Mr A.Dekhinet

University of BATNA 2 / Computer Science Department

a.dekhinet@univ-batna2.dz

http://staff.univ-batna2.dz/dekhinet-abdelhamid

Chapter 5

dApp : Solidity language

Introduction

- Solidity is a high-level language to write smart contracts for Ethereum.
- Influenced by C++, Python, and JavaScript.
- Smart contracts can be defined as encapsulated units, similar to classes in object-oriented languages.
- Supports multiple inheritance.
- Supports function overloading.
- Supports import statements to help modularize your code.
- Statically-typed language, which means that the data type of each variable needs to be specified.
- Other languages for smart contracts : Go, Python, JS, Vyper, LLL, Serpent, Bamboo, …

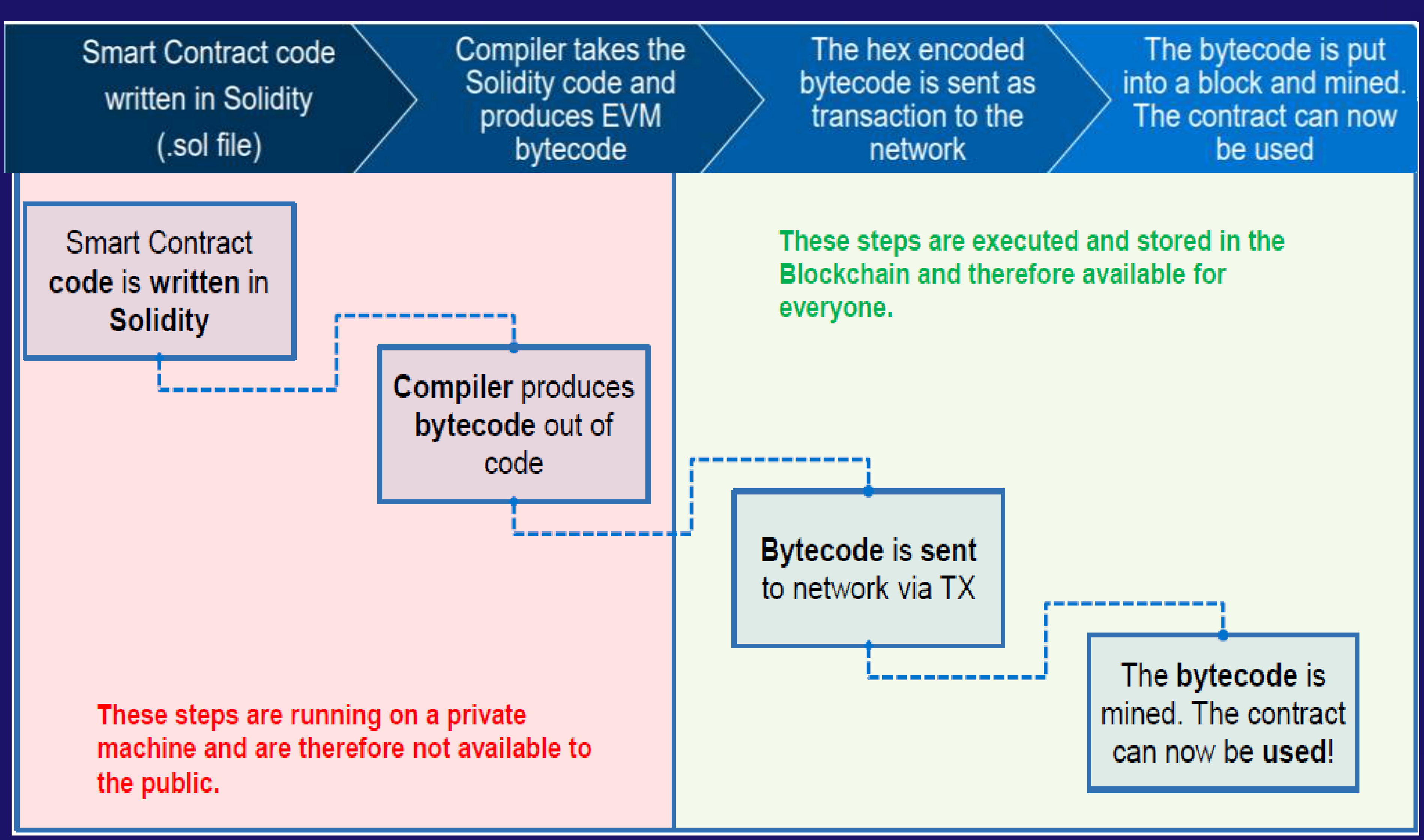
Introduction

- A smart contract has its own, persistent state on the blockchain which is defined by state variables in the contract.
- Functions are used to change the state of the smart contract or to perform other computations.
- Solidity is compiled to bytecode which is persistent and immutable once deployed to the blockchain.
- Solidity code is stored in files with extension .sol .
- A good practice is to have one separate .sol file per contract.

Introduction

- The Solidity compiler takes a .sol file as input and generates the corresponding sequence of EVM opcode instructions and an ABI.
- ABI (Application Binary Interface) : JSON file that can be used by other applications to interact with smart contract.
- The opcode instructions are then encoded as hex bytecode.
- The contract is deployed via a special transaction containing the bytecode as payload.
- Once the transaction is mined, a new contract account on the Ethereum network is created.
- The contract is now usable.

Solidity Smart Contract Deployement



Solidity Smart Contract Compilation

```
• // SPDX-License-Identifier: MIT
• pragma solidity ^0.8.17;
• contract Hello {
•     function greet() public returns(bytes32)
        {
•         return "Hello world";
•       }
• }
```

Compile

ABI


```
[
  {
    "inputs": [],
    "name": “greet",
    "outputs": [
          {
            "internalType": "bytes32",
            "name": "",
            "type": "bytes32"
          }
     ],
       "stateMutability": "nonpayable",
         "type": "function"
  }
]
```


ByteCode


```
6080604052348015610 00f575f80fd5b5060c
e8061001c5f395ff3fe6080604052348015 60
0e575f80fd5b5060043610602 6575f3560e01
c8063cfae32171460 2a575b5f80fd5b603060
44565b604051603b9190608156 5b604051809
10390f35b5f7f48656c6c6f20776f726c6400
0000000000000000000000000000000000000
000905090565b5f8190509190505 65b607b81
