Solidity Part 2

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)
```

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)
```

Inheritence in Solidity

In object-oriented programming, inheritance is the mechanism of basing an object or class upon another object or class.
An object created through inheritance, a “child object”, acquires some or all of the properties and behaviors of the “parent object”

In Solidity we use the ***is*** keyword to show that the current contract is inheriting from a parent contract, for example here Destructible is the child contract and Owned is the parent contract.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;


contract Owned {
    constructor() { owner = payable(msg.sender); }
    address payable owner;
}


// Use `is` to derive from another contract. Derived
// contracts can access all non-private members including
// internal functions and state variables. These cannot be
// accessed externally via `this`, though.
contract Destructible is Owned {
    // The keyword `virtual` means that the function can change
    // its behaviour in derived classes ("overriding").
    function destroy() virtual public {
        if (msg.sender == owner) selfdestruct(owner);
    }
}
```

See Solidity Documentation (https://docs.soliditylang.org/en/v0.8.7/contracts.html?highlight=inheritence#inheritance)

Contract Components

Constructors

Every contract can be deployed with a `constructor` . It’s optional to use and can be useful for initialising the contract’s state i.e deploying an ERC20 contract with X tokens available.

The constructor is executed only when the contract is deployed.

Internal functions

The ERC20 contract has internal functions which are available to call i.e. `_mint`

Internal functions cannot be called externally. They are only visible in their own contract and its child contracts.

Enums

See documentation (https://docs.soliditylang.org/en/v0.8.7/types.html?highlight=string%20literal#enums)
The keyword Enum can be used to create a user defined enumerations, similar to other languages.
For example

```
1
2       enum ActionChoices { GoLeft, GoRight, GoStraight, SitStill }
3       ActionChoices choice;
4       ActionChoices constant defaultChoice =
5       ActionChoices.GoStraight;
6
```

Storage, memory and calldata

See documentation (https://docs.soliditylang.org/en/v0.8.7/types.html?highlight=calldata#data-location)

Storage

Storage data is permanent, forms part of the smart contract's state and can be accessed across all functions. Storage data location is expensive and should be used only if necessary. The `storage` keyword is used to define a variable that can be found in storage location.

Memory

Memory data is stored in a temporary location and is only accessible within a function. Memory data is normally used to store data temporarily whilst executing logic within a function. When the execution is completed, the data is discarded. The `memory` keyword is used to define a variable that is stored in memory location.

Calldata

Calldata is the location where external values from outside a function into a function are stored. It is a non-modifiable and non-persistent data location. The `calldata` keyword is required to define a variable stored in the calldata location.

The difference between calldata and memory is subtle, calldata variables cannot be changed.
For example :

```
1
2  pragma solidity ^0.8.0;
3
4  contract Test {
5
6  function memoryTest(string memory _exampleString)
7  public pure
8  returns (string memory) {
9      _exampleString = "example";  // You can modify memory
10     string memory newString = _exampleString;
11     // You can use memory within a function's logic
12     return newString;  // You can return memory
13 }
14
15 function calldataTest(string calldata _exampleString) external
16 pure returns (string calldata) {
17     // cannot modify _exampleString
18     // but can return it
19     return _exampleString;
20 }
21 }
```

Constant and Immutable variables

State variables can be declared as constant or immutable. In both cases, the variables cannot be modified after the contract has been constructed. For constant variables, the value has to be fixed at compile-time, while for immutable, it can still be assigned at construction time.

It is also possible to define constant variables at the file level.

```
// define a constant a file level
uint256 constant X = 32**22 + 8;

contract C {
    string constant TEXT = "abc";
    bytes32 constant MY_HASH = keccak256("abc");
    uint256 immutable decimals;
    uint256 immutable maxBalance;
    address immutable owner = msg.sender;

    constructor(uint256 _decimals, address _reference) {
        decimals = _decimals;
        // Assignments to immutables can even access the environment.
        maxBalance = _reference.balance;
    }
```

Interfaces

Interfaces in Solidity work the same way as in other languages.
The interface specifies the function signatures, but the implementation is specified in child contracts.
Use the ***interface*** keyword to declare an interface
For example

