PER ALTRI APPUNTI CONSULTARE IL SITO:

https://luigi-v.github.io/Appunti_Universita/

Per altri appunti consultare il sito: *https://luigi-v.github.io/Appunti_Universita/*

By *Vollono Luigi*

SOLIDITY

Solidity è un linguaggio di programmazione di alto livello orientato al contratto per l'implementazione di smart contract. Solidity è fortemente influenzato da C++, Python e JavaScript ed è stato progettato per indirizzare la ***Ethereum Virtual Machine*** (***EVM***). Solidity è *tipizzato staticamente*, supporta l'ereditarietà, le librerie e il linguaggio di programmazione di tipi complessi definiti dall'utente.

ETHEREUM:

Ethereum è un sistema decentralizzato, ad es. piattaforma blockchain che esegue smart contract, ovvero applicazioni che funzionano esattamente come programmato senza alcuna possibilità di tempi di inattività, codice nascosto, frode o interferenza di terze parti. Uno ***smart contract*** è un protocollo informatico inteso a facilitare, verificare o far rispettare digitalmente la negoziazione o l'esecuzione di un contratto. Gli smart contract consentono l'esecuzione di transazioni credibili senza terze parti. Queste transazioni sono tracciabili e irreversibili.

La macchina virtuale Ethereum, nota anche come ***EVM***, è l'ambiente di runtime per smart contract in Ethereum. La macchina virtuale Ethereum si concentra sulla fornitura di sicurezza e sull'esecuzione di codice non attendibile da parte dei computer di tutto il mondo.

L'EVM è specializzato nella prevenzione degli attacchi ***Denial-of-service*** e garantisce che i programmi non abbiano accesso allo stato dell'altro, assicurando che la comunicazione possa essere stabilita senza alcuna potenziale interferenza.

La macchina virtuale Ethereum è stata progettata per fungere da ambiente di runtime per smart contract basati su Ethereum.

SINTASSI DI BASE:

Un file sorgente di Solidity può contenere un numero qualsiasi di definizioni di contratto, direttive di importazione e direttive di pragma.

```
pragma solidity >=0.4.0 <0.6.0;

contract SimpleStorage {
   uint storedData;
   function set(uint x) public {
      storedData = x;
   }
   function get() public view returns (uint) {
      return storedData;
   }
}
```

- ***pragma***: La prima riga è una direttiva ***pragma*** che dice che il codice sorgente è scritto per Solidity versione 0.4.0 o qualcosa di più recente che non interrompe le funzionalità fino alla versione 0.6.0, quest'ultima esclusa.
  Una direttiva pragma è sempre locale rispetto a un file sorgente e se si importa un altro file, il pragma da quel file non si applicherà automaticamente al file importato.
  Un pragma per un file che non verrà compilato prima della versione 0.4.0 e non funzionerà anche su un compilatore a partire dalla versione 0.5.0 verrà scritto come segue:

```
pragma solidity ^0.4.0;
```

- ***contract***: Un ***contratto Solidity*** è una raccolta di codice (le sue funzioni) e dati (il suo stato) che risiede a un indirizzo specifico sulla blockchain di Ethereum. La riga `uint storedData` dichiara una variabile di stato denominata storedData di tipo uint e le funzioni set e get possono essere utilizzate per modificare o recuperare il valore della variabile.
- ***import:*** Sebbene l'esempio sopra non disponga di un'istruzione di importazione, Solidity supporta istruzioni di importazione molto simili a quelle disponibili in JavaScript.
  La seguente istruzione importa tutti i simboli globali da "filename":
  L'esempio seguente crea un nuovo simbolo globale symbolName i cui membri sono tutti i simboli globali di "filename":

```
import "filename";
```

```
import * as symbolName from "filename";
```

COMMENTI:

Solidity supporta commenti sia in stile C che in stile C++, quindi:

- Qualsiasi testo tra // e la fine di una riga viene trattato come un commento e viene ignorato da Solidity Compiler.
- Qualsiasi testo compreso tra i caratteri /* e */ viene trattato come un commento. Questo può estendersi su più righe.

```
function getResult() public view returns(uint){
   // This is a comment. It is similar to
   // comments in C++
}
```

TIPI DI DATI:

Solidity offre al programmatore un assortimento di tipi di dati incorporati e definiti dall'utente. La seguente tabella elenca sette tipi di dati C++ di base:

TIPO	PAROLA CHIAVE	VALORE
Boolean	bool	vero / falso
Integer	int/uint	Interi con segno e senza segno di varie dimensioni.
Integer	int8 to int256	Int con segno da 8 bit a 256 bit. int256 è uguale a int.
Integer	uint8 to uint256	Int senza segno da 8 bit a 256 bit. uint256 è uguale a uint.
Fixed Point Numbers	fixed/unfixed	Numeri a virgola fissa firmati e non firmati di varie dimensioni.
Fixed Point Numbers	fixed/unfixed	Numeri a virgola fissa firmati e non firmati di varie dimensioni.
Fixed Point Numbers	fixedMxN	Numero a virgola fissa con segno dove M rappresenta il numero di bit presi per tipo e N rappresenta i punti decimali. M dovrebbe essere divisibile per 8 e va da 8 a 256. N può essere da 0 a 80. fixed è uguale a fixed128x18.
Fixed Point Numbers	ufixedMxN	Numero a virgola fissa senza segno dove M rappresenta il numero di bit presi dal tipo e N rappresenta i punti decimali. M dovrebbe essere divisibile per 8 e va da 8 a 256. N può essere da 0 a 80. ufixed è uguale a ufixed128x18.