606b565b82525050565b5f602082019050609
25f8301846074565b9291505056fea2646970
667358221220b6ebb686a38ddf8f82ed0261c
e5f0849a30ff26735b78e15ecd485c49608ec
df64736f6c63430008160033
```

Blockchain for Smart Contract Deployment

- Smart contract can be deployed in several blockchains
- Private: e.g., Ganache sets a personal Ethereum blockchain for running tests, executing commands, and inspecting the state while controlling how the chain operates.
- Public Test (Testnet): Like Ropsten, Kovan and Rinkeby which are existing public blockchains used for testing and which do not use real funds.
- Public Real (Mainnet): Like Bitcoin and Ethereum which are used for real and available to join.

Anatomy of Smart Contract

```
pragma solidity ^0.4.17;
contract SimpleDeposit {

    address owner;
    mapping (address => uint) balances;
    uint public feed; /* A getter method is automatically created */

    event LogDepositMade(address from, uint amount);

    constructor() public {
           owner = msg.sender;
    }

    modifier minAmount(uint amount) {
           require(msg.value >= amount);
            _;
     }

    function SimpleDeposit() public payable {
        balances[msg.sender] = msg.value;
    }

    function deposit() public payable minAmount(1 ether) {
        balances[msg.sender] += msg.value;
        LogDepositMade(msg.sender, msg.value);
    }

    function getBalance() public view returns (uint  balance) {
        return balances[msg.sender];
    }

    function withdraw(uint amount) public {
        require(owner=msg.sender);
        if (balances[msg.sender] >= amount) {
        balances[msg.sender] -= amount;
        msg.sender.transfer(amount);
    }
}
```

Smart contract can contains declarations of :

- *State Variables*
- *Struct Types*
- *Enum Types*
- *Import statement*
- *Functions*
- *Function Modifiers*
- *Function Constructor*
- *Events*
- *Errors*
- *Comments*
- *Etc …*

And solidity file can be composed of four high-level construct :

- *SPDX license identifier*
- *Pragma*
- *Comments*
- *Import*
- *Contracts/Library/Interface*

Data type

- **Boolean (bool)** : Boolean value, *true* or *false*, By default, it is *false*.
- **Integer (int, uint)** : Signed (*int*) and unsigned (*uint*) integers of various size, declared in increments of 8 bits from *int8* to *int256* (unsigned of 8 up to 256 bits). Without a size suffix, 256-bit quantities are used, to match the word size of the EVM.
- **Fixed point (fixed, ufixed)** : Fixed-point numbers, declared with *(u)fixedMxN* where *M* is the size in bits (increments of 8 up to 256) and *N* is the number of decimals after the point (up to 18); e.g., *ufixed32x2*.

