Argomenti trattati nelle slide Solidity

MetaMask

Valute digitali

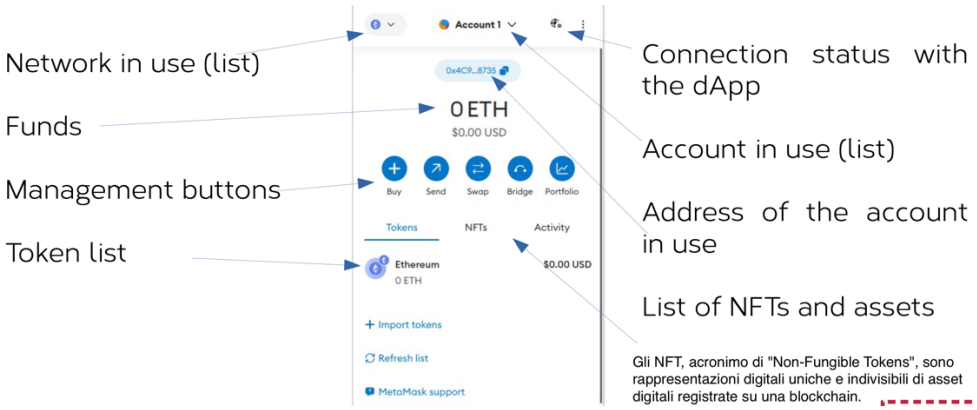
MetaMask è uno strumento estremamente utile per la gestione degli asset, simile a un portafoglio digitale, e per la creazione di DApp (Decentralized Applications). Si tratta di un'estensione per browser che offre funzionalità avanzate rispetto a un semplice portafoglio digitale. MetaMask è comunemente definito come un "ponte" o "bridge" poiché agisce come un collegamento diretto tra il nostro browser web e la rete blockchain.

MetaMask consente agli utenti di interagire con le blockchain in modo semplice e intuitivo, fornendo un'interfaccia utente amichevole per l'accesso alle DApp, la gestione delle chiavi private e la firma di transazioni. Tra le sue principali funzionalità vi sono la creazione e l'importazione di portafogli, la visualizzazione del saldo e della cronologia delle transazioni, e la possibilità di connettersi a varie reti blockchain, inclusi testnet e mainnet.

Inoltre, MetaMask offre una piattaforma di sviluppo per gli sviluppatori che desiderano creare e distribuire le proprie DApp. Fornisce strumenti per la creazione e il test delle DApp, nonché per l'interazione con i contratti intelligenti presenti sulla blockchain.

In sintesi, MetaMask è una potente estensione del browser che semplifica l'accesso e l'utilizzo delle DApp e delle blockchain, offrendo agli utenti una vasta gamma di funzionalità per la gestione degli asset digitali e la partecipazione alle attività sulla blockchain.

- **Argomento importante: NFT**
  Gli NFT, acronimo di "Non-Fungible Tokens", sono rappresentazioni digitali uniche e indivisibili di asset digitali registrate su una blockchain.



Remix

Remix è un ambiente di sviluppo integrato (IDE) per la scrittura, il test e il debug di contratti intelligenti su diverse blockchain, con un'attenzione particolare a Ethereum. È una delle principali piattaforme utilizzate dagli sviluppatori per sviluppare contratti intelligenti in Solidity, il linguaggio di programmazione più diffuso per la scrittura di contratti intelligenti su Ethereum e altre blockchain compatibili con EVM (Ethereum Virtual Machine).
Le principali caratteristiche di Remix includono:

1. **Editor di codice**: Remix offre un editor di codice integrato che supporta la scrittura e la modifica dei contratti intelligenti in Solidity(Language Programming), insieme a funzionalità di evidenziazione della sintassi e completamento automatico del codice.

2. **Simulatore di contratti**: Remix include un simulatore di contratti che consente agli sviluppatori di eseguire e testare i loro contratti intelligenti direttamente nell'ambiente di sviluppo. Questo permette agli sviluppatori di verificare il comportamento dei loro contratti e individuare eventuali errori o bug prima di deployarli sulla blockchain.

3. **Debugging**: Remix offre strumenti per il debugging dei contratti intelligenti, inclusa la possibilità di impostare punti di interruzione e monitorare lo stato del contratto durante l'esecuzione.

4. **Gestione dei contratti**: Remix consente agli sviluppatori di creare, deployare e gestire i loro contratti intelligenti direttamente dall'IDE, semplificando il processo di sviluppo e deployment.

5. **Integrazione con le principali blockchain**: Remix offre integrazioni con diverse blockchain, inclusi testnet e mainnet di Ethereum, consentendo agli sviluppatori di testare e deployare i loro contratti su diverse reti blockchain.

In sintesi, Remix è uno strumento fondamentale per lo sviluppo di contratti intelligenti su Ethereum e altre blockchain compatibili con EVM, offrendo un ambiente integrato e intuitivo per la scrittura, il test e il debug dei contratti intelligenti.

SOLC

Solc è l'acronimo di "SOLidity Compiler", ed è il compilatore del linguaggio Solidity, utilizzato per trasformare il codice sorgente scritto in Solidity in bytecode, un linguaggio di basso livello comprensibile dall'Ethereum Virtual Machine (EVM).

Il bytecode prodotto da Solc è ciò che effettivamente viene eseguito sulla blockchain Ethereum quando si deploya un contratto intelligente. Questo bytecode definisce le operazioni e le istruzioni che la EVM deve eseguire quando il contratto viene chiamato o interagisce con altri contratti sulla rete.

Remix, un popolare ambiente di sviluppo per Ethereum, include numerose versioni del compilatore Solidity (solc) integrate direttamente nell'ambiente, permettendo agli sviluppatori di compilare e testare i loro contratti intelligenti senza la necessità di installare il compilatore localmente.

Tuttavia, è possibile installare il compilatore Solidity (solc) anche localmente seguendo le istruzioni fornite nella documentazione ufficiale di Solidity. Una volta installato, è possibile utilizzare il comando `solc --version` per verificare la versione del compilatore installata nel proprio sistema.

In sintesi, Solc è il compilatore del linguaggio Solidity utilizzato per convertire il codice sorgente in bytecode eseguibile dalla EVM. Viene utilizzato ampiamente dagli sviluppatori per sviluppare e deployare contratti intelligenti su Ethereum e altre blockchain compatibili con Solidity.

SOLIDITY

Solidity è il linguaggio di programmazione più utilizzato per la scrittura di contratti intelligenti. Originariamente progettato per l'ambiente Ethereum e l'Ethereum Virtual Machine (EVM), Solidity è oggi utilizzato anche su numerose blockchain simili all'EVM.

Si tratta di un linguaggio di alto livello che offre un'astrazione potente e intuitiva per la scrittura di contratti intelligenti. Solidity è ispirato a linguaggi di programmazione popolari come Java, C++ e Python, ma è progettato specificamente per soddisfare le esigenze della blockchain e delle applicazioni decentralizzate (DApp).