NOTA: è anche possibile rappresentare i numeri a virgola fissa con segno e senza segno come fixedMxN/ufixedMxN dove M rappresenta il numero di bit presi per tipo e N rappresenta i punti decimali. M dovrebbe essere divisibile per 8 e va da 8 a 256. N può essere da 0 a 80.

TIPO DI DATO ADDRESS:

Il tipo di dato ***address*** contiene il valore di 20 byte che rappresenta la dimensione di un indirizzo Ethereum.

Un indirizzo può essere utilizzato per ottenere il saldo utilizzando il metodo `.balance` e può essere utilizzato per trasferire il saldo a un altro indirizzo utilizzando il metodo `.transfer`.

```
address x = 0x212;
address myAddress = this;
if (x.balance < 10 && myAddress.balance >= 10) x.transfer(10);
```

VARIABILI:

Solidity supporta tre tipi di variabili:

- ***Variabili di stato*** - Variabili i cui valori sono archiviati in modo *permanente* in una memoria del contratto;
- ***Variabili locali*** - Variabili i cui valori sono presenti fino all'esecuzione della funzione;
- ***Variabili globali*** - Esistono variabili speciali nello spazio dei nomi globale utilizzato per ottenere informazioni sulla blockchain.

Solidity è un linguaggio tipizzato staticamente, il che significa che lo stato o il tipo di variabile locale deve essere specificato durante la dichiarazione.

Ogni variabile dichiarata ha sempre un valore predefinito in base al suo tipo. Non esiste il concetto di "undefined " o "null".

VARIABILE DI STATO:

Variabili i cui valori sono archiviati in modo permanente in un archivio del contratto.

```
pragma solidity ^0.5.0;
contract SolidityTest {
   uint storedData;      // State variable
   constructor() public {
      storedData = 10;   // Using State variable
   }
}
```

VARIABILE LOCALE:

Variabili i cui valori sono disponibili solo all'interno di una funzione in cui è definita. I parametri della funzione sono sempre locali a quella funzione.

```
pragma solidity ^0.5.0;
contract SolidityTest {
   uint storedData; // State variable
   constructor() public {
      storedData = 10;
   }
   function getResult() public view returns(uint){
      uint a = 1; // local variable
      uint b = 2;
      uint result = a + b;
      return result; //access the local variable
   }
}
```

VARIABILE GLOBALE:

Queste sono ***variabili speciali*** che esistono nell'area di lavoro globale e forniscono informazioni sulla blockchain e sulle proprietà delle transazioni.

NOME	VALORI DI RITORNO
blockhash(uint blockNumber) returns (bytes32)	Hash del blocco dato: funziona solo per i 256 blocchi più recenti, esclusi quelli correnti.
block.coinbase (address payable)	Indirizzo del miner del blocco attuale.
block.difficulty (uint)	Difficoltà del blocco attuale.
block.gaslimit (uint)	Limite gas del blocco corrente.
block.number (uint)	Numero di blocco attuale.
block.timestamp (uint)	Timestamp del blocco corrente in secondi dall'epoca unix.
gasleft() returns (uint256)	Gas rimanente.
msg.data (bytes calldata)	calldata completi.
msg.sender (address payable)	Mittente del messaggio (chiamante attuale).
msg.sig (bytes4)	Primi quattro byte del calldata (identificatore di funzione).
msg.value (uint)	Numero di Wei inviati con il messaggio.
now (uint)	Data e ora del blocco corrente.
tx.gasprice (uint)	Prezzo del gas della transazione.
tx.origin (address payable)	Mittente della transazione

SCOPE DELLE VARIABILI:

Lo ***scope delle variabili*** locali è limitato alla funzione in cui sono definite, ma le *variabili di stato* possono avere tre tipi di scope:

- ***Pubblico*** - È possibile accedere alle variabili di stato pubbliche internamente e tramite messaggi. Per una variabile di stato pubblica, viene generata una funzione getter automatica;
- ***Interna*** – È possibile accedere alle variabili di stato interne solo internamente dal contratto in corso o dal contratto che ne deriva senza utilizzarlo;
- ***Privato*** - Le variabili di stato private sono accessibili solo internamente dal contratto corrente, non sono definite nel contratto derivato da esso.

```
pragma solidity ^0.5.0;
contract C {
   uint public data = 30;
   uint internal iData= 10;

   function x() public returns (uint) {
      data = 3; // internal access
      return data;
   }
}
contract Caller {
   C c = new C();
   function f() public view returns (uint) {
      return c.data(); //external access
   }
}
contract D is C {
   function y() public returns (uint) {
      iData = 3; // internal access
      return iData;
   }
   function getResult() public view returns(uint){
      uint a = 1; // local variable
      uint b = 2;
      uint result = a + b;
      return storedData; //access the state variable
   }
}
```

OPERATORI:

Solidity supporta i seguenti tipi di operatori:

- ***Operatori aritmetici***: +, -, *, /, %, ++, --;
- ***Operatori di confronto***: ==, !=, >, <, >=, =<;
- ***Operatori logici (o relazionali)***: &&, ||, !;
- ***Operatori bit a bit***: & (AND), | (OT), ^ (XOR), ~ (NEGATO), << (SHIFT SINISTRO), >> (SHIFT DESTRO), >>> (SHIFT A DESTRA CON ZERO);
- ***Operatori di assegnazione***: =, +=, -=, /=, %=;
- ***Operatori condizionali (o ternari)***: valuta prima un'espressione per un valore vero o falso e quindi esegue una delle due istruzioni fornite a seconda del risultato della valutazione. Sintassi: **? : (Condizionale)**.