  N.B : Fixed point numbers are not fully supported by Solidity yet. They can be declared, but cannot be assigned to or from. In another word, not useful.
- **Byte** : Fixed-size array of bytes, holds a sequence of bytes from one to up to 32 (*bytes1* up to *bytes32*).
- **String** : String literals are written with either double or single-quotes. String data type can be considered as a variable-sized array of bytes. Byte has an advantage that it uses less gas, so better to use when we know the length of data.
- **Array** : Array data type can be static (Fixed size) or dynamic (Variable size). The type T of an array of k element is written as *T[k]* , and an array of dynamic size as *T[ ]* (e.g an array of 5 dynamic arrays of uint is specified as *uint[ ][5]*).
- **Enum** : User-defined type for enumerating discrete values: *enum Name {Label1, Label2, ...}*.
- **Struct :** User-defined data containers for grouping variables, allows users to create and define their own type in the form of structures : *struct Name {Type1 Variable1; Type2 Variable2; ...}*.
- **Mapping** : Mapping is the most used reference type, that stores the data in a ***key-value*** pair. It is like a hashtable or hashmap in Java, where data can be retrieved by key : *mapping (Key_Type ⇒ Value_Type) Variable_Name*
- **Address** : Hold a 20-byte value which represents the size of an Ethereum address. The address object has many member functions, the main ones being *balance* (Returns the account balance) and *transfer* (Transfers ether to the account).

Data types

```
•   pragma solidity ^0.8.17;
•   contract Data_Types {
•
•       bool transferable;
•       bytes1 public b = "a";
•       string hello = "Hello world";
•       bytes32[] public names;
•       bytes8[20] city;
•       uint32 age;
•       uint[5] odds = [1,3,5,7,9];
•
•       address payable payor;
•       mapping (address => uint) balances;
•
        enum season { spring,summer,autunm,winter }
•       season public Season;
•
        struct Order {
•           string buyer;
•           string product;
•           uint quantity;
•       }
•       mapping (address => Order) Orders;
•       Order ord;
•   }
```

Types of variables : Three types of variables

- **Local variables** : Declared within a function and are only accessible within that function. They are not stored in the Blockchain and their lifetime ends when the function execution is completed. Function parameters are always local.
- **State variables** : Declared at the contract level, outside function. They represent the contract's state on the blockchain and are accessible within the entire contract. State variables values are permanently stored in contract storage. Update the state requires a transactions and therefore costs ether.

Variable scope

Scope of local variables is limited to function in which they are defined but State variables can have three types of scopes :

Public − Public variables are accessible from within the contract and can be accessed from external contracts as well. Solidity automatically generates a getter function for public state variables.

Internal − Internal variables are accessible within the contract they are defined in and derived from contracts. They are not accessible from external contracts.

Private − Private variables are only accessible within the contract they are defined in. They are not accessible from derived contracts or external contracts.

- **Global variables** : Are predefined and special variables provided by the Solidity language. When a contract is executed in the EVM, it has access to a small set of global objects. These include the *block*, *msg*, and *tx* objects A global variable represents an attribute of an object (*block*, *msg*, *tx*, *Address*).

Variable Names

Solidity variable names are case-sensitive and should not start with a numeral (0–9). They must begin with a letter or an underscore character.

