Access Control

- Aiutano a definire chi ha il permesso di fare cosa
- Aiutano a porre un controllo su chi accede ad un contratto

Vediamo come implementarlo:

Ownership e Ownable

- La realizzazione del concetto di controllo dell'accesso più semplice è data dall'idea di **proprietario** (**owner**) di un contratto.
- C'è un account che è il proprietario del contratto e che ha il potere di amministrarlo.
- Approccio utile nel caso di singolo user con funzioni di Admin

OpenZeppelin fornisce *Ownable* per implementare la ownership nei contratti

```
pragma solidity ^0.5.0;

import "@openzeppelin/contracts/ownership/Ownable.sol";

contract MyContract is Ownable {
    function normalThing() public {
        // anyone can call this normalThing()
    }

    function specialThing() public onlyOwner {
        // only the owner can call specialThing()!
    }
}
```

- Di default l'owner di un contratto Ownable è l'account che lo ha deployato, che è quello che spesso si vuole.
- Con **transferOwnership** si trasferisce l'ownership da un account ad un altro
- Con **renounceOwnership** per rinunciare all'ownership. Se un contratto non ha più un owner, tutti i metodi segnati con onlyOwner non possono essere più chiamati
- Un contratto può essere l'owner di un altro contratto

Role-Based Access Control

- Ci consente di definire dei livelli di autorizzazione
- RBAC ci consente di avere questa flessibilità
- Ogni ruolo ha un set di azioni che può compiere
- Invece che usare onlyOwner dappertutto adesso possiamo avere onlyAdminRole, onlyModeratorRole etc.
- Possiamo definire anche come vengono assegnati i ruoli agli account, come trasferire questi ruoli etc.
- E' un approccio molto diffuso

Usare AccessControl

OpenZeppelin fornisce *AccessControl* per implementare il controllo degli accessi basato su ruolo:
- Per ogni ruolo che vogliamo definire, creeremo un nuovo *role identifier* che è usato per garantire, revocare e controllare se un account ha quel ruolo.
- Ecco un esempio che fa uso dell'AccessControl in un token ERC20 per definire un ruolo 'minter' che permette agli account sotto questo ruolo, di creare nuovi tokens:

```
// contracts/MyToken.sol
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/access/AccessControl.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract MyToken is ERC20, AccessControl {
    // Create a new role identifier for the minter role
    bytes32 public constant MINTER_ROLE = keccak256("MINTER_ROLE");

    constructor(address minter) ERC20("MyToken", "TKN") {
        // Grant the minter role to a specified account
        _setupRole(MINTER_ROLE, minter);
    }

    function mint(address to, uint256 amount) public {
        // Check that the calling account has the minter role
        require(hasRole(MINTER_ROLE, msg.sender), "Caller is not a minter");
        _mint(to, amount);
    }
}
```

Tokens

- E' la rappresentazione di qualcosa nella blockchain.
- Soldi, tempo, servizi, etc.
- Rappresentare le cose come token permette agli smart contracts di potervi interagire, scambiarli, crearli e distruggerli.

Differenza tra token contract e token:
- Token contract: è uno smart contract ethereum. Inviare un token significa chiamare un metodo di uno smart contract che qualcuno ha scritto e deployato. Un token contract è un zapping tra indirizzi a saldi più qualche metodo per aggiungere e sottrarre Ester da questi saldi.
- Sono questi saldi che rappresentano i Token veri e propri. Diciamo che qualcuno ha Token quando il suo saldo nel token contract è diverso da 0.
- Questi saldi possono essere intesi come soldi, punti esperienza in un gioco, o qualsiasi cosa ed ognuno di questi token sarebbero memorizzati in diversi token contracts.

Tipi differenti di Token

- Fungible : beni fungible possono essere scambiati come l'ether. Importa QUANTI se ne hanno.
- Non fungible : sono beni unici e distinti. Importa QUALI si hanno.

Standards

Chiamati EIPx e ERCx e servono a documentare come uno smart contract può interoperare con altri smart contract.
- ERC20: lo standard di token più diffuso per beni fungibili.
- ERC721: la soluzione de-facto per token non fungibili. Usato per collezionismo e giochi
- ERC777: uno standard più potente per token fungibili. Retrocompatibile con ERC20

ERC20

Un token ERC20 tiene traccia dei token fungibili. Questo lo rende utile per modellare, voti, denaro etc.

Costruire un contratto ERC20 Token

- Usando i contratti possiamo creare il nostro token ERC20 che verrà usato per tenere traccia del Gold (oro, GLD), una valuta interna in un gioco ipotetico.
- Ecco come potrebbe essere fatto un GLD:

```
pragma solidity ^0.5.0;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20Detailed.sol";

contract GLDToken is ERC20, ERC20Detailed {
    constructor(uint256 initialSupply) ERC20Detailed("Gold", "GLD", 18) public {
        _mint(msg.sender, initialSupply);
    }
}
```

I nostri contratti vengono usati spesso attraverso l'ereditarietà e qui stiamo riutilizzando ERC20 per l'implementazione base standard e ERC20Detailed per ottenere le proprietà *name, symbol e decimals.* In più, possiamo creare una *InitialSupply* di tokens, che verrà assegnata agli indirizzi che deployano il contratto.

Una volta deployato, possiamo fare query al saldo di chi ha fatto il deplori del contratto

```
GLDToken.balanceOf(deployerAddress)
>1000
```

Possiamo anche trasferire i token ad altri accounts

```
GLDToken.trasfer(otherAddress,300)
GLDToken.balanceOf(otherAddress)
>300
GLDToken.balanceOf(deployerAddress)
>1000
```