LOOP:

Solidity supporta tutti i loop:

- ***While***:

```
while (expression) {
  Statement(s) to be executed if expression is true
}
```

- ***do…while***:

```
do {
  Statement(s) to be executed;
} while (expression);
```

- ***For***:

```
for (initialization; test condition; iteration statement) {
  Statement(s) to be executed if test condition is true
}
```

- ***Control loop***:

```
if(n == 5){
        continue;
 }
if(j==0){
        break;
}
```

CONDIZIONI:

Solidità supporta le seguenti forme di istruzione ***if..else***:

```
if (expression 1) {
  Statement(s) to be executed if expression 1 is true
} else if (expression 2) {
  Statement(s) to be executed if expression 2 is true
} else {
  Statement(s) to be executed if no expression is true
}
```

STRINGHE:

Solidity supporta il valore letterale ***String*** utilizzando sia le virgolette doppie (") che le virgolette singole ('). Fornisce stringa come tipo di dati per dichiarare una variabile di tipo String.

```
contract SolidityTest {
   string data = "test";
}
```

Nell'esempio sopra, "test" è un letterale stringa e data è una variabile stringa. Il modo migliore consiste nell'utilizzare i tipi di byte anziché String poiché l'operazione di stringa richiede più gas rispetto all'operazione di byte.

Solidity fornisce una conversione integrata tra byte in stringa e viceversa. In Solidity possiamo assegnare facilmente il letterale String a una variabile di tipo byte32. Solidity lo considera come un byte32 letterale.

```
contract SolidityTest {
   bytes32 data = "test";
}
```

I byte possono essere convertiti in String utilizzando il costruttore string().

```
bytes memory bstr = new bytes(10);
string message = string(bstr);
```

ARRAY:

In Solidity, un ***array*** può essere di dimensione fissa in fase di compilazione o di dimensione dinamica. Per l'array di archiviazione, può avere anche diversi tipi di elementi. In caso di array di memoria, il tipo di elemento non può essere mappato e nel caso in cui debba essere utilizzato come parametro di funzione, il tipo di elemento dovrebbe essere un tipo ABI.

Tutti gli array sono costituiti da locazioni di memoria contigue. L'indirizzo più basso corrisponde al primo elemento e l'indirizzo più alto all'ultimo.

Per dichiarare un array di dimensione fissa in Solidity, il programmatore specifica il tipo degli elementi e il numero di elementi richiesti da un array come segue:

```
type arrayName [ arraySize ];
```

Questo è chiamato array a dimensione singola. L'arraySize deve essere una costante intera maggiore di zero e il tipo può essere qualsiasi tipo di dati Solidity valido.

Ad esempio, per dichiarare un ***array di 10*** elementi chiamato balance di type uint, usa questa istruzione:

```
uint balance[10];
```

Per dichiarare un array di ***dimensione dinamica*** in Solidity, specifica il tipo degli elementi come segue:

```
type[] arrayName;
```

È possibile inizializzare gli elementi dell'array Solidity uno per uno o utilizzando una singola istruzione come segue:

```
uint balance[3] = [1, 2, 3];
```

Se si omette la dimensione dell'array, viene creato un array abbastanza grande da contenere l'inizializzazione:

```
uint balance[] = [1, 2, 3];
```

Per ***assegnare*** un valore ad una specifica posizione del vettore, bisogna specificare la posizione tra parentesi:

```
balance[2] = 5;
```

Per ***accedere*** a un elemento bisogna indicizzare l'array:

```
uint salary = balance[2];
```

Gli ***array di memoria*** dinamici vengono creati utilizzando la parola chiave new.

Gli array, in Solidity, hanno due membri:

- ***length***, restituisce la dimensione dell'array. length può essere utilizzato per modificare la dimensione dell'array dinamico impostandolo;
- ***push***, consente di aggiungere un elemento a un array di archiviazione dinamico alla fine. Restituisce la nuova lunghezza dell'array.

```
contract test {
   function testArray() public pure{
      uint len = 7;

      //dynamic array
      uint[] memory a = new uint[](7);

      //bytes is same as byte[]
      bytes memory b = new bytes(len);

      assert(a.length == 7);
      assert(b.length == len);

      //access array variable
      a[6] = 8;

      //test array variable
      assert(a[6] == 8);

      //static array
      uint[3] memory c = [uint(1) , 2, 3];
      assert(c.length == 3);
   }
}
```

STRUCT:

Per definire una ***struct***, è necessario utilizzare la parola chiave struct.
La parola chiave struct definisce un nuovo tipo di dati, con più di un membro.
Il formato dell'istruzione struct è il seguente:

```
struct struct_name {
  type1 type_name_1;
  ...
}
```

Per ***accedere*** a qualsiasi membro di una struttura, utilizziamo l'operatore di accesso ai membri (.). L'operatore di accesso al membro è codificato come un punto tra il nome della variabile della struttura e il membro della struttura a cui si desidera accedere. Utilizzeresti la struttura per definire le variabili di tipo struttura.
L'esempio seguente mostra come utilizzare una struttura in un programma.

```
contract test {
   struct Book {
      string title;
      string author;
      uint book_id;
   }
   Book book;

   function setBook() public {
      book = Book('Learn Java', 'TP', 1);
   }
   function getBookId() public view returns (uint) {
      return book.book_id;
   }
}
```

MAPPING:

La ***mappatura*** è un tipo di riferimento come array e strutture. La sintassi per dichiarare un tipo di mappatura: `mapping(_KeyType => _ValueType)`
In cui si ha:

- ***_KeyType***, può essere qualsiasi tipo incorporato più byte e stringa. Non sono consentiti tipi di riferimento o oggetti complessi;
- ***_ValueType***, può essere di qualsiasi tipo.