Global variables

- **Transaction/message call context** : The *msg* object is the transaction call (EOA originated) or message call (contract originated) that launched this contract execution.

msg.sender *(address) : The address that initiated this contract call*	
msg.value *(uint) : The value of ether sent with this call (in wei)*	
msg.gas *(Replaced by Gasleft function) : The amount of gas left of the contract execution*	
msg.data *(bytes) : Complete Calldata of the called function, holds the function identifier and its parameters values in the form of bytes*	
msg.sig *(bytes4) : The first four bytes of the call data (the function identifier).*	

- **Transaction context** : The *tx* object provides a means of accessing transaction-related information.

tx.gasprice *(uint): The gas price in the calling transaction*
tx.origin *(address): The address of the originating EOA for this transaction.*

Global variables

- **Block context** : The block object contains information about the current block

block.blockhash(blockNumber) *(Replaced by blockhash function) : The block hash of the specified block number, up to 256 blocks in the past.*
block.coinbase *(address) : The current block miner's address, recipient of the block reward.*
block.difficulty *(uint) : The difficulty (proof of work) of the current block*
block.gaslimit *(uint) : The maximum amount of gas that can be spent across all transactions included in the current block*
block.number *(uint) : The current block number (blockchain height)*
block.timestamp *(uint) : The current block timestamp placed by the miner*

- **Address object**

address.balance *(uint): The balance of the address, in wei. For example, the current contract balance is address(this).balance.*
address.transfer(amount) *function : Transfers the amount (in wei) to this address, throwing an exception on any error.*
address.send(amount) *function : Similar to transfer, only instead of throwing an exception, it returns false on error.*
address.call(payload) *: Low-level CALL function*
address.callcode(payload) *: Low-level CALLCODE function*
address.delegatecall() *: Low-level DELEGATECALL function*

Flow control statement

- While Loop
- Do …. While Loop
- For Loop
- Loop control : break and continue
- If statement
- If ... else statement
- if...else if... statement

Function

- Within a contract, we define functions that can be called by an EOA transaction or another contract.
- Functions are used to change and to read the state of a contract.
- The syntax we use to declare a function is as follows :

```
function FunctionName([parameters])
{public|private|internal|external}
[pure|constant|view|payable]
[modifiers]
[returns (return types)]
```

Function visibility

- public : Public is the default; such functions can be called by other contracts or EOA transactions, or from within the contract.
- external : External functions are like public functions, except they cannot be called from within the contract unless explicitly prefixed with the keyword this.
- internal : Internal functions are only accessible from within the contract ,they cannot be called by another contract or EOA transaction. They can be called by derived contracts (Those that inherit this one).
- private : Private functions are like internal functions but cannot be called by derived contracts

Function behavior

The set of keywords (pure, constant, view, payable) and other kind of declarations (fallback, Modifiers, …) affect the behavior of the function :

- view or constant : State read only function, it cannot modify the state variables nor alter the state of the blockchain. The term *constant* is an alias for *view*.

```
uint state ; // State variable
function add(uint a, uint b) public view returns (uint sum) { return a + b + state }
```

- pur : It cannot read or modify the state variables.

```
uint state ; // State variable
function add(uint a, uint b) public pur returns (uint sum) { return a + b }
```

- payable : A payable function is one that can receive incoming payments. Functions not declared as payable will reject incoming payments.

```
function deposit() external payable {
        // Payable keyword allows receiving Ether
        // Hide Code : address(this).balance += msg.value
    }
```

Function Modifier

- The function modifier is a special type of function in solidity.
- Modifiers are most often used to create conditions that apply to many functions within a contract.
- Modifiers are basically a reusable piece of code.
- The syntax of the definition start with the keyword *modifier* :

modifier FunctionName { …………..}

```
modifier isOwner() {
        require(msg.sender == owner, "Not the owner");
        _; // Actual function code is injected here
}
```

- Chaining of function modifiers : It is possible to apply multiple modifiers to a function. The modifiers will be resolved sequentially, starting from left to right.

Function Modifier

```
contract Owned {
    mapping (address => uint) balances;
    address public owner;

    constructor() {
        owner = msg.sender;
    }

    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }

    modifier isRich {
        require(msg.sender.balance > 1120 ether);
        _;
    }

    function deposit() public payable onlyOwner isRich returns (uint) {
        balances[msg.sender] += msg.value;
        return balances[msg.sender];
    }

}
```

Function Constructor

- Constructor is a special function that is only used once.
- When a contract is deployed, it also runs the constructor function, to initialize the state of the contract : Setting initial values of state variables or performing setup tasks.
- The constructor is run in the same transaction as the contract creation.
- The constructor function is optional, If there is no constructor defined, the default constructor will be executed automatically.

```
contract Owned {
    address public owner;
    constructor() {
        owner = msg.sender;
    }
}
```