Il paradigma di programmazione di Solidity è orientato agli oggetti, il che significa che consente agli sviluppatori di definire e manipolare oggetti e classi all'interno dei loro contratti intelligenti. Questo approccio object-oriented rende Solidity più familiare e accessibile per i programmatori provenienti da altri background di programmazione.

Solidity presenta caratteristiche uniche che lo rendono particolarmente adatto allo sviluppo di applicazioni decentralizzate. Ad esempio, offre supporto integrato per i contratti intelligenti, consentendo agli sviluppatori di definire contratti, interfacce, librerie e altri costrutti di programmazione specifici della blockchain.

Inoltre, Solidity offre meccanismi di sicurezza avanzati per prevenire errori e vulnerabilità nei contratti intelligenti, come la gestione dei modelli di accesso, la gestione delle eccezioni e la prevenzione degli attacchi di tipo reentrancy e overflow.

In sintesi, Solidity è un linguaggio di programmazione orientato agli oggetti progettato per lo sviluppo di contratti intelligenti su blockchain. Con la sua sintassi intuitiva, il suo supporto per i costrutti di programmazione blockchain e le sue caratteristiche di sicurezza avanzate, Solidity è diventato uno strumento fondamentale per la creazione di applicazioni decentralizzate e contratti intelligenti affidabili e sicuri.
In Solidity si scrivono i Contratti Intelligenti (SC), o contratti. Un Contratto Intelligente è simile a una classe di programmazione orientata agli oggetti, in quanto contiene dati e operazioni, e può ereditare da altri Contratti Intelligenti. È possibile scrivere anche librerie e contratti astratti.
Un file di codice può includere altri file e contenere il codice di più Contratti Intelligenti.
Tuttavia, l'unità di codice da "installare" sulla blockchain è sempre uno e solo uno Smart Contract.
Il codice, dopo la compilazione, diventa bytecode.

In Solidity, **pragma** è una parola chiave utilizzata per specificare la versione del compilatore Solidity da utilizzare per compilare il contratto intelligente. Viene utilizzato all'inizio del file di codice sorgente Solidity per informare il compilatore sulla versione del linguaggio da utilizzare e per impostare alcune opzioni di compilazione.
pragma solidity ^0.8.0;

- **Solidity Contract**

La parola chiave `contract` consente di definire un blocco di codice simile a una classe, all'interno del quale saranno presenti tutte le istruzioni e i dati necessari per implementare la funzionalità desiderata. Il nome del contratto deve avere la prima lettera in maiuscolo.

Una volta che il codice sorgente è stato compilato, ogni contratto può essere caricato (deployed) sulla blockchain indipendentemente dagli altri contratti nel codice. A ciascun contratto sarà assegnato un indirizzo sulla blockchain.

Ad esempio, un semplice contratto in Solidity potrebbe apparire così:

```
contract MyContract {
    // Dichiarazione di variabili di stato
    uint256 public myVariable;

    // Definizione del costruttore
    constructor() {
        myVariable = 100;
    }

    // Definizione di una funzione
    function setVariable(uint256 newValue) public {
        myVariable = newValue;
    }

    // Altre funzioni...
}
```

In questo esempio, `MyContract` è il nome del contratto, che inizia con una lettera maiuscola, come richiesto. Il contratto contiene una variabile di stato `myVariable`, un costruttore e una funzione `setVariable` per impostare il valore della variabile di stato. Una volta compilato, questo contratto può essere caricato sulla blockchain e utilizzato interagendo con il suo indirizzo.

Solidity supporta i tag di documentazione NatSpec. Questi tag possono essere utilizzati per documentare contratti, funzioni e variabili nel codice sorgente Solidity.

Ecco un esempio di utilizzo dei tag di documentazione NatSpec in un contratto Solidity:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;

/// @title Contratto vuoto
/// @author Nessuno
/// @notice Questo codice sorgente è vuoto!
/// @dev Non c'è niente da descrivere per il dev.
contract MyEmptyContract {
   // all'interno del contratto
}
```
In questo esempio, abbiamo utilizzato i seguenti tag di documentazione NatSpec:

- `@title`: Descrive il titolo del contratto.
- `@author`: Specifica l'autore del contratto.
- `@notice`: Fornisce una breve descrizione o avviso sul contratto.
- `@dev`: Fornisce informazioni specifiche per i dev o i programmatori.

È possibile esportare la documentazione utilizzando il compilatore Solidity con il seguente comando:
solc --userdoc --devdoc example.sol

Solidity include anche istruzioni condizionali e cicli. Le parole chiave del linguaggio per le condizioni, i cicli e i salti sono:

- `if`: per le istruzioni condizionali.
- `else`: per gestire le condizioni alternative.
- `while`: per i cicli basati su condizioni.
- `do`: per i cicli con condizione di uscita posticipata.
- `for`: per i cicli con un numero fisso di iterazioni.
- `break`: per uscire da un ciclo.
- `continue`: per passare all'iterazione successiva di un ciclo.
- `return`: per restituire un valore da una funzione.

- **Visibilità**
  La visibilità delle funzioni in Solidity può essere stabilita in diversi modi, determinando come possono essere chiamate e da dove possono essere accessibili. Ecco una spiegazione delle diverse modalità di visibilità:

  1. **Private**: Le funzioni contrassegnate come private possono essere chiamate solo da altre funzioni all'interno del contratto stesso. Non sono accessibili da contratti esterni o da altri account.

```
contract MyContract {
   function privateFunction() private {
      // Logica della funzione
   }
}
```

2. **Public**: Le funzioni contrassegnate come pubbliche possono essere chiamate da qualsiasi account, inclusi altri contratti e account esterni alla blockchain. Sono visibili e accessibili da qualsiasi parte del codice.

```
contract MyContract {
  function publicFunction() public {
    // Logica della funzione
  }
}
```

3. **Internal**: Le funzioni contrassegnate come interne possono essere chiamate solo all'interno del contratto stesso o dai contratti che ereditano dal contratto in cui sono definite. Non sono accessibili da contratti esterni o da altri account.

```
contract MyContract {
  function internalFunction() internal {
    // Logica della funzione
  }
}
```

4. **External**: Le funzioni contrassegnate come esterne possono essere chiamate solo da account esterni al contratto, come ad esempio da un'altra istanza di contratto o da un account Ethereum. Non possono essere chiamate dalle altre funzioni interne del contratto stesso.