La mappatura può avere solo un tipo di archiviazione e viene generalmente utilizzata per le variabili di stato. La mappatura può essere contrassegnata come pubblica. Solidity crea automaticamente getter per questo.

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

   function updateBalance(uint newBalance) public {
      balances[msg.sender] = newBalance;
   }
}
contract Updater {
   function updateBalance() public returns (uint) {
      LedgerBalance ledgerBalance = new LedgerBalance();
      ledgerBalance.updateBalance(10);
      return ledgerBalance.balances(address(this));
   }
}
```

CONVERSIONI:

Solidity consente la ***conversione implicita*** ed ***esplicita***. Il compilatore Solidity consente la conversione implicita tra due tipi di dati purché non sia possibile alcuna conversione implicita e non vi sia perdita di informazioni. Ad esempio, uint8 è convertibile in uint16 ma int8 è convertibile in uint256 poiché int8 può contenere un valore negativo non consentito in uint256.

Possiamo convertire esplicitamente un tipo di dati in un altro usando la sintassi del costruttore:

```
int8 y = -3;
uint x = uint(y);
//Adesso x = 0xfffff..fd == rappresentazione a
//due complementi di -3 in formato a 256 bit.
```

La conversione in caratteri più piccoli costa bit di ordine superiore:

```
uint32 a = 0x12345678;
uint16 b = uint16(a); // b = 0x5678
```

La conversione al tipo più alto aggiunge bit di riempimento a sinistra:

```
uint16 a = 0x1234;
uint32 b = uint32(a); // b = 0x00001234
```

La conversione in byte più piccoli costa dati di ordine superiore:

```
bytes2 a = 0x1234;
bytes1 b = bytes1(a); // b = 0x12
```

La conversione in byte più grandi aggiunge bit di riempimento a destra:

```
bytes2 a = 0x1234;
bytes4 b = bytes4(a); // b = 0x12340000
```

La conversione tra byte di dimensione fissa e int è possibile solo quando entrambi hanno la stessa dimensione:

```
bytes2 a = 0x1234;
uint32 b = uint16(a);         // b = 0x00001234
uint32 c = uint32(bytes4(a)); // c = 0x12340000
uint8 d = uint8(uint16(a));   // d = 0x34
uint8 e = uint8(bytes1(a));   // e = 0x12
```

I numeri esadecimali possono essere assegnati a qualsiasi tipo intero se non è necessario il troncamento:

```
uint8 a = 12;        // no error
uint32 b = 1234;     // no error
uint16 c = 0x123456; // error, as truncation required to 0x3456
```

ETHER UNITS:

In Solidity possiamo usare ***wei***, ***finney***, ***szabo*** o ***ether*** come suffisso di un letterale da usare per convertire varie denominazioni basate su ether. L'unità più bassa è wei e 1e12 rappresenta 1 x 10^{12}.

```
assert(1 wei == 1);
assert(1 szabo == 1e12);
assert(1 finney == 1e15);
assert(1 ether == 1e18);
assert(2 ether == 2000 fenny);
```

TIME UNITS:

Simile alla valuta, Solidity ha ***unità di tempo*** in cui l'unità più bassa è il secondo e possiamo usare secondi, minuti, ore, giorni e settimane come suffisso per indicare il tempo.

```
assert(1 seconds == 1);
assert(1 minutes == 60 seconds);
assert(1 hours == 60 minutes);
assert(1 day == 24 hours);
assert(1 week == 7 days);
```

FUNZIONI:

Il modo più comune per definire una ***funzione*** in Solidity è utilizzare la parola chiave ***function***, seguita da un nome di funzione univoco, un elenco di parametri (che potrebbe essere vuoto) e un blocco di istruzioni racchiuso tra parentesi graffe.

La sintassi di base è mostrata qui:

```
function function-name(parameter-list) scope returns() {
  //statements
}
```

Per invocare una funzione da qualche parte più avanti nel contratto, basta scrivere il nome di quella funzione in seguito.

C'è la possibilità di passare parametri diversi mentre si chiama una funzione. Questi parametri passati possono essere catturati all'interno della funzione e qualsiasi manipolazione può essere eseguita su quei parametri. Una funzione può accettare più parametri separati da virgole.

```
contract SolidityTest {
   constructor() public{
   }
   function getResult() public view returns(string memory){
      uint a = 1;
      uint b = 2;
      uint result = a + b;
      return integerToString(result);
   }
   function integerToString(uint _i) internal pure returns (string memory) {
      uint j = _i;
      uint len;
      while (j != 0) {
         len++;
         j /= 10;
      }
      bytes memory bstr = new bytes(len);
      uint k = len - 1;
      while (_i != 0) {
         bstr[k--] = byte(uint8(48 + _i % 10));
         _i /= 10;
      }
      return string(bstr);//access local variable
   }
}
```

Una funzione Solidity può avere un'istruzione return facoltativa. Questo è necessario se vuoi restituire un valore da una funzione. Questa istruzione dovrebbe essere l'ultima istruzione in una funzione.

In Solidity, una funzione può restituire anche più valori.

