Ethereum Smart contracts development

With Javascript (2021)

Andrea Lisi, andrealisi.12lj@gmail.com
Part 1
Solidity overview

A brief summary of a Solidity smart contract
A smart contract is similar to a Java class.

It is composed by:
- Declaration
- A State (attributes)
- A list of functions (methods)
State variables determine the state of that smart contract

Solidity supports various data types:

- **Fixed length**
  - `bool`, `(u)int`, `bytes32`, `address`

- **Variable length**
  - `bytes`, `string`

- **array, mapping(key_type => value_type)**
Smart contracts: state

Array
- Fixed length or dynamic length, can be iterated over
- Removing an element requires a decision
  - Leave a blank hole, replace with last element (breaks ordering), shift elements (costly)

Mapping(key => value)
- All non-assigned values are Zero (false for bool, 0 for uint, etc)
- Support random access, it is not possible to iterate over the keys unless you keep a separate list of all the keys with significant value
Smart contracts: functions

Functions compose the code of the smart contract

Functions have labels that declare how they interact with the state:

- A **view** function **only reads** the state;
- A **pure** function does not read or write the state
- Otherwise, the function writes (and reads) the state
  - The state modification will be placed in a transaction
  - It will be written on the blockchain
  - Therefore, it costs a fee to the user
Smart contracts: functions

```solidity
uint public counter;

function increment() public {
    counter = counter + 1;
}

function getSquare() public view returns(uint) {
    return counter**2;
}

function computeSquareOf(uint _a) public pure returns(uint) {
    return _a**2;
}
```
Fees and gas

A function modifying the state writes data on the blockchain

- It requires a transaction

Each transaction costs a fee to the user

- The fee is proportional to the required amount of computation (EVM OPCODES)
- Each OPCODE has a costs named gas
Fees and gas

Each transaction costs a fee to the user

● Before each transaction, a user can set in their wallet:
  ○ The gas price: i.e. how much Ether they are willing to pay for each unit of gas
  ○ The gas limit: i.e. how many units of gas they are willing to consume for that transaction
Smart contracts: visibility

State variables and functions can have different visibilities

- **Private**
  - A private state variable or function is exposed only to the contract itself
- **Public**
  - A public function is exposed to other contracts; a public state is a shortcut that creates a getter function with the name of the variable
Smart contracts: visibility

State variables and functions can have different visibilities

- **Internal**
  - An internal state variable or function is exposed to child contracts and the contract itself

- **External**
  - (Only functions) An external function is exposed only to other contracts. They are more efficient with large inputs

  ■ **Warning:** `foo()` does not work; `this.foo()` does

Accounts

In Ethereum any entity (account) has associated

- An address: e.g. 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
- A balance in Ether greater or equal than 0

The two types of accounts are:

- **Contract Accounts**: are controlled by code, and a transaction activates its code
- **Externally Owned Accounts (EOA)**: are controlled by private keys and sign transactions
Solidity defines various global variables and functions

- Ether units: wei, gwei, szabo, ...
- Time units: seconds, minutes, ...
- Functions: keccak256, abi.encode, abi.decode, ...
- Transaction data: msg
  - msg.sender: the transaction sender (address)
  - msg.value: the transaction associated ETH (uint)
- ...

https://docs.soliditylang.org/en/v0.8.3/units-and-global-variables.html
A function can be labelled as **payable** if it *expects* to receive Ether

- Once received the Ether the contract’s balance is automatically increased, unless the transaction does not revert
- **msg.value** stores the received Ether (uint)

```solidity
function foo() public payable {
    address payer = msg.sender; // Who sent the Ether
    uint received = msg.value; // How much *in wei*
    uint current = address(this).balance; // The current balance of the contract
}
```
Smart contracts: receive Ether

If a smart contract receives plain Ether, i.e. a transaction to the contract does not invoke a function:

- Trigger the `receive` function (>= Solidity 0.6.*)

If a transaction invokes a function that does not match any of the functions exposed by the contract, or as before but `receive` is not implemented:

- Trigger the `fallback` function

As before, but neither `receive` nor `fallback` are implemented

- Throws exception
Smart contracts: receive Ether

contract Example {
    // “address payable” labels an address meant to receive ETH from this contract
    address payable known_receiver;
    function forward() public payable {
        known_receiver.transfer(msg.value);
    }

    // All of them have in their body at most 2300 units of gas of computation available if called by send() or transfer() (see next slide)
    receive() external payable {} // receive function
    fallback() external payable {} // fallback function Solidity >= 0.6.*
    function() public payable {} // fallback function Solidity < 0.6.*
}
Smart contracts: send Ether

If the contract has balance > 0, then it can send Ether as well

- Solutions that gives the receiver a gas limit of only 2300 units
  - `address.send(amount)` Send amount to `address`, returns True if everything goes well, False Otherwise
  - `address.transfer(amount)` Throws exception if it fails