```
contract MyContract {
  function externalFunction() external {
    // Logica della funzione
  }
}
```

Queste modalità di visibilità consentono ai programmatori di controllare l'accesso e l'utilizzo delle funzioni all'interno dei loro contratti, garantendo la sicurezza e il corretto funzionamento delle operazioni.

- **Costruttore**

Il costruttore è una funzione speciale che viene eseguita quando il file il contratto viene distribuito nella blockchain.
Di seguito riporterò un'immagine che lo rappresenta:

Syntax:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;
contract ConnractName {
   // ...
   constructor(arguments) {
      // something to be executed
   }
}
```

Esercizio

The file must contain code written in a version of solidity compatible with 0.8.1 up to, but excluding, 0.9.0. The file must be released under the GPL-3.0 license. The file contains the code to create a “SaveValue” smart contract that has a public “value” variable of type uint16.
This variable is assigned the value 10 during deployment, via the constructor.

```
1   // SPDX-License-Identifier: GPL-3.0
2   pragma solidity ^0.8.1;
3
4   contract SaveValue
5   {
6       uint16 public myValue;
7       constructor()     63360 gas 39000 gas
8       {
9           myValue = 10;
10      }
11  }
```

- **Compilazione**

Dopo la compilazione di un contratto Solidity, il compilatore Solidity mostrerà eventuali errori e avvertimenti riscontrati durante il processo di compilazione. Se non ci sono errori, verrà visualizzato il risultato della compilazione, che includerà la lista dei contratti compilati.
Per ciascun contratto compilato, sarà possibile leggere i dettagli della compilazione, compreso l'ABI (Application Binary Interface) e il bytecode generato. L'ABI e il bytecode sono informazioni essenziali per interagire con il contratto sulla blockchain.
Per ottenere l'ABI e il bytecode di un contratto compilato, è possibile visualizzare l'output della compilazione fornito dal compilatore Solidity, o utilizzare strumenti di sviluppo come Remix IDE o Truffle Suite. Una volta ottenuti l'ABI e il bytecode, è possibile copiarli e incollarli in un file di testo o in un altro file per l'uso futuro.
L'ABI viene utilizzata per interagire con il contratto dalla parte del client, consentendo di chiamare le funzioni del contratto e di ottenere informazioni sulle sue variabili di stato. Il bytecode, d'altra parte, è il codice eseguibile che viene effettivamente caricato sulla blockchain quando il contratto viene deployato, definendo il comportamento e la logica del contratto.

- **Deploy(Distribuzione)**

Remix consente di deployare contratti su diversi ambienti, tra cui:

- Ambienti virtuali basati su JavaScript
- Rete blockchain "iniettata" utilizzando MetaMask
- Rete blockchain via RPC
- Altri tipi di connessioni

Per deployare il contratto sull'ambiente virtuale che simula la versione post-merge di Ethereum utilizzando Remix, segui questi passaggi:

1. Accedi alla scheda "Deploy & Run Transactions" in Remix.
2. Verifica le impostazioni e assicurati di selezionare l'ambiente virtuale JavaScript VM.
3. Clicca su "Deploy" per avviare il processo di deploy del contratto.

Dopo aver cliccato su "Deploy", otterrai un'istanza del contratto e potrai vedere che l'account ha consumato Ether. Assicurati di essere connesso a MetaMask e che sia selezionata la rete Sepolia. MetaMask ti chiederà di consentire la connessione tra MetaMask e Remix e dovrai scegliere uno o più indirizzi da collegare.

Una volta completati questi passaggi, il contratto sarà deployato con successo sull'ambiente virtuale JavaScript VM di Remix e sarai pronto per interagire con esso tramite l'interfaccia di Remix o tramite MetaMask, a seconda della tua configurazione e delle tue preferenze.

- **Funzioni**

Le funzioni sono uno degli elementi fondamentali nella programmazione di contratti intelligenti in Solidity. Esse consentono di definire il comportamento e la logica del contratto, interagendo con i dati globali o i valori dei parametri.

Ecco la sintassi base per definire una funzione in Solidity:

```
function nomeFunzione(parametri) modificatori {
   // corpo della funzione
}
```

Dove:
- `function` è la parola chiave utilizzata per definire una funzione.
- `nomeFunzione` è il nome della funzione.
- `parametri` è l'elenco dei parametri della funzione, separati da virgole se ce ne sono più di uno.
- `modificatori` sono opzionali e possono essere utilizzati per specificare alcune condizioni o restrizioni sulla funzione.

Ad esempio, una funzione semplice che somma due numeri potrebbe essere definita come segue:

```
function somma(uint256 a, uint256 b) public returns (uint256) {
   return a + b;
}
```

In questo esempio, la funzione `somma` accetta due parametri di tipo `uint256`, esegue la somma dei due parametri e restituisce il risultato.

Si noti che il costruttore è una funzione speciale che viene eseguita al momento del deploy del contratto sulla blockchain. Il costruttore non ha un nome e ha lo stesso nome del contratto. Il costruttore è spesso utilizzato per inizializzare lo stato del contratto o eseguire altre operazioni necessarie al momento della creazione del contratto.

- **Pure & View**

In Solidity, l'esecuzione delle funzioni comporta tipicamente il consumo di gas, che è una misura dello sforzo computazionale richiesto per eseguire la funzione sulla rete Ethereum. Tuttavia, ci sono casi speciali in cui l'esecuzione delle funzioni potrebbe non consumare gas:

1. **Funzioni Pure**: Una funzione pure è una funzione che non modifica alcun dato e non legge alcun dato dalla blockchain. Queste funzioni sono puramente computazionali e

operano solo sui loro parametri di input. Sono utili per eseguire calcoli basati esclusivamente sui valori di input senza interagire con la blockchain. Le funzioni pure sono sempre gratuite da chiamare, poiché non richiedono alcuna risorsa della blockchain.

Esempio:

```
function add(uint256 a, uint256 b) public pure returns (uint256) {
   return a + b;
}
```

2. **Funzioni di Visualizzazione (View Functions)**: Una funzione di visualizzazione è una funzione che legge solo dati dalla blockchain ma non modifica alcun dato. Queste funzioni sono di sola lettura e non modificano lo stato della blockchain. Poiché non modificano lo stato, sono anche gratuite da chiamare a condizione che non siano chiamate da un'altra funzione che modifica lo stato, o se un contratto chiama la funzione tramite una chiamata di basso livello non statica o delegatecall.

Esempio:

```
function getBalance(address account) public view returns (uint256) {
   return balances[account];
}
```

non consumano gas perché non modificano lo stato

In sintesi, le funzioni pure e le funzioni di visualizzazione in Solidity non consumano gas quando vengono chiamate sotto determinate condizioni. Sono utili per eseguire calcoli e leggere dati dalla blockchain senza incorrere in costi di gas.