```
contract Test {
   function getResult() public view returns(uint product, uint sum){
      uint a = 1; // local variable
      uint b = 2;
      product = a * b;
      sum = a + b;
      //alternative return statement to return
      //multiple values
      //return(a*b, a+b);
   }
}
```

MODIFICATORI DI FUNZIONI:

I ***modificatori*** di funzione vengono utilizzati per modificare il comportamento di una funzione. Ad esempio, per aggiungere un prerequisito.

NOTA: Funzioni e indirizzi dichiarati ***payable*** possono ricevere ether nel contratto.

Il corpo della funzione è inserito dove il simbolo speciale "_;" appare nella definizione di un modificatore. Quindi, se la condizione del modificatore è soddisfatta durante la chiamata a questa funzione, la funzione viene eseguita, e in caso contrario viene generata un'eccezione.

```
contract Owner {
   address owner;
   constructor() public {
      owner = msg.sender;
   }
   modifier onlyOwner {
      require(msg.sender == owner);
      _;
   }
   modifier costs(uint price) {
      if (msg.value >= price) {
         _;
      }
   }
}
```

Il nome del modificatore sulla funzione viene specificato prima della parentesi "{" della funzione dove deve essere applicato.

```
contract Register is Owner {
   mapping (address => bool) registeredAddresses;
   uint price;
   constructor(uint initialPrice) public { price = initialPrice; }

   function register() public payable costs(price) {
      registeredAddresses[msg.sender] = true;
   }
   function changePrice(uint _price) public onlyOwner {
      price = _price;
   }
}
```

FUNZIONI VIEW:

Le ***funzioni view*** assicurano che non modificheranno lo stato. Una funzione qualsiasi può essere dichiarata come vista. Le seguenti istruzioni, se presenti nella funzione, sono considerate come modifica dello stato e il compilatore genererà un avviso in tali casi:

- Modificare le variabili di stato;
- Emissione di eventi;
- Creazione di altri contratti;
- Usando selfdestruction;
- Invio di Ether tramite chiamate;
- Chiamare qualsiasi funzione che non sia contrassegnata come view o pure;
- Utilizzo di chiamate di basso livello;
- Utilizzo di un assembly in linea contenente determinati codici operativi.

Il metodo Getter è per impostazione predefinita una funzioni view.

```
contract Test {
   function getResult() public view returns(uint product, uint sum){
      uint a = 1; // local variable
      uint b = 2;
      product = a * b;
      sum = a + b;
   }
}
```

```
0: uint256: product 2
1: uint256: sum 3
```

FUNZIONI PURE:

Le ***funzioni pure*** assicurano che non leggano o modifichino lo stato. Una funzione qualsiasi può essere dichiarata pure. Le seguenti istruzioni, se presenti nella funzione, sono considerate come lettura dello stato e il compilatore genererà un avviso in tali casi:

- Lettura delle variabili di stato;
- Accesso a address(this).balance o <address>.balance;
- Accedere a una qualsiasi delle variabili speciali di block, tx, msg (è possibile leggere msg.sig e msg.data);
- Chiamare qualsiasi funzione non contrassegnata come pure;
- Utilizzo di un assembly in linea che contiene determinati codici operativi.

NOTA: Le funzioni pure possono utilizzare le funzioni revert() e require() per ripristinare potenziali cambiamenti di stato se si verifica un errore.

```
contract Test {
   function getResult() public pure returns(uint product, uint sum){
      uint a = 1; // local variable
      uint b = 2;
      product = a * b;
      sum = a + b;
   }
}
```

```
0: uint256: product 2
1: uint256: sum 3
```

FUNZIONI FALLBACK:

La ***funzione di fallback*** è una funzione speciale disponibile per un contratto. Ha le seguenti caratteristiche:

- Viene chiamato quando viene chiamata una funzione inesistente sul contratto;
- È necessario che sia contrassegnato come ***external***;
- Non ha nome;
- Non ha argomenti;
- Non può restituire nulla;
- Può essere definito uno per contratto;
- Se non contrassegnato come payable, genererà un'eccezione se il contratto riceve ether semplice senza dati.

```
contract Test {
   uint public x ;
   function() external { x = 1; }
}
contract Sink {
   function() external payable { }
}
contract Caller {
   function callTest(Test test) public returns (bool) {
      (bool success,) =
address(test).call(abi.encodeWithSignature("nonExistingFunction()"));
      require(success);
      // test.x is now 1

      address payable testPayable = address(uint160(address(test)));

      // Sending ether to Test contract,
      // the transfer will fail, i.e. this returns false here.
      return (testPayable.send(2 ether));
   }
   function callSink(Sink sink) public returns (bool) {
      address payable sinkPayable = address(sink);
      return (sinkPayable.send(2 ether));
   }
}
```

FUNZIONI DI OVERLOADING:

È possibile avere più definizioni per lo stesso nome di funzione nello stesso ambito. La definizione della funzione deve differire l'una dall'altra per i tipi e/o il numero di argomenti nell'elenco degli argomenti. Non è possibile eseguire l'overload di dichiarazioni di funzioni che differiscono solo per il tipo restituito.