- Solution with customizable gas limit
  - `address.call{options}(data bytes)` Returns True or False
  - `(bool result, ) = address.call{gas: 123123, value: msg.value}(""`;
Smart contracts: send Ether

Send/transfer: pros & cons

- A fixed gas limit prevents the receiver to execute too much code
  - It may consume too much gas to the original transaction sender
  - The receiver can execute malicious code, attempting an attack (e.g. reentrancy attack)
- Future updates to the gas associated to OPCODES (e.g. Istanbul fork) may break contracts already deployed working with limits of 2300 units of gas

https://consensys.net/diligence/blog/2019/09/stop-using-soliditys-transfer-now/
Smart contracts: events

It is possible to declare an event in Solidity similarly to a function, and it can be fired with the `emit` keyword.

- Events are placed in the transaction log, useful for client apps.
References

Solidity documentation V 0.8.3: https://docs.soliditylang.org/en/v0.8.3/index.html

Accounts: https://ethereum.org/en/whitepaper/#ethereum-accounts

Sending Ether:
https://vomtom.at/solidity-0-6-4-and-call-value-curly-brackets/

Best practices: https://consensys.github.io/smart-contract-best-practices/

Smart contracts: development

It is possible to implement Ethereum smart contracts with the Solidity programming language.

Smart contracts can be developed and executed within:

- The browser IDE Remix, https://remix.ethereum.org/
- The CLI tool Truffle, https://www.trufflesuite.com/truffle
Extra

Advanced Solidity functionalities
Abi functions

The contract Abi (Application Binary Interface) is the standard contract-to-contract communication in Ethereum, to encode and decode functions, parameters, etc in known data, in bytes, to:

- Call a function of an external contract;
- Pass input arguments;
- And more.

https://docs.soliditylang.org/en/v0.8.4/abi-spec.html
https://docs.soliditylang.org/en/v0.8.4/units-and-global-variables.html#abi-encoding-and-decoding-functions
contract Decoder {

    function encodeArgs(uint _a, bool _b) public pure returns(bytes memory) {
        bytes memory data = abi.encode(_a, _b);
        return data;
    }

    function decodeArgs(bytes memory data) public pure returns(uint, bool) {
        (uint _a, bool _b) = abi.decode(data, (uint, bool));
        return (_a, _b);
    }

}
Abi functions

```solidity
contract HashContract {

    function encodeArgs(uint _a, bool _b) public pure returns(bytes memory) {
        bytes memory data = abi.encode(_a, _b);
        return data;
    }

    // The hash of arbitrary data can be computed with bytes32 hash =
    // keccak256(abi.encode(param1, param2, ...));

    function computeHash(bytes memory data) public pure returns(bytes32) {
        bytes32 hash = keccak256(data);
        return hash;
    }
}
```
Calling contract functions

How to call a function of another smart contract?

- If you have the source code, you can **import** it on your Solidity file. Therefore, you have visibility of the contract’s type and functions, and the compiler understands them.

- If you **DO NOT** have the source code, you can use a low-level **call** to a function of a smart contract with the function’s **selector** as input:
  - The selector are the first 4 bytes of the hash of the function signature, i.e. `functionName(param1, param2, ...)`. 
Calling contract functions: import

```solidity
import "External.sol"

contract Caller {
    External contractExternal;
    constructor(address _c) public {
        contractExternal = External(_c);
    }
    function increment() public {
        contractExternal.increment();
    }
    function increment(uint _a) public {
        contractExternal.increment(_a);
    }
}
```

```solidity
contract External {
    uint public c;
    function increment() public {
        c = c + 1;
    }
    function increment(uint _a) public {
        c = c + _a;
    }
}
```
Calling contract functions: `.call()`

```solidity
contract External {

    uint public c;

    function increment() public {
        c = c + 1;
    }

    function increment(uint _a) public {
        c = c + _a;
    }
}

c合同 Caller {

    address contractExternal;

    constructor(address _c) public {
        contractExternal = _c;
    }

    function increment() public {
        bytes4 selector = bytes4(keccak256("increment()"));
        bytes memory data = abi.encodeWithSelector(selector);
        (bool outcome, ) = contractExternal.call(data);
        if(!outcome) revert();
    }
```
Calling contract functions: `.call()`

```solidity
contract External {
    uint public c;

    function increment() public {
        c = c + 1;
    }

    function increment(uint _a) public {
        c = c + _a;
    }
}

contract Caller {
    address contractExternal;

    constructor(address _c) public {
        contractExternal = _c;
    }

    function increment(uint _a) public {
        bytes4 selector = bytes4(keccak256("increment(uint)"));
        bytes memory data = abi.encodeWithSelector(selector, _a);
        (bool outcome, ) = contractExternal.call(data);
        if(!outcome) revert();
    }
}
```