Esercizio

Let's create a new Test_uint8 contract that contains:
- a public variable of type uint8 (8 bit) named 'little'.
- a constructor that assigns 250 to the variable 'little'.
- an 'increment' function without arguments and publish that adds 1 to the variable 'small'
- a function 'assign5' without arguments and public that assigns 5 to the variable 'small'
- a 'decrement' function without arguments and publish that subtracts 1 from the variable 'small'

```
1  // SPDX-License-Identifier: GPL-3.0
2  pragma solidity ^0.8.1;
3  contract Test_uint8
4  {
5      uint8 public little;
6      constructor() {    136626 gas 112200 gas
7      little = 250;
8      }
9      function increment() public{    infinite gas
10     little += 1;
11     }
12     function assign5() public{    24389 gas
13     little = 5;
14     }
15     function decrement() public{    infinite gas
16     little -= 1;
17     }
18 }
```

- **Data Types in Solidity**

  In Solidity, oltre ai tipi di dati semplici menzionati, esistono anche i tipi di dati strutturati che consentono di organizzare e gestire dati più complessi. Ecco una panoramica dei tipi di dati strutturati in Solidity:

  - **address** which contains a blockchian address
    The address type contains an address compatible with the Ethereum blockchain. At any time we can recall two system variables of type address.
    - **msg.sender**: is the address of the person who interacts directly with the contract.
    - **tx.origin** : is the address of performing the original interaction towards the blockchain. It is used in case of chain calls.
    - global names: in addition to msg and tx there are other global variables.
  - **bytes32** (or other lengths) containing raw data
  - **string** : which contains text strings
  - **uint256** (or other dimensions) containing unsigned integers
  - **bool** which contains true or false
  - **Array** sono simili agli array di java, utilizzano l'annotazione con le parentesi quadre

```
bool[3] boolVet;
boolvet[1] = true;
int[5][3] v2; // array of 3 arrays of 5 int each.
v2[0]=[1,-72,3,13,4];
delete v2; // sets all elements to zero
v2.length;  // number of elements
```

Inoltre ritroviamo il **Mapping**: Una mappatura (mapping) è una struttura dati chiave-valore in cui ogni chiave è unica e mappa a un valore specifico. È simile a un dizionario o a un hash table in altri linguaggi di programmazione. Le mappature sono molto utilizzate per associare un valore a un indirizzo o a un identificatore univoco.

mapping(referenceType => referredType) public nameMapping; // Mappatura degli indirizzi agli importi

Questi tipi di dati strutturati consentono di gestire in modo più efficiente e organizzato dati più complessi all'interno dei contratti Solidity. Possono essere utilizzati per modellare dati in modo più aderente alla logica dell'applicazione e per semplificare le operazioni di accesso e manipolazione dei dati.

```
struct Voter {
    uint weight; // weight is accumulated by delegation
    bool voted;  // if true, that person already voted
    address delegate; // person delegated to
    uint vote;   // index of the voted proposal
}
mapping(address => Voter) public voters;
```

Esercizio:
We create a “Certificates” contract that records some information for each certificate generated for the attendees of a course. The contract has a "price" variable with the value of 1/1000 of ether. The contract implements a struct “certificate” and a mapping.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.1;

contract Certificates
{
    uint price = 1 ether / 10000;
    struct certificate
    {
        uint8 grade;
        address professor;
        uint timestamp;
        bytes signature;
    }
    mapping (address => certificate) certificates;
}
```

- **Record in Solidity**
  - **Structure**:
    Solidity ci permette di dichiarare delle strutture.

```
struct Person{
    string name;
    uint birthdate;
    enum gender;
}
Person aPerson;
aPerson.name = "Andrea";
```

- **Balance in Solidity**
  Il saldo è la quantità della criptovaluta principale espressa attraverso l'unità più piccola (ad esempio wei su Ethereum) associata a un indirizzo. Il metodo di bilancio dell'indirizzo consente di leggere il saldo di qualsiasi indirizzo.

- **Valore di un messaggio.**
  Il messaggio ha sempre un valore associato in ether, che per impostazione predefinita è zero. Il valore del messaggio viene trasferito al contratto intelligente. All'interno del codice, è possibile recuperare il valore del contratto tramite la variabile msg.value.

- **Payble**
  payable è un indirizzo pagabile
  Indirizzo e funzioni pagabili.
  All'interno del contratto possiamo distinguere tra indirizzi "normali" e indirizzi pagabili. Solidity consente agli indirizzi pagabili di inviare criptovaluta al contratto e ricevere criptovaluta dal contratto.

  address payable user;
  Per creare una funzione che, quando chiamata, consente di inviare criptovaluta al contratto, devo specificare il modificatore payable.

```
function deposito() public payable
{
   totalAmount += msg.value;
}
```

- **Transfer**
  Inviare criptovaluta dal contratto a un indirizzo pagabile.
  Il modo più semplice ma non raccomandato è utilizzare il metodo transfer dell'indirizzo.
  Supponiamo di avere un indirizzo dest pagabile:

```
dest.transfer(amount); // il contratto invia l' "amount" a dest.
```

  Il modello consigliato per l'invio di criptovaluta dal contratto si basa sul metodo di chiamata dell'indirizzo, su cui approfondiremo in seguito.

```
(bool success, ) = dest.call{value: amount}("");
require(success, "Trasferimento fallito.");
```

Esercizio

Create a Moneybox contract that contains a variable uint256 deposits the number of deposits made and an address payable owner variable. The constructor stores the address of the creator of the contract in owner.
The contract has three public functions:

- A payable "deposit" function that adds one to the number of deposits if the message value is greater than zero.
- A "withdrawal" function that sends the collected funds to the creator of the contract via transfer.
- A "getDeposits" view function that returns the value of deposits.

```
// // SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.1;

contract Moneybox
{
    uint256 deposits;
    address payable owner;
    constructor(){    122613 gas 98200 gas
    owner = payable(msg.sender);
    }

    function deposit() public payable     infinite gas
    {
        if (msg.value > 0) {
        deposits += 1;
        }
    }

    function withdrawal() public     infinite gas
    {
        owner.transfer(address(this).balance);
    }

    function getDeposits() public view returns(uint256 num_deposits)    2415 gas
    {
        return deposits;
    }
}
```

- **Send**

Il metodo send dell'indirizzo è simile al metodo transfer, ma si comporta in modo leggermente diverso.

```
anAddress.send(uint256 amount) returns (bool)
```

Invia l'importo passato come argomento in Wei all'indirizzo specificato. In caso di fallimento, non viene eseguito un revert, ma il metodo restituisce false. Inoltre, viene addebitata una commissione fissa di 2300 gas.

Calls to contracts

Esegue una chiamata a basso livello con un payload di bytes.