```
contract Test {
   function getSum(uint a, uint b) public pure returns(uint){
      return a + b;
   }
   function getSum(uint a, uint b, uint c) public pure returns(uint){
      return a + b + c;
   }
   function callSumWithTwoArguments() public pure returns(uint){
      return getSum(1,2);
   }
   function callSumWithThreeArguments() public pure returns(uint){
      return getSum(1,2,3);
   }
}
```

```
0: uint256: 3
0: uint256: 6
```

FUNZIONI MATEMATICHE:

Solidity fornisce anche ***funzioni matematiche*** integrate. Di seguito sono riportati metodi molto utilizzati:

- ***addmod(uint x, uint y, uint k) returns (uint)***, calcola (x + y) % k dove l'addizione viene eseguita con precisione arbitraria e non si avvolge a 2^{256};
- ***mulmod(uint x, uint y, uint k) returns (uint)***, calcola (x * y) % k dove l'addizione viene eseguita con precisione arbitraria e non si avvolge a 2^{256}.

FUNZIONI CRITTOGRAFICHE:

Solidity fornisce anche ***funzioni crittografiche*** integrate. Di seguito sono riportati metodi importanti:

- ***keccak256(bytes memory) returns (bytes32)***, calcola l'hash Keccak-256 dell'input;
- ***sha256(bytes memory) returns (bytes32)***, calcola l'hash SHA-256 dell'input;
- ***ripemd160(bytes memory) returns (bytes20)***, calcola l'hash RIPEMD-160 dell'input;
- ***sha256(bytes memory) returns (bytes32)***, calcola l'hash SHA-256 dell'input;
- ***ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)***, recupera l'indirizzo associato alla chiave pubblica dalla firma della curva ellittica o restituisce zero in caso di errore. I parametri della funzione corrispondono ai valori ECDSA della firma: r - primi 32 byte di firma; s: secondo 32 byte di firma; v: ultimo 1 byte di firma. Questo metodo restituisce un indirizzo.

WITHDRAWAL PATTERN:

Il ***Withdrawal pattern*** garantisce che non vengano effettuate chiamate di trasferimento diretto che rappresentano una minaccia per la sicurezza. Il seguente contratto mostra l'uso insicuro della chiamata di trasferimento per inviare ether.

```
contract Test {
   address payable public richest;
   uint public mostSent;

   constructor() public payable {
      richest = msg.sender;
      mostSent = msg.value;
   }
   function becomeRichest() public payable returns (bool) {
      if (msg.value > mostSent) {
         // Insecure practice
         richest.transfer(msg.value);
         richest = msg.sender;
         mostSent = msg.value;
         return true;
      } else {
         return false;
      }
   }
}
```

Il contratto di cui sopra può essere reso inutilizzabile facendo sì che il più ricco sia un contratto di fallback. Quando la funzione fallback fallisce, anche la funzione `becomeRichest()` fallisce e il contratto si bloccherà per sempre. Per mitigare questo problema, possiamo utilizzare il withdrawal pattern.

Nel modello withdrawal, reimposteremo l'importo in sospeso prima di ogni trasferimento. Garantirà che solo il contratto del chiamante fallisca.

```
contract Test {
   address public richest;
   uint public mostSent;

   mapping (address => uint) pendingWithdrawals;

   constructor() public payable {
      richest = msg.sender;
      mostSent = msg.value;
   }
   function becomeRichest() public payable returns (bool) {
      if (msg.value > mostSent) {
         pendingWithdrawals[richest] += msg.value;
         richest = msg.sender;
         mostSent = msg.value;
         return true;
      } else {
         return false;
      }
   }
   function withdraw() public {
      uint amount = pendingWithdrawals[msg.sender];
      pendingWithdrawals[msg.sender] = 0;
      msg.sender.transfer(amount);
   }
}
```

RESTRICTED ACCESS:

L'***accesso limitato*** a un contratto è una pratica comune. Per impostazione predefinita, uno stato del contratto è di sola lettura a meno che non sia specificato come pubblico.

Possiamo limitare chi può modificare lo stato del contratto o chiamare le funzioni di un contratto usando i modificatori. Creeremo e utilizzeremo più modificatori come spiegato di seguito:

- ***onlyBy***, una volta utilizzato su una funzione, solo il chiamante menzionato può chiamare questa funzione;
- ***onlyAfter***, una volta usata su una funzione, quella funzione può essere chiamata dopo un certo periodo di tempo;
- ***costs***, una volta utilizzato su una funzione, il chiamante può chiamare questa funzione solo se viene fornito un certo valore.

```
contract Test {
    address public owner = msg.sender;
    uint public creationTime = now;

    modifier onlyBy(address _account) {
        require(
            msg.sender == _account,
            "Sender not authorized."
        );
        _;
    }
    function changeOwner(address _newOwner) public onlyBy(owner) {
        owner = _newOwner;
    }
    modifier onlyAfter(uint _time) {
        require(
            now >= _time,
            "Function called too early."
        );
        _;
    }
    function disown() public onlyBy(owner) onlyAfter(creationTime + 6 weeks) {
        delete owner;
    }
    modifier costs(uint _amount) {
        require(
            msg.value >= _amount,
            "Not enough Ether provided."
        );
        _;
        if (msg.value > _amount)
            msg.sender.transfer(msg.value - _amount);
    }
    function forceOwnerChange(address _newOwner) public payable costs(200 ether) {
        owner = _newOwner;
        if (uint(owner) & 0 == 1) return;
    }
}
```

CONTRATTI:

Il ***contratto*** in Solidity è simile a una classe in C++. Un contratto ha le seguenti proprietà:

- ***Costruttore***: Una funzione speciale dichiarata con la parola chiave constructor che verrà eseguita una volta per contratto e viene invocata quando viene creato un contratto;
- ***Variabili di stato***: Variabili per contratto per memorizzare lo stato del contratto;
- ***Funzioni***: Funzioni per contratto che possono modificare le variabili di stato per alterare lo stato di un contratto.

Di seguito sono riportati vari quantificatori di visibilità per funzioni/variabili di stato di un contratto:

- ***external***: Le funzioni esterne sono destinate ad essere chiamate da altri contratti. Non possono essere utilizzati per chiamate interne. Per chiamare una funzione esterna all'interno del contratto è richiesta la chiamata this.function_name(). Le variabili di stato non possono essere contrassegnate come esterne;
- ***public***: Funzioni pubbliche/Variabili possono essere utilizzate sia esternamente che internamente. Per la variabile di stato pubblico, Solidity crea automaticamente una funzione getter;
- ***internal***: Funzioni interne/Variabili possono essere utilizzate solo internamente o tramite contratti derivati;
- ***private***: Funzioni private/Variabili possono essere utilizzate solo internamente e nemmeno da contratti derivati.