- We can destroy the contract using Self destruct built-in function
- The function allows you to effectively remove a contract from the blockchain and send its remaining ether to a designated recipient.
- When a contract is destroyed, storage space is freed up in the blockchain as its code and data are removed.

```
selfdestruct(recipientAddress);
```

Function overloading

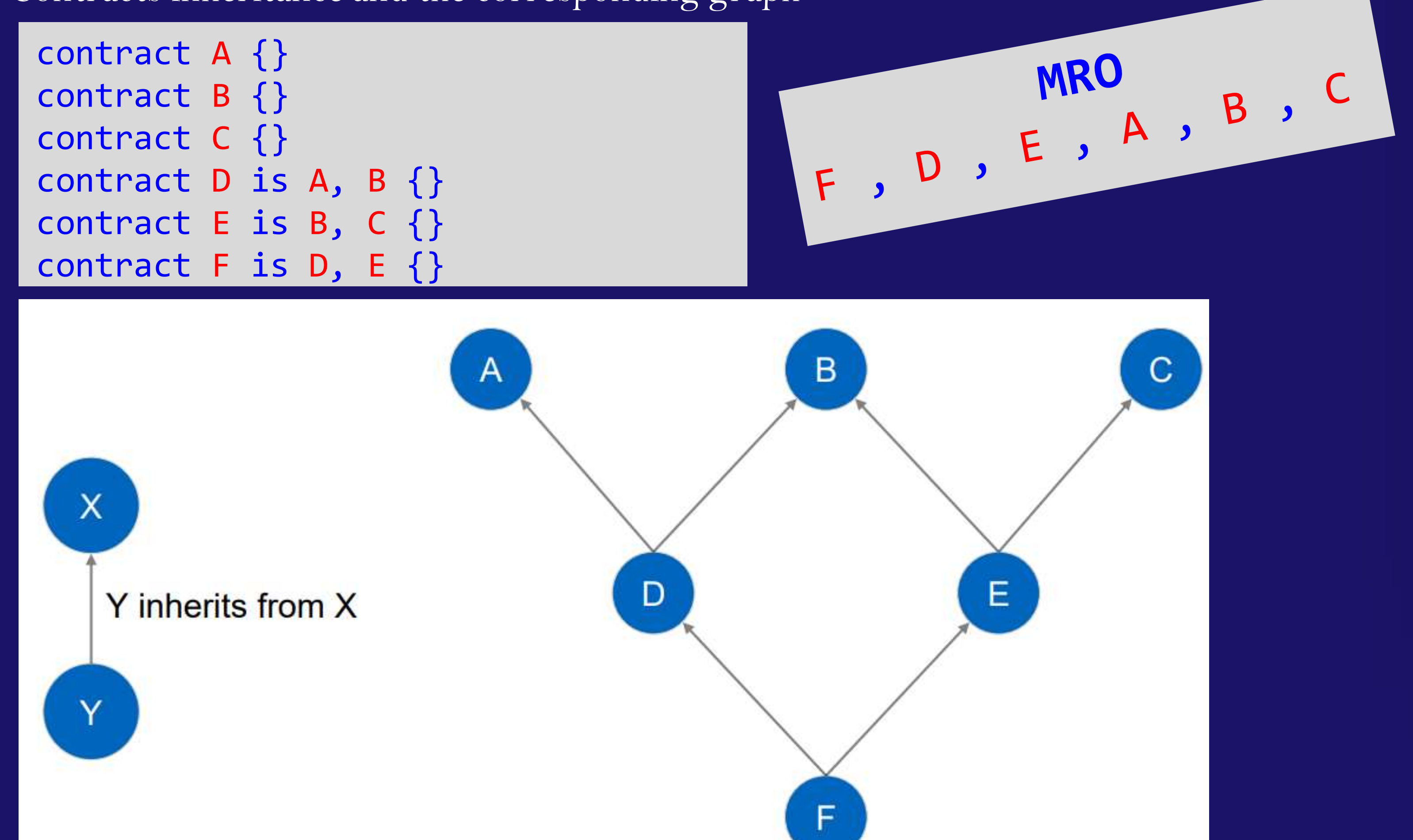
- Solidity supports the overload of functions,
- Multiple definition of the same function with a different signature,
- It can be helpful if a method needs to be adapted to certain situations.

```
function sendEther(uint amount) {
        require(this.balance >= amount);
        msg.sender.transfer(amount);
}

function sendEther(uint amount, address to) {
       require(this.balance >= amount);
        to.transfer(amount);
}
```