```
<indirizzo>.call(bytes memory)
```

restituisce (bool, bytes memory):
Restituisce true in caso di successo, false in caso di fallimento e i dati di ritorno. Inoltra tutto il gas disponibile alla chiamata. La chiamata esegue una transazione sulla blockchain di cui possiamo modificare ogni parametro. La transazione comporta il consumo di gas.

Esempio: call viene utilizzato per trasferire ether come pattern sicuro, invece di transfer. Per fare ciò, possiamo specificare il valore da inviare insieme alla transazione aggiungendo alcune etichette. Ad esempio, per inviare 100 wei e limitare il gas a 5000, usiamo questa sintassi:

```
(bool success,) = unIndirizzo.call{value: 100, gas: 5000}
```

- **Cosa sono i Wei?**
  "Wei" è l'unità più piccola di valore nella blockchain Ethereum ed è comunemente utilizzata per misurare e rappresentare piccole quantità di ether e altre criptovalute basate su Ethereum. Un ether è suddiviso in 10^18 (1 seguito da 18 zeri) wei. Quindi, 1 ether è uguale a 1.000.000.000.000.000.000 wei.

Call Payload

Il payload in bytes passato come argomento deve corrispondere a una funzione del contratto chiamato. I primi 4 byte del payload sono i primi 4 byte dell'hash SHA3 della "firma" della funzione (solo il nome completo e i tipi dei parametri, senza spazi), come riportato nell'ABI.
Esempio di calcolo della prima parte del payload:

```
bytes4(keccak256("setX(uint256)"))
```

I parametri seguono, in blocchi di 32 byte (256 bit).
Se conosciamo l'ABI del contratto chiamato, possiamo utilizzare la funzione di sistema "abi.encodeWithSignature" che converte una stringa con il prototipo della funzione (firma) seguito dal valore dei parametri nel payload in bytes.

```
bytes memory payload = abi.encodeWithSignature(
```

"nomeFunzione(tipo1, tipo2, tipo3)", param1, param2, param3);
(bool success, bytes memory result) = indirizzoDestinatario.call(payload);

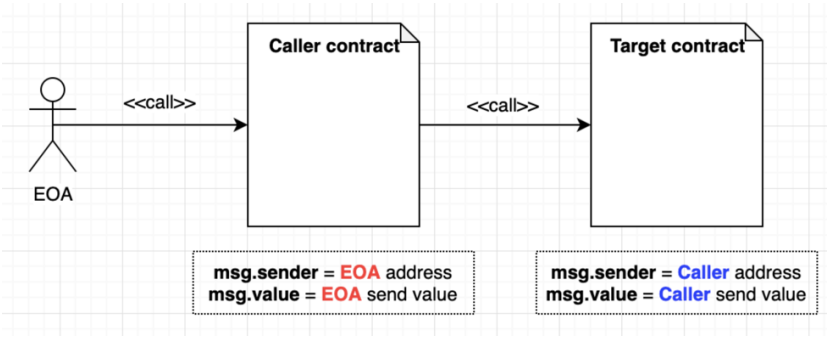
Questo esempio mostra come costruire correttamente il payload per chiamare una funzione di un contratto esterno utilizzando l'ABI. Successivamente, la chiamata viene eseguita con il metodo "call" dell'indirizzo destinatario, passando il payload come argomento.

Static Call

La funzione staticcall è progettata per effettuare chiamate con la certezza che non cambieranno lo stato della blockchain.

```
<indirizzo>.staticcall(bytes memory)
        return (bool, bytes memory):
```

Consente di chiamare le funzioni di visualizzazione (view) di altri contratti. Funziona come la call, ma non consente di specificare il valore da inviare con la chiamata.
Solleva un'eccezione e termina con un "revert" se la chiamata altera lo stato del contratto.

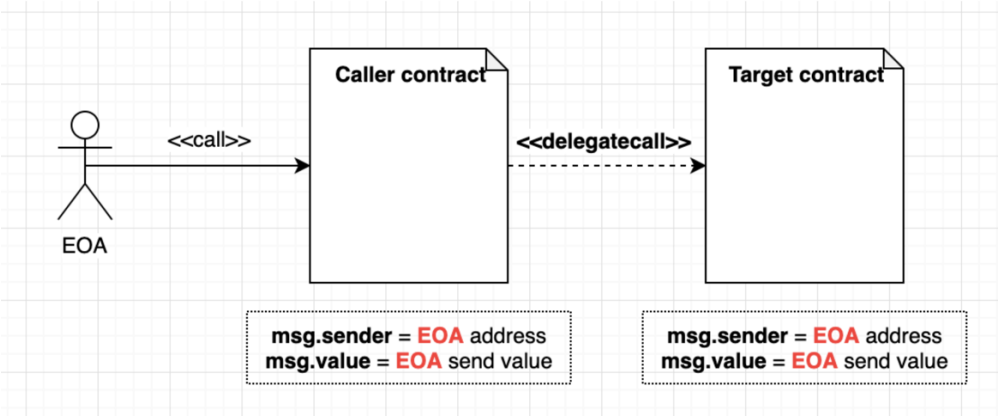


DelegateCall

Il delegatecall utilizza solo il codice del contratto chiamato, ma lo storage e l'Ether del chiamante.
Può essere utilizzato per chiamare contratti già distribuiti contenenti librerie di funzioni utili.

```
<indirizzo>.delegatecall(bytes memory)
        restituisce (bool, bytes memory):
```

Esegue una chiamata a basso livello con il tipo di payload bytes memory. Restituisce true in caso di successo, false in caso di fallimento e dati di ritorno; inoltra tutto il gas disponibile alla chiamata.



Recap:
Le chiamate a basso livello (calls) in Solidity sono utilizzate per eseguire operazioni avanzate che coinvolgono l'interazione con altri contratti sulla blockchain Ethereum. Queste chiamate consentono ai contratti di comunicare tra loro in modo flessibile e di eseguire operazioni più complesse rispetto alle tradizionali transazioni di ether.

In sintesi, le chiamate a basso livello consentono ai contratti di interagire in modo avanzato e flessibile sulla blockchain Ethereum, aprendo la porta a una vasta gamma di casi d'uso e possibilità di sviluppo di applicazioni decentralizzate.

Esercizio:
Write two contracts in the same targetAndCaller.sol file:
-A Target contract that has a public checkCaller function with no arguments that returns a boolean based on the comparison between the msg.sender and the tx.origin.
-A Caller contract with one function: callTarget(address _target) which calls the _target's staticcall method to execute the checkCaller function.
The callTarget function of the second contract returns the bytes memory value returned by the staticcall.

bytes memory payload = abi.encodeWithSignature(
"functionName(type1,type2,type3)", param1, param2, param3);

Deploy the first contract and run checkCaller.
Deploy the second contract and execute callTarget passing the address of the first contract.

```solidity
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;

contract Target
{
    function checkCaller() public view returns (bool)     337 gas
    {
        return msg.sender == tx.origin;
    }
}

contract Caller
{
    function callTarget(address _targetAddress) public view returns (bytes memory)     infin
    {
        // create the function payload
        bytes memory payload = abi.encodeWithSignature("checkCaller()");
        // execute the call
        (bool success, bytes memory result) = _targetAddress.staticcall(payload);
        // check if the call was successful
        require(success, "Call failed");
        return result;
    }
}
```

Events & Logs

Gli eventi sono messaggi specifici che la blockchain emette su richiesta dei contratti intelligenti. Sono utili per creare una documentazione di "log" sulla blockchain e per catturare eventi da segnalare al di fuori della blockchain e all'interfaccia utente. Utilizziamo la parola chiave `event` per dichiarare un evento e `emit` per invocarlo.
La sintassi per dichiarare un evento è la seguente (da scrivere nel codice principale del contratto):

```
event eventName(declaration of arguments to emit);
```

La sintassi per chiamare un evento è la seguente (da scrivere all'interno di una funzione):

```
emit eventName(values to emit);
```

Durante l'esecuzione del codice, possono essere generati eventi, con dati associati. Il tipo e i dati relativi agli eventi sono memorizzati all'interno del registro delle transazioni che li generano. Esiste un indice (filtro di Bloom) memorizzato sulla blockchain per trovare gli eventi.

Questa struttura dati (log) può essere letta dall'esterno della blockchain, ma non dalle funzioni dei contratti intelligenti. I log possono essere utilizzati per registrare informazioni non più utili per l'esecuzione dei contratti intelligenti, ma di interesse per gli osservatori esterni.

Esempio: dichiariamo l'evento "valueUpdated" che emette un uint8 chiamato "newValue".

