtual` keyword for the function that will be modified.

To override the function likely to be modified, you have to add the `override` keyword to your child contract’s function.

(see bellow)

> 💡 When modifying function, you only modify a function of the child contract. The parent contract stay the same.

```
/* 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();
    }
}
```

- **Abstract contracts *(main use case - blueprint of a contract with unimplemented functions)* :** like a normal contract, except that you may have unimplemented functions (function with nothing in it like with `interface`). You can use the `abstract` keyword if you want, but it’s optional. This type of contracts can’t be deployed, but other contracts can inherit from it.

```
abstract contract MyAbstractContract {
  int a;
  int b;
```

```
  function foo() external virtual pure returns (string memory);

  function createSum(int x, int y) external {
    a = x;
    b = y;
  }

  function returnSum() external view returns (int) {
    return a + b;
  }
}

contract MyContract is MyAbstractContract {
  // if nothing in it, MyContract would be abstract too because it inherits

  function foo() external override pure returns (string memory) {
    return "Hello World";
  }
}
```

💡 Good Stackoverflow answer about the difference between interfaces and abstract contracts — Good article about the same subject.

- **Libraries *(main use case - reuse gas-efficiently code)* :** like contract, except that you can't declare a state variable nor send ether. An example of library : SafeMath provided by OpenZeppelin. As you see in the example bellow, it's easy to use. That's more lightweight/gas efficient than creating a whole contract and calling it. You can deploy a library, but it's better not to by declaring all functions inside as internal.

```
library SafeMath {
    function add(uint x, uint y) internal pure returns (uint) {
        uint z = x + y;
        require(z >= x, "uint overflow");

        return z;
    }
}

contract TestSafeMath {

    // first way to proceed
    using SafeMath for uint;
    function testAdd1(uint x, uint y) public pure returns (uint) {
        return x.add(y);
    }

    // second way to proceed
    function testAdd2(uint x, uint y) public pure returns (uint) {
        return SafeMath.add(x, y);
    }

}
```

- **Contract that creates other contracts**

I created this sheet for myself in order to really clarifying these concepts by explaining them with my own words. It probably contains errors (you can contact via Twitter or email - xeway@pm.me - if you encounter an issue). These code snippets, and these explanations are mostly by Smart Contract Programmer (his website too) and EatTheBlocks. Also special thanks to @JetJadeja for having reviewed this sheet.