Inheritance

- Solidity supports multiple inheritance of contracts,
- Solidity uses, similar to Python, the C3 superclass linearization algorithm to define the Method Resolution Order (MRO or FRO) of the inherited functions,
- The keyword *super* references the next contract in the FRO : super(F) = D .
- Example : Contracts inheritance and the corresponding graph



Inheritance

```
contract A {
        function getNumber() returns (uint a) {
                return 13;
        }
}

contract B is A {
        function getNumber() returns (uint a) {
                return super.getNumber() + 1;
        }
}

contract C is A {
        function getNumber() returns (uint a) {
                return super.getNumber() + 2;
        }
}

contract F is C, B {
        function getNumber() returns (uint a) {
                return super.getNumber();
        }
}
```

What would happen if `F.getNumber()` is called?

The FRO is :

- F, C, B, A
- In F super will be resolved to C
- In C super will point to B
- In B super will point to A

`The final result is :`

13 + 1 + 2 = 16

Abstract contract

- Solidity supports abstract contracts. A contract is implicitly declared as abstract, if one or more functions are abstract. A function is considered abstract when it does not have a body.
- Abstract contracts cannot be compiled to bytecode. A contract that inherits from an abstract contract must implement and override all methods from the base contract.
- Abstract contracts offer a way to decouple the definition of a contract from its actual implementation

```
contract CarInsurance {
        function payMonthlyFee() returns (boolean result);
}
```

Interface

- an interface is a special type of contract that defines a set of functions that other contracts can implement,
- An interface is similar to an abstract class but is more restrictive. It is not allowed to define a constructor or variables in interface. Furthermore, interfaces cannot inherit from a contract or implement another interface,
- Interface can inherit from other interfaces
- In interface all declared functions must be external

```
interface Token {
        function transfer(address _to, uint256 _value) external;
        function balanceOf(address _owner) external view returns (uint256);
}

contract MyToken is Token {
    function transfer(address _to, uint256 _value) external {
        // Implement the transfer function here
    }

    function balanceOf(address _owner) external view returns (uint256) {
        // Implement the balanceOf function here
    }
}
```

Events : *event* , *emit*

- Events are a way to log that something has occurred,
- When a transaction completes (successfully or not), it produces a transaction receipt,
- The transaction receipt contains log entries that provide information about the actions that occurred during the execution of the transaction,
- Events are the Solidity high-level objects that are used to construct these logs,
- Events are especially useful for light clients and DApp, which can watch for specific events and report them to the user interface
- Event objects take arguments that are serialized and recorded in the transaction logs, in the blockchain,
- emit is a keyword used to trigger events

```
contract EventsExample {
        event OwnerChanged(address _oldOwner, address _newOwner);
        function transfer(address _newOwner) public {
                require(owner == msg.sender, "Sender not authorized");
                emit OwnerChanged(owner, _newOwner); owner = _newOwner;
        }
}
```

Events

transaction cost	43044 gas
execution cost	21580 gas
hash	0x306ba94b3681cc650cf62d55ae4a121aea240a5dd7077cb660c03f77a82ae537
input	0x82a...00064

decoded input	{
	"uint256 newBalance": "100"
	}

decoded output	{}

logs	[
	{
	"from": "0x692a70d2e424a56d2c6c27aa97d1a86395877b3a",
	"topic": "0x5f66d2a93b609bc6596b75c6dbb0e4f3f7cafd4b3b617157ff304d1076e58375",
	"event": "Update",
	"args": {
	"0": "0xCA35b7d915458EF540aDe6068dFe2F44E8fa733c",
	"1": "100",
	"_user": "0xCA35b7d915458EF540aDe6068dFe2F44E8fa733c",
	"_newBalance": "100",
	"length": 2
	}
	}
	]

value	0 wei