```
event valueUpdated(uint8 newValue);
```

Esempio: emitiamo l'evento "valueUpdated" che prende la variabile `newVal` come input.

```
emit valueUpdated(newVal);
```

Nota: È possibile dichiarare fino a 3 argomenti di evento come indicizzati. Questi verranno utilizzati nel calcolo della firma dell'evento (è possibile filtrare la ricerca degli eventi utilizzando quei parametri). Quelli non indicizzati saranno inclusi nei byte dell'evento.

RECAP:
Gli eventi sono utilizzati per registrare e comunicare informazioni specifiche sulla blockchain in modo che possano essere catturate e interpretate dagli osservatori esterni, come altre applicazioni o interfacce utente
Essi quindi forniscono un meccanismo per la registrazione e la comunicazione di informazioni importanti sulla blockchain, consentendo agli sviluppatori di costruire applicazioni più sofisticate e interattive.

Esercizio

In the Certificates contract we declare the "newAttendance" event which emits an address named "attendee" and a uint named "payment". In the addMe function we emit the newAttendance event which takes as input the caller's address and the msg.value.

```
1  // SPDX-License-Identifier: GPL-3.0
2  pragma solidity ^0.8.1;
3
4  contract Certificates
5  {
6      event newAttendance(address attendee, uint payment);
7      uint price = 1 ether / 1000;
8      struct certificate
9      {
10         uint8 grade;
11         address professor;
12         uint timestamp;
13         bytes signature;
14     }
15     mapping (address => certificate) certificates;
16     mapping (address => bool) attendance;
17     function addMe() public payable    26051 gas
18     {
19         attendance[msg.sender]=true;
20         emit newAttendance(msg.sender, msg.value);
21     }
22 }
```

Error Management

Queste sono funzioni utili in Solidity per gestire errori e condizioni indesiderate durante l'esecuzione del contratto. Ecco una spiegazione di ciascuna:

1. assert(bool condition):

- Questa funzione viene utilizzata per verificare condizioni interne al contratto.
- Se la condizione booleana passata come argomento è falsa, causa un errore di "Panico" e provoca il rollback di tutte le modifiche di stato effettuate fino a quel momento nell'esecuzione del contratto.
- È utile per rilevare errori interni al contratto che non dovrebbero mai verificarsi.

2. require(bool condition, string memory message):

- Questa funzione viene utilizzata per verificare condizioni di input o esterne prima di eseguire un'azione nel contratto.
- Se la condizione booleana passata come argomento è falsa, la funzione causa il rollback di tutte le modifiche di stato effettuate fino a quel momento e produce un messaggio di errore opzionale specificato come secondo argomento.
- È utile per garantire che le condizioni di input o le precondizioni siano soddisfatte prima di eseguire un'operazione.

Il rollback è un'operazione che annulla o annulla le modifiche riportando allo stato precedente.

3. revert(string memory reason):

- Questa funzione viene utilizzata per interrompere l'esecuzione del contratto e provocare il rollback di tutte le modifiche di stato effettuate fino a quel momento.
- È possibile specificare un motivo opzionale per il rollback, che verrà visualizzato come un messaggio di errore.
- È utile per interrompere l'esecuzione del contratto in modo sicuro quando si verificano condizioni anomale o errori imprevisti.

L'uso di queste funzioni aiuta a garantire la sicurezza e la robustezza del contratto, consentendo di gestire correttamente situazioni indesiderate o errori durante l'esecuzione.

Solidity Modifier

In Solidity è possibile creare nuovi modificatori di funzione.
Le definizioni dei modificatori in Solidity sono porzioni di codice con una sintassi simile a quella delle funzioni e che vengono eseguite prima del corpo di ciascuna funzione modificata. Posso riutilizzare lo stesso modificatore in qualsiasi funzione, scrivendo il suo nome nella dichiarazione, dopo gli altri modificatori (ad esempio, public, pure, ecc.).
L'obiettivo è di abbreviare le funzioni e creare codice più corretto, ordinato e leggibile. Supponiamo di avere un contratto con molte funzioni che possono essere eseguite solo dall'indirizzo salvato nella variabile owner. Per ogni funzione dovrei scrivere una riga contenente il require.

```
require(msg.sender == owner, "accesso negato");
```

Invece di scrivere lo stesso require in tutte le funzioni, posso implementare un modificatore.

```
modifier onlyowner ()
{
   require(msg.sender == owner, "accesso negato");
   _;
}
```

Una volta che il modificatore unicoproprietario è stato definito, lo posso usare nella dichiarazione della funzione, ovvero:

```
function modifyValue(uint32 newValue) public onlyowner
{
// function body
}
```

FallBack

Il fallback è la funzione che viene eseguita quando si tenta di interagire con il contratto inviando un messaggio (il payload) non riconosciuto come una funzione e i suoi argomenti.
Ad esempio, il fallback può essere chiamato da remix scrivendo un messaggio esadecimale nell'interazione a basso livello. Nota: Lo stesso messaggio sarà disponibile all'interno di msg.data in Solidity.

Esempio: un fallback che può ricevere Ether e modificare due variabili

```
fallback() external payable
{
   x = 1;
   y = msg.value;
}
```

In questo esempio, il fallback viene eseguito quando il contratto riceve un messaggio non riconosciuto come una funzione. Viene impostata una variabile x a 1 e una variabile y al valore dell'Ether ricevuto durante l'interazione.

Receive

La funzione **receive** permette al contratto di ricevere ether senza aver inviato alcun messaggio. Di seguito è riportato un esempio di come implementare la funzione receive in Solidity:

```
contract Sink {
   event Received(address sender, uint amount);

   receive() external payable {
      emit Received(msg.sender, msg.value);
   }
}
```

In questo esempio, la funzione receive è dichiarata come external e payable, il che significa che può essere chiamata dall'esterno del contratto e può ricevere Ether. Quando il contratto riceve fondi senza un messaggio specifico, viene eseguita la funzione receive, che emette un evento Received per registrare l'indirizzo del mittente e l'importo ricevuto. Questo consente di monitorare facilmente i fondi ricevuti dal contratto.

Contracts creation

Un contratto intelligente può creare istanze di altri contratti utilizzando la parola chiave new seguita dal nome del contratto e, eventualmente, dagli argomenti del costruttore.

Ecco un esempio di come creare un'istanza di un contratto Deposit all'interno di un altro contratto:

```
pragma solidity ^0.8.0;

contract Deposit {
    address public owner;

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

contract AnotherContract {
    Deposit public depositInstance;

    constructor() {
        // Creazione di un'istanza del contratto Deposit
        depositInstance = new Deposit();
    }
}
```

In questo esempio, il contratto AnotherContract crea un'istanza del contratto Deposit utilizzando la sintassi new Deposit(). Questa istanza viene memorizzata in una variabile di stato depositInstance di tipo Deposit. Quando viene creato un nuovo contratto AnotherContract, viene anche creato automaticamente un'istanza del contratto Deposit e assegnato a depositInstance.

- **Factory Pattern**
  Per creare un nuovo contratto, è possibile utilizzare il pattern di fabbrica. Una fabbrica di contratti è progettata per distribuire istanze di contratto su richiesta.

