Lesson 21 - Libraries / Token Standards

This Week

- Libraries / Token Standards / Utilities
- Auditing / Governance
- Upgrading Contracts
- Interoperability / Review

Adding Other Contracts and Libraries

When thinking about interacting with other contracts / libraries, it is useful to think of what happens at compile time, and what happens at runtime.

Compile time

If your contract references another contract or library, whether for inheritance, or for an external function call, the compiler needs to have the relevant code available to it.
You use the **import** statement to make the code available in your compilation file, alternatively you could copy the code into your compilation file it has the same effect.
Sometimes you need to gather all the contracts into one file, for example when getting you contract verified on etherscan. This process is known as flattening and there are plugins in Remix and Truffle to help with this.

If you inherit another contract, for example the Open Zeppelin Ownable contract, on compilation, the functions and variables from the parent contract (except those marked as private) are merged into your contract and become part of the resulting bytecode. From that point on the origin of the functions, are irrelevant.

Run time

There are 2 ways that your contract can interact with other deployed bytecode at run time.

1. External calls
   Your contract can make calls to other contract's functions during a transaction, to do so it needs to have the function signature available (this is checked at compile time) and the other contract's address available.

```
pragma solidity ^0.8.0;

contract InfoFeed {
    uint256 price;
    function info() public view returns (uint256 ret_) {
        return price;
        }
    // other functions
}

contract Consumer {
    InfoFeed feed;

    constructor(InfoFeed _feed){
        feed = _feed;
    }

    function callFeed() public view returns (uint256) {
        return feed.info();
    }
}
```

2. Using libraries

   A library is a type of smart contract that has no state, instead their functions run in the context of your contract.

See Documentation

For example we could use the Math library from Open Zeppelin
https://github.com/OpenZeppelin/openzeppelin-contracts/contracts/utils/math/Math.sol

```
We import it so that the compiler has access to the code
```

```
pragma solidity ^0.8.0;
import "https://github.com/OpenZeppelin/openzeppelin-contracts
/contracts/utils/math/Math.sol";

contract Test {
    using Math for uint256;


     function bigger(uint256 _a, uint256 _b) public pure
returns(uint256){
         uint256 big = _a.max(_b);
         return(big);
    }

}
```

The keyword ***using*** associates a datatype with our library, we can then use a variable of that datatype with the dot notation to call a library function

```
uint256 big = _a.max(_b);
```

You can reference already deployed libraries, at deploy time a linking process takes place which gives your contract the address of the library.
The the library has external or public functions these need to be linked to your contract at deploy time.
If the library functions are internal, they will be inlined into your contract at compile time.

Another useful library is String Utils

Abstract Contracts

See Docs

Contracts must be marked as abstract when at least one of their functions is not implemented or when they do not provide arguments for all of their base contract constructors.
Also a contract may be marked abstract, if you do not intend for the contract to be created directly.

Abstract contracts are similar to Interfaces but an interface is more limited in what it can declare.
Such abstract contracts can not be instantiated directly.
An abstract contract is declared using the `abstract` keyword .

```
pragma solidity 0.8.17;

abstract contract Dog {
    function bark() public virtual returns (bytes32);
}
contract Rover is Dog {
    function bark() public pure override returns (bytes32) {
          return "woof";
    }
}
```

Token Standards

ERC20

ERC20 is an example of a fungible token
It is a standard accepted by developers, exchanges and wallet creators
To be an ERC20 token, your contract must implement the following functions and events

```
FUNCTIONS

totalSupply()

balanceOf(account)

transfer(recipient, amount)

allowance(owner, spender)

approve(spender, amount)

transferFrom(sender, recipient, amount)
```

```
EVENTS

Transfer(from, to, value)

Approval(owner, spender, value)
```

Safe Functions:

Safe functions (SafeTransferFrom etc.) were introduced to prevent tokens being sent to contracts that didn't know how to handle them, and thus becoming stuck in the contract.