```
contract C {
    //private state variable
    uint private data;
    //public state variable
    uint public info;

    //constructor
    constructor() public {
        info = 10;
    }
    //private function
    function increment(uint a) private pure returns(uint) { return a + 1; }
    //public function
    function updateData(uint a) public { data = a; }
    function getData() public view returns(uint) { return data; }
    function compute(uint a, uint b) internal pure returns (uint) {
        return a + b;
    }
}
//External Contract
contract D {
    function readData() public returns(uint) {
        C c = new C();
        c.updateData(7);
        return c.getData();
    }
}
//Derived Contract
contract E is C {
    uint private result;
    C private c;

    constructor() public {
        c = new C();
    }
    function getComputedResult() public {
        result = compute(3, 5);
    }
    function getResult() public view returns(uint) { return result; }
    function getData() public view returns(uint) { return c.info(); }
}
```

EREDITARIETÀ:

L'***ereditarietà*** è un modo per estendere la funzionalità di un contratto. Solidity supporta sia l'ereditarietà singola che multipla:

- Un ***contratto derivato*** può accedere a tutti i membri non privati, inclusi i metodi interni e le variabili di stato. Ma l'utilizzo di questo non è consentito;
- L'***override della funzione*** è consentito a condizione che la firma della funzione rimanga la stessa. In caso di differenza dei parametri di output, la compilazione fallirà;
- Possiamo chiamare la funzione di un super contratto utilizzando la parola chiave ***super*** o il nome del super contratto;
- In caso di ***eredità multipla***, la chiamata di funzione tramite super dà la preferenza alla maggior parte dei contratti derivati.

Nell'esempio precedente è presente un tipo di contratto derivato.

COSTRUTTORE:

Il ***costruttore*** è una funzione speciale dichiarata utilizzando la parola chiave constructor. È una funzione opzionale e viene utilizzata per inizializzare le variabili di stato di un contratto. Di seguito sono riportate le caratteristiche chiave di un costruttore:

- Un contratto può avere un solo costruttore;
- Il codice del costruttore viene eseguito una volta quando viene creato un contratto e viene utilizzato per inizializzare lo stato del contratto;
- Dopo l'esecuzione del codice del costruttore, il codice finale viene distribuito sulla blockchain. Questo codice include funzioni pubbliche e codice raggiungibile tramite funzioni pubbliche. Il codice del costruttore o qualsiasi metodo interno utilizzato solo dal costruttore non sono inclusi nel codice finale;
- Un costruttore può essere public o internal;
- Un costruttore internal contrassegna il contratto come astratto;
- Nel caso in cui non sia definito alcun costruttore, nel contratto è presente un costruttore predefinito;
- Nel caso in cui il contratto di base abbia un costruttore con argomenti, ogni contratto derivato deve passarli;
- Il costruttore di base può essere inizializzato direttamente usando il seguente modo:

```
contract Base {
   uint data;
   constructor(uint _data) public {
      data = _data;
   }
}
contract Derived is Base (5) {
   constructor() public {}
}
```

- Il costruttore di base può essere inizializzato indirettamente usando il seguente modo:

```
contract Base {
   uint data;
   constructor(uint _data) public {
      data = _data;
   }
}
contract Derived is Base {
   constructor(uint _info) Base(_info * _info) public {}
}
```

- Non sono consentiti modi diretti e indiretti di inizializzare il costruttore del contratto di base;
- Se il contratto derivato non passa argomenti al costruttore del contratto di base, il contratto derivato diventerà astratto.

CONTRATTI ASTRATTI:

Il ***contratto astratto*** è uno che contiene almeno una funzione senza alcuna implementazione. Tale contratto viene utilizzato come contratto di base. Generalmente un contratto astratto contiene sia funzioni implementate che funzioni astratte. Il contratto derivato implementerà la funzione astratta e utilizzerà le funzioni esistenti come e quando richiesto.

Nel caso in cui un contratto derivato non implementi la funzione astratta, questo contratto derivato verrà contrassegnato come astratto.

```
contract Calculator {
   function getResult() public view returns(uint);
}
contract Test is Calculator {
   function getResult() public view returns(uint) {
      uint a = 1;
      uint b = 2;
      uint result = a + b;
      return result;
   }
}
```

INTERFACCE:

Le ***interfacce*** sono simili ai contratti astratti e vengono create utilizzando la parola chiave interface. Di seguito sono riportate le caratteristiche:

- L'interfaccia non può avere alcuna funzione con l'implementazione;
- Le funzioni di un'interfaccia possono essere solo di tipo external;
- L'interfaccia non può avere un costruttore;
- L'interfaccia non può avere variabili di stato;
- L'interfaccia può avere enum, struct a cui è possibile accedere utilizzando la notazione del punto del nome dell'interfaccia.