```
contract Clone
{
    address public owner;
    constructor()
    {
        owner = address(tx.origin); //msg.sender;
    }
}

contract CloneFactory
{
    uint256 public number_Of_clones;
    mapping(uint => address) clones;
    function createClone() public returns(address aClone)
    {
        Clone c = new Clone();
        number_Of_clones += 1;
        clones[number_Of_clones] = address(c);
        return address(c);
    }
}
```

Lo statement di import permette di inserire il contenuto di un altro file nel proprio contratto.

```
import "nomeFile";
import * as simbolo from "nomeFile";
import "nomeFile" as simbolo;
```

Nei casi 2 e 3, le entità globali contenute nel file sono importate come: `simbolo.NomeEntità`.
È anche possibile importare una singola entità dal file.

```
import {simbolo1 as alias} from "nomeFile";
```

E da un URL (con REMIX):

```
import"https://github.com/OpenZeppelin/openzeppelincontracts/blob/master/contracts/access/Ownable.sol";
```

È possibile importare più file Solidity nel proprio contratto. In questo modo, è come se si inserisse tutto il codice già implementato nelle fonti importate nel proprio file.

```
import "./unFileSorgente.sol";
```

È anche possibile importare da GitHub:

```
import"https://github.com/OpenZeppelin/openzeppelincontracts/blob/master/contracts/token/ERC20/IERC20.sol";
```

- **Inheritance(Eredità)**

Ogni contratto può ereditare tutte le funzionalità o dichiarazioni di altri contratti (e librerie) dopo aver importato il codice. Solidity consente l'ereditarietà multipla.

Utilizziamo la parola chiave `is`.

```
import "./unFileSorgente.sol";

contract MioContratto is AltroContratto, UnTerzoContratto
{
   // corpo
}
```

Tutti i codici sorgente dei contratti ereditati devono essere disponibili per il compilatore (non vengono ereditati dalle istanze nella blockchain ma vengono ereditati a livello di codice).
Se il contratto A eredita da B, il codice di B viene copiato nel contratto A. A verrà creato e inserito nella blockchain.
Con "is" si ereditano tutte le variabili di stato e le funzioni interne e pubbliche.

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.8.17;
3  contract mySupercontract
4  {
5      uint internal i;
6      uint private priv;
7      uint public pub;
8  }
9
10 contract mySubcontract is mySupercontract
11 {
12     function set() public     44372 gas
13     {
14         i = 10;
15         //priv = 10; It is not visible
16         pub = 10;
17     }
18 }
```

- **Eredità & Costruttore**
  Quando il contratto da cui ereditiamo ha un costruttore siamo obbligati a chiamarlo (con i parametri appropriati) nel contratto derivato. Ci sono due modi principali per chiamare il costruttore.

  1. Durante la fase di dichiarazione del contratto, passando i valori al contratto da cui è ereditato per impostare i parametri in modo fisso.

```
contract DerivedConstructor is BaseConstructor(valori)
{
   // corpo
}
```

  2. Attraverso il costruttore del contratto derivato, lasciando il settaggio dei parametri al creatore del contratto derivato.

```
contract DerivedConstructor2 is BaseConstructor
{
   constructor(uint256 _numero) BaseConstructor(_numero) {}
}
```

Proxy Pattern

Nell'Ethereum, la creazione di un contratto e la ricreazione di una struttura dati contenente dati comportano costi significativi. Supponiamo di avere un contratto intelligente A utilizzato da altri contratti intelligenti e che contiene una quantità considerevole di dati nella sua memoria. Poiché non è possibile modificare il contratto A in alcun modo, in caso di bug o difetti, l'unica soluzione è installare un nuovo contratto A', copiando tutti i dati da A. Tuttavia, questa operazione sarebbe estremamente costosa in termini di consumo di gas. Inoltre, tutti i contratti intelligenti che utilizzano A dovrebbero aggiornare il proprio riferimento per puntare all'indirizzo di A', innescando una cascata di modifiche nell'intero sistema.
Il pattern del proxy è una soluzione a questo problema. Un proxy P è un contratto che tiene in memoria:

- L'indirizzo del contratto delegato A, e poi di A', A", ecc.
- L'indirizzo del proprietario, autorizzato a cambiare l'indirizzo menzionato

I contratti esterni inviano messaggi a P, che li redirige prima a A, e poi eventualmente a A', A", ecc.

In questo modo, sostituendo A con A', basta cambiare l'indirizzo all'interno di P, e tutti i contratti esterni continueranno a funzionare.

Questo può essere realizzato utilizzando:

- La funzione fallback, che consente di inviare una chiamata a P che P inoltrerà al delegato senza la firma della funzione appartenente all'ABI di P.
- Il delegatecall, che consente a P di passare il messaggio in modo che il delegato veda l'originale come chiamante e non P.

Poiché sostituire A con A' comporta la ricreazione di tutta la memoria di A, un'operazione molto costosa, è possibile mettere la memoria in P, lasciando la logica di controllo (da sostituire) in A, A'.

I messaggi includono l'indirizzo del SC destinatario, i dati sul gas, eventuali ETH inviati, il payload.

Il proxy P si limita a registrare l'indirizzo del SC e l'indirizzo del proprietario, l'unico autorizzato a cambiare l'indirizzo del SC a cui inoltrare il msg.