ERC721

ERC721 is an example of a non fungible token (NFT)
It has the following functions and events

```
FUNCTIONS
balanceOf(owner)
ownerOf(tokenId)
safeTransferFrom(from, to, tokenId)
transferFrom(from, to, tokenId)
approve(to, tokenId)
getApproved(tokenId)
setApprovalForAll(operator, _approved)
isApprovedForAll(owner, operator)
safeTransferFrom(from, to, tokenId, data)
```

```
EVENTS
Transfer(from, to, tokenId)
Approval(owner, approved, tokenId)
ApprovalForAll(owner, operator, approved)
```

ERC777

See proposal
An enhancement to ERC20, it allows operators to send tokens on behalf of another address .
It uses hooks, for send and receive to give more flexibility.

ERC1155 - Multi token standard

See Standard
ERC1155 is designed to be a **fungibility-agnostic** and **gas-efficient** token contract, allowing for a single contract to represent multiple fungible and non-fungible tokens, along with batched operations for increased gas efficiency.

In order to allow for multiple tokens to be represented in the contract the signatures of for example the balanceOf function needs a token ID as a parameter.

This allows gas savings in projects requiring multiple tokens, only one contract need be deployed.

Batching operations are also possible to allow multiple transfers within the same transaction.

The interface is

```
interface IERC1155 is IERC165 {

event TransferSingle(address indexed operator, address indexed
from, address indexed to, uint256 id, uint256 value);

event TransferBatch(
address indexed operator,
address indexed from,
address indexed to,
uint256[] ids,
uint256[] values
);

event ApprovalForAll(address indexed account, address indexed
operator, bool approved);

event URI(string value, uint256 indexed id);

function balanceOf(address account, uint256 id) external view
returns (uint256);

function balanceOfBatch(address[] calldata accounts, uint256[]
calldata ids) external view returns (uint256[] memory);

function setApprovalForAll(address operator, bool approved)
external;

function isApprovedForAll(address account, address operator)
external view returns (bool);

function safeTransferFrom(address from,address to,uint256
id,uint256 amount,bytes calldata data) external;

function safeBatchTransferFrom(address from,address to,uint256[]
calldata ids,uint256[] calldata amounts,bytes calldata data)
external;
```

Useful Open Source Collections

Open Zeppelin Token Contracts

Even though the concept of a token is simple, they have a variety of complexities in the implementation. Because everything in Ethereum is just a smart contract, and there are no rules about what smart contracts have to do, the community has developed a variety of standards (called EIPs or ERCs) for documenting how a contract can interoperate with other contracts.

- ERC20: the most widespread token standard for fungible assets, albeit somewhat limited by its simplicity.
- ERC721: the de-facto solution for non-fungible tokens, often used for collectibles and games.
- ERC777: a richer standard for fungible tokens, enabling new use cases and building on past learnings. Backwards compatible with ERC20.
- ERC1155: a novel standard for multi-tokens, allowing for a single contract to represent multiple fungible and non-fungible tokens, along with batched operations for increased gas efficiency.

Open Zeppelin Access Control / Security Contracts

- Ownable
- AccessControl
- TimeLockController
- Pausable
- Reentrancy Guard
- PullPayment

Open Zeppelin Governance Contracts

Implements on-chain voting protocols similar to Compound's Governor Alpha & Bravo

Open Zeppelin Cryptography Contracts

- ECDSA contract for checking signatures.
- MerkleProof for proving an item is in a Merkle tree.

Open Zeppelin Introspection Contracts

Contracts to allow runtime checks whether a target contract implements a particular interface

Open Zeppelin Maths Contracts

SafeMath - to prevent under / overflow etc. Some of this functionality is part of Solidity since version 0.8.0

Open Zeppelin Payment Contracts

- Payment splitter
- Escrow

Open Zeppelin Collections Contracts

- Enumerable Set
- Enumerable Map

Open Zeppelin Miscellaneous Contracts

- Address
- Multicall

Open Zeppelin Upgradability Contracts

- Proxy

We will cover upgradable contracts in a later lesson.

Useful maths libraries

Solmate
Prb-Math
WadMath
ABDK
Example use
Useful series about maths in Solidity