```
interface Calculator {
   function getResult() external view returns(uint);
}
contract Test is Calculator {
   constructor() public {}
   function getResult() external view returns(uint){
      uint a = 1;
      uint b = 2;
      uint result = a + b;
      return result;
   }
}
```

LIBRERIE:

Le ***librerie*** sono simili ai contratti ma sono principalmente destinate al riutilizzo. Una libreria contiene funzioni che possono essere richiamate da altri contratti. Solidity ha alcune restrizioni sull'uso di una libreria. Di seguito sono riportate le caratteristiche chiave di una Solidity Library:

- Le funzioni di libreria possono essere chiamate direttamente se non modificano lo stato. Ciò significa che le funzioni pure o view possono essere chiamate dall'esterno della libreria;
- La libreria non può essere distrutta in quanto si presume che sia stateless;
- Una libreria non può avere variabili di stato;
- Una libreria non può ereditare alcun elemento;
- Una libreria non può essere ereditata.

```
library Search {
   function indexOf(uint[] storage self, uint value) public view returns (uint) {
      for (uint i = 0; i < self.length; i++) if (self[i] == value) return i;
      return uint(-1);
   }
}
contract Test {
   uint[] data;
   constructor() public {
      data.push(1);
      data.push(2);
   }
   function isValuePresent() external view returns(uint){
      uint value = 2;
      //search if value is present in the array using Library function
      uint index = Search.indexOf(data, value);
      return index;
   }
}
```

La direttiva "*using A for B;*" può essere utilizzato per collegare le funzioni di libreria della libreria A a un dato tipo B. Queste funzioni utilizzeranno il tipo chiamante come primo parametro (identificato utilizzando self).

```
library Search {
   function indexOf(uint[] storage self, uint value) public view returns (uint) {
      for (uint i = 0; i < self.length; i++)if (self[i] == value) return i;
      return uint(-1);
   }
}
contract Test {
   using Search for uint[];
   uint[] data;
   constructor() public {
      data.push(1);
      data.push(2);
   }
   function isValuePresent() external view returns(uint){
      uint value = 2;
      //Now data is representing the Library
      uint index = data.indexOf(value);
      return index;
   }
}
```

EVENTI:

L'***evento*** è un membro ereditabile di un contratto. Viene emesso un evento, memorizza gli argomenti passati nei log delle transazioni. Questi log sono archiviati su blockchain e sono accessibili utilizzando l'indirizzo del contratto fino a quando il contratto non è presente sulla blockchain. Un evento generato non è accessibile dall'interno dei contratti, nemmeno quello che li ha creati ed emessi.

Un evento può essere dichiarato utilizzando la parola chiave event:

```
//Declare an Event
event Deposit(address indexed _from, bytes32 indexed _id, uint _value);
//Emit an event
emit Deposit(msg.sender, _id, msg.value);
```

Per capire come funziona un evento in Solidity:

Prima crea un contratto ed emetti un evento.

```
contract Test {
   event Deposit(address indexed _from, bytes32 indexed _id, uint _value);
   function deposit(bytes32 _id) public payable {
      emit Deposit(msg.sender, _id, msg.value);
   }
}
```

Per accedere all'evento del contratto si può utilizzare un codice in Javascript:

```
var abi = /* abi as generated using compiler */;
var ClientReceipt = web3.eth.contract(abi);
var clientReceiptContract = ClientReceipt.at("0x1234...ab67" /* address */);
var event = clientReceiptContract.Deposit(function(error, result) {
   if (!error)console.log(result);
});
```

Dovrebbe stampare dettagli simili a quanto segue:

```
{
   "returnValues": {
      "_from": "0x1111...FFFFCCCC",
      "_id": "0x50...sd5adb20",
      "_value": "0x420042"
   },
   "raw": {
      "data": "0x7f...91385",
      "topics": ["0xfd4...b4ead7", "0x7f...1a91385"]
   }
}
```

GESTORE DEGLI ERRORI:

Solidity fornisce varie funzioni per la gestione degli errori. In genere, quando si verifica un errore, lo stato viene riportato allo stato originale. Altri controlli servono a prevenire l'accesso non autorizzato al codice. Di seguito sono riportati alcuni dei metodi importanti utilizzati:

- ***assert(bool condition)***: Nel caso in cui la condizione non sia soddisfatta, questa chiamata al metodo causa un opcode non valido e qualsiasi modifica apportata allo stato viene annullata. Questo metodo deve essere utilizzato per gli errori interni;
- ***require(bool condition)***: Nel caso in cui la condizione non sia soddisfatta, questa chiamata al metodo ritorna allo stato originale. Questo metodo deve essere utilizzato per errori negli input o nei componenti esterni;
- ***require(bool condition, string memory message)***: Nel caso in cui la condizione non sia soddisfatta, questa chiamata al metodo ritorna allo stato originale. Questo metodo deve essere utilizzato per errori negli input o nei componenti esterni. Fornisce un messaggio personalizzato;
- ***revert()***: Questo metodo interrompe l'esecuzione e ripristina tutte le modifiche apportate allo stato;
- ***revert (string memory reason)***: Questo metodo interrompe l'esecuzione e ripristina tutte le modifiche apportate allo stato. Fornisce un'opzione per fornire un messaggio personalizzato.

```
contract Vendor {
   address public seller;
   modifier onlySeller() {
      require(
         msg.sender == seller,
         "Only seller can call this."
      );
      _;
   }
   function sell(uint amount) public payable onlySeller {
      if (amount > msg.value / 2 ether)
         revert("Not enough Ether provided.");
      // Perform the sell operation.
   }
}
```

Quando viene chiamato revert, restituirà i dati esadecimali come segue:

```
0x08c379a0                         // Function selector for Error(string)
0x0000000000000000000000000000000000000000000000000000000000000020 // Data offset
0x000000000000000000000000000000000000000000000000000000000000001a // String length
0x4e6f7420656e6f756768204574686572207072f76696465642e000000000000 // String data
```