- Il fallback di P, utilizzando il codice assembly a basso livello di EVM:
  - Recupera il payload della chiamata fallita (che include i primi 4 byte dell'hash della firma della funzione)
  - Lo invia nuovamente come msg all'indirizzo del SC per cui è proxy, tramite un delegatecall
  - Recupera il payload di ritorno e lo restituisce all'indirizzo della chiamata iniziale

- **Proxy Upgradable**
  L'indirizzo del contratto "implementation" viene registrato su uno slot di memoria specifico (standard ERC 1967) ottenuto come:

  ```
  bytes32(uint256(keccak256('eip1967.proxy.implementation')) - 1)
  ```

  L'implementazione di base utilizza il codice a basso livello:

  ```
  bytes32 costante _IMPLEMENTATION_SLOT =
  0x360894a13ba1a3210667c828492db98dca3e2076cc3735a920a3ca505d382bbc;
  struct AddressSlot
  {
          address value;
  }
  funzione getAddressSlot(bytes32 slot) internal pure return (AddressSlot storage r)
  {
          assembly
          {
                  r.slot := slot
          }
  }
  ```

TOKEN

Un token è un'unità di valore digitale emessa su una blockchain. Può rappresentare asset digitali, diritti di accesso, partecipazioni a servizi o qualsiasi altra forma di valore. I token possono essere utilizzati per rappresentare una vasta gamma di asset fisici o virtuali e possono essere scambiati o trasferiti tra utenti sulla blockchain.

Gli ERC20 fungible tokens sono stati proposti da Fabian Vogelsteller e Vitalik Buterin nel 2015. Il loro standard è definito nell'Ethereum Improvement Proposal (EIP) 20. Questo standard stabilisce un insieme di regole e interfacce che consentono ai token di funzionare in modo interoperabile su blockchain compatibili con Ethereum.

I token ERC20 fungibili sono forniti sotto licenza MIT da OpenZeppelin. Questo contratto standard può essere utilizzato per creare e gestire token conformi allo standard ERC20 all'interno di un contratto intelligente su Ethereum.

Da notare che non esiste una funzione pubblica di mint all'interno del contratto standard ERC20. Tuttavia, è possibile aggiungere una funzione di mint personalizzata al proprio contratto ERC20 se necessario.

Esempio: Importiamo il contratto ERC20 da OpenZeppelin e aggiungiamo il costruttore al nostro contratto MyToken che estende ERC20.

```
import "https://github.com/OpenZeppelin/openzeppelin-
contracts/blob/master/contracts/token/ERC20/ERC20.sol";

contract MyToken is ERC20
{
   // corpo personalizzato
   constructor() ERC20("NomeToken", "SIMBOLO") {}
}
```

È possibile aggiungere una funzione di mint personalizzata o assegnare tutti i token al creatore del contratto, a seconda delle esigenze del progetto.

Esercizio

Import the ERC20 implementation into your code and create a contract called SchoolToken that calls the ERC20 constructor with the name "BCSCHOOL" and the symbol "BST". The contract implements a function called mint which takes as input an address and a uint256 and calls the _mint function of the basic contract (ERC20). Also implement a modifier to allow only the contract creator to mint new tokens.

```solidity
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.7;
import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol";

//definisco il contratto SchoolToken, che eredita dal contratto ERC20.
contract SchoolToken is ERC20
{
    address owner; //La variabile owner viene utilizzata per memorizzare l'indirizzo del proprietario del contratto.

    //definisce un errore personalizzato che verrà utilizzato per segnalare quando un utente che non è il proprietario cerca di eseguire un'operazione riservata
    //al proprietario.
    error notTheOwner(address _address);

    //il constructor serve per inizializzare lo stato del contratto, e per eseguire operazioni necessarie al momento della creazione
    constructor() ERC20("BCSCHOOL", "BST") //ERC20(--) chiama il costruttore del contratto ERC20, passando il nome e il simbolo del token    infinite gas 799600 gas
    {
        owner = msg.sender; // imposto l'indirizzo del proprietario come l'indirizzo di chi ha creato il contratto
    }

    //Assicura che solo il proprietario possa eseguire le funzioni contrassegnate con questo modificatore. Se l'indirizzo del chiamante non corrisponde
    //all'indirizzo del proprietario, viene emesso un errore personalizzato.
    modifier onlyOwner()
    {
        if (msg.sender != owner) revert notTheOwner(msg.sender);
        _;
    }

    //consente al proprietario di creare nuovi token e assegnarli a un destinatario specifico. La funzione può essere chiamata solo dal proprietario del
    //contratto (grazie al modificatore onlyOwner). La funzione utilizza la funzione _mint ereditata dal contratto ERC20
    function mint(address recipient, uint256 quantity) public onlyOwner    infinite gas
    {
        _mint(recipient,quantity);
    }
}
```

Standard ERC-721

Lo standard ERC-721 consente la gestione di token non fungibili (NFT) su Ethereum. È stato proposto da William Entriken, Dieter Shirley, Jacob Evans e Nastassia Sachs nel gennaio 2018.

Questo standard definisce tre eventi principali:

1. `Transfer(address _from, address _to, uint256 _tokenId)`: Questo evento viene emesso quando un token viene trasferito da un indirizzo `_from` a un indirizzo `_to`.

2. `Approval(address _owner, address _approved, uint256 _tokenId)`: Viene emesso quando un proprietario di token dà il permesso a un altro indirizzo `_approved` di trasferire un token che possiede.

3. `ApprovalForAll(address _owner, address _operator, bool _approved)`: Questo evento viene emesso quando un proprietario di token abilita o disabilita un operatore (ad esempio, un'app) per trasferire tutti i suoi token.

Ogni token ERC-721 è identificato univocamente da un numero intero uint256, e non possono essere creati due token con lo stesso numero.

Un token ERC-721 può avere un proprietario unico, che può possedere più token. Inoltre, il proprietario ha il potere di approvare o revocare il trasferimento di un singolo token o di tutti i suoi token a un indirizzo specifico.

È importante notare che la politica per la creazione di nuovi token (mint) dipende dall'implementazione specifica del contratto.

Example:
Similarly, we want to implement an ERC721 contract, adding the constructor.

```
import "https://github.com/OpenZeppelin/openzeppelin-contracts /blob /master /contracts
/token/ERC721/ERC721.sol";

contract MyToken is ERC721
{
        // custom body
        constructor() ERC20("theToken", "TTK") {}
        function mint(address recipient, uint256 quantity)
        public
        {
                _mint(recipient,quantity);
        }
}
```

Argomento aggiuntivo
NFT

Una NFT, o token non fungibile (Non-Fungible Token), è un tipo di token crittografico su blockchain che rappresenta la proprietà unica o la proprietà di un asset digitale o fisico. A differenza dei token fungibili, come ad esempio le criptovalute, che sono intercambiabili tra loro e hanno lo stesso valore, le NFT sono uniche e non possono essere sostituite con altre.

Le NFT sono utilizzate per rappresentare una vasta gamma di asset digitali, tra cui opere d'arte digitali, video, musica, giochi, collezionabili virtuali, proprietà immobiliari virtuali e molto altro ancora. Ogni NFT ha un identificativo univoco che la rende distinguibile da tutte le altre, consentendo agli utenti di confermare la proprietà e la provenienza dell'asset digitale.

Le NFT sono diventate particolarmente popolari nel mondo dell'arte digitale e della collezione digitale, poiché consentono agli artisti di autenticare e monetizzare le proprie opere d'arte digitali e ai collezionisti di possedere asset digitali unici e autenticati.