📖 Solidity deep dive: Language specs

- Solidity
- Contratti e istanze
- Tipi
- Modificatori di visibilità
- msg, block, this
- modifier
- Esempi di smart contract
- Q&A

Solidity

Solidity è un linguaggio **contract-oriented** ad alto livello che viene compilato dal suo compilatore in EVM bytecode.

È influenzato da C++, Javascript e Python.

È un linguaggio **statically-typed**, supporta l'**ereditarietà** e consente la definizione di tipi **user-defined**.

La versione più recente di Solidity è la v0.8.12.

Smart contract e istanze

È possibile pensare ad uno smart contract (.sol) come alla ***classe*** che definisce le ***istanze*** (bytecode deployato) che derivano da esso.

È infatti possibile avere su blockchain contratti che derivano dalla stessa definizione (.sol) a indirizzi diversi.

Ogni contratto ha un constructor che viene eseguito appena lo stesso viene deployato.

Layout di uno smart contract

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.12;

import “filename”;

contract A {

  // commento

  /*
  commento
  multilinea
  */

}
```

Layout di uno smart contract - pragma solidity e SPDX License

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.12;
```

In un file solidity possiamo trovare subito due componenti:

- SPDX License: specifica la licenza associata allo smart contract secondo lo standard SPDX (Software Package Data Exchange)

- pragma solidity *: specifica la versione del compilatore con cui compilare il file corrente. La gestione delle versioni segue le regole di NPM (^, >=, <=, etc.)

Layout di uno smart contract - import di altri file

```
import “filename”;
```

L’import di un file e degli elementi definito in esso può avvenire in vari modi:

- import “filename”;
- import * as symbolName from “filename”;
- import {symbol1 as alias, symbol2} from “filename”;

Layout di uno smart contract - commenti

```
// commento

/*
commento
multilinea
*/
```

Come in altri linguaggi i commenti possono essere estesi su una sola linea (//) o su più linee (/**/)

Tipi

- bool, valore booleano (true o false)
- int/uint, valore intero signed o unsigned (int8, uint8, .., int256, uint256)
- string, stringa UTF-8 di lunghezza variabile
- address, rappresentazione di un indirizzo Ethereum (20 byte)
  - address(x).balance, ritorna il saldo dell'indirizzo x
  - address(x).transfer(y), trasferisce y wei all'indirizzo x
- type[], array dinamico di tipo type

Tipi di dato

```
enum State {Open, Closed}

struct Auction {
  string product;
  State state;
}

mapping(uint => Auction) auctions;
```

- `struct`, tipo di dato strutturato
- `enum`, enumerazione (interi, come in altri linguaggi)
- `mapping(x => y)`, hash table con chiave di tipo x e valore di tipo y

Modificatori di visibilità

- external, il metodo può essere chiamato attraverso una transazione da un EOA o da un altro contratto (non dal contratto stesso)

- public, il metodo può essere chiamato internamente al contratto o da contratti esterni (via messaggi)

- internal, il metodo può essere chiamato soltanto dal contratto stesso o da contratti che ereditano da esso

- private, il metodo può essere chiamato solo dal contratto in cui è definita

Oltre a questi modificatori di visibilità abbiamo altri due modificatori per i metodi:

- pure, indica un metodo che non fa uso dello stato di un contratto

- view, indica un metodo che non modifica lo stato del contratto

contract, function

```
contract Auction {

  function settle(uint256 price) external returns (bool) {}

}
```

In solidity un contratto viene definito con la keyword `contract`.

Il metodo di un contratto viene definito con la keyword `function`. Il tipo di ritorno si definisce con la keyword `returns`.

Faucet

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.12;


contract Faucet {

  function withdraw(uint256 amount) external {
    require(amount <= 0.1 ether);
    msg.sender.transfer(amount);
  }

  receive() external payable { }
}
```

require

La keyword require può essere usata per fermare l’esecuzione del contratto (e di fatto far fallire la transazione) nel casi in cui una certa condizione non venga rispettata.

È possibile indicare un messaggio di errore nel caso in cui la condizione specificata in require non venga soddisfatta.

In Solidity esistono altri modi per fermare l’esecuzione della transazione e sollevare un errore/eccezione.

`msg.sender`, `msg.value`

Quando un contratto viene eseguito è possibile accedere al messaggio che lo ha invocato grazie alla variabile globale `msg`.

`msg.value` riporta il valore (in wei) trasportato dal messaggio.

`msg.sender` riporta il mittente del messaggio (un contratto o un EOA che ha inviato una transazione).

payable

Un metodo può accettare un messaggio con `msg.value > 0` soltanto nel caso in cui esso sia dichiarato payable.

È necessario dichiarare un `address payable` nel caso in cui si voglia chiamare il metodo `transfer` su di esso.

Uno smart contract può accettare semplici trasferimenti di ETH dichiarando il metodo `receive`.

Possibili modifiche a Faucet

- **Aggiunta di un limite temporale per i prelievi di ETH**
- Possibilità di modifica della soglia (0.1 ether)
- Ritiro di tutti gli ETH contenuti nel contratto
- **Controllo dell’accesso ad alcune funzioni**
- che altro?

block

Oltre a msg in Solidity è possibile accedere anche alla variabile globale block che contiene varie informazioni:

- block.timestamp riporta il timestamp del blocco in cui la transazione viene accettata

- block.number riporta il numero del blocco in cui la transazione viene accettata

- block.coinbase riporta l'indirizzo del miner che ha minato il blocco in cui la transazione viene accettata

- block.basefee, block.chainid, block.gaslimit e altri

Gestione del tempo in Solidity

In Solidity è possibile indicare le unità di tempo attraverso varie keyword (seconds, minutes, hours, days, weeks).

È buona norma non dipendere dalla precisione del timestamp del blocco se si vuole essere precisi nella considerazione del tempo poiché potrebbe essere manipolato (+/- 15 minuti) oppure un blocco potrebbe richiedere più o meno tempo ad essere minato rispetto ai canonici 13.5 secondi.

FaucetV2

```sol
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.12;


contract FaucetV2 {
  mapping(address => uint64) public withdraws;

  function withdraw(uint256 amount) external {
    require(amount <= 0.1 ether);
    require(address(this).balance >= amount);

    uint64 last = withdraws[msg.sender];
    require(block.timestamp - last > 1 days);

    withdraws[msg.sender] = block.timestamp;
    msg.sender.transfer(amount);
  }

  receive() external payable { }
}
```

this

La keyword this rappresenta l’istanza in esecuzione del contratto.

È convertibile al tipo address con address(this) per utilizzare balance.

Ownable

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.12;


contract Ownable {
  address public owner;

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

    _;
  }

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

modifier

Un modifier è un metodo che può essere definito *decoratore* dell'esecuzione di un altro metodo.

Con **_;** indichiamo la partenza del metodo che stiamo decorando.

Dopo l'istruzione **_;** possiamo continuare con altre operazioni all'interno del modifier.