```
interface DataFeed {
function getData(address token) external returns (uint value);
}
```

Fallback and Receive functions

receive() external payable { ... }
Called when the contract receives ether

fallback () external [payable]
Called if a function cannot be found matching the required function signature.
It also handles the case when ether is received but there is no receive function

Checking inputs and dealing with errors

require / assert / revert / try catch

See Error handling (https://docs.soliditylang.org/en/v0.8.7/control-structures.html?highlight=require#error-handling-assert-require-revert-and-exceptions)
"The **require** function either creates an error without any data or an error of type **Error(string)**.
It should be used to ensure valid conditions that cannot be detected until execution time.
This includes conditions on inputs or return values from calls to external contracts."
Example

```
require(_amount > 0,"Amount must be > 0");
```

The **assert** function creates an error of type **Panic(uint256)**.
Assert should only be used to test for internal errors, and to check invariants.
Properly functioning code should never create a Panic, not even on invalid external input.
Example

```
assert(a>b);
```

The ***revert*** statement acts like a throw statement in other languages and causes the EVM to revert.
The require statement is ofen used in its place.
It can take a string as an error message, or a Error object.
For example

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract VendingMachine {
    address owner;
    error Unauthorized();
    function buy(uint amount) public payable {
        if (amount > msg.value / 2 ether)
            revert("Not enough Ether provided.");
        // Alternative way to do it:
        require(
            amount <= msg.value / 2 ether,
            "Not enough Ether provided."
        );
        // Perform the purchase.
    }
    function withdraw() public {
        if (msg.sender != owner)
            revert Unauthorized();

        payable(msg.sender).transfer(address(this).balance);
    }
}
```

try / catch statements can be used to catch errors in calls to external contracts.

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

interface DataFeed {
function getData(address token) external returns (uint value);
}

contract FeedConsumer {
    DataFeed feed;
    uint errorCount;
    function rate(address token) public
    returns (uint value, bool success) {
        // Permanently disable the mechanism if there are
        // more than 10 errors.
        require(errorCount < 10);
        try feed.getData(token) returns (uint v) {
            return (v, true);
        } catch Error(string memory /*reason*/) {
            // This is executed in case
            // revert was called inside getData
            // and a reason string was provided.
            errorCount++;
            return (0, false);
        } catch Panic(uint /*errorCode*/) {
            // This is executed in case of a panic,
            // i.e. a serious error like division by zero
            // or overflow. The error code can be used
            // to determine the kind of error.
            errorCount++;
            return (0, false);
        } catch (bytes memory /*lowLevelData*/) {
            // This is executed in case revert() was used.
            errorCount++;
            return (0, false);
        }
    }
}
```

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 hapens at runtime.

Compile time

If your contract references another contract or library, whether for inheritence, 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 (https://docs.soliditylang.org/en/latest/contracts.html#libraries)

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

Useful Open Source Collections

About Open Zeppelin:

Open Zeppelin are well known in the Ethereum community. They provide a set of audited smart contracts and libraries that are a standard in the industry.

Inheriting these contracts will provide a significantly higher degree of security and robustness in your code.

See https://docs.openzeppelin.com/contracts/4.x/

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.

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.

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

Importing from Github in Remix

See Documentation (https://remix-ide.readthedocs.io/en/latest/import.html)
You can import directly from github or npm

```
import "https://github.com/OpenZeppelin/openzeppelin-contracts
/contracts/access/Ownable.sol";
```

or

```
import "@openzeppelin/contracts@4.2.0/token/ERC20/ERC20.sol";
```

Exercises

Make your contract industry standard with Open Zeppelin

1. Change your Volcano coin to inherit from Open Zeppelin's Ownable contract, removing any functions, variables and modifiers that are no longer needed.

2. Change your contract to also inherit from the Open Zeppelin ERC20 contract.
   You have a wide choice of ERC20 contracts in Open Zeppelin to choose from, but ERC20.sol is enough for this exercise.
   Note that the parent contract constructor takes 2 parameters (a string token name and string symbol as an input) therefore we need to put these right after `ERC20` in the inheritence declaration
   For example

```
1  contract ExtropyERC20 is ERC20("Extropy Coin", "EXT") {
```

3. Remove any functions from your contract that are supplied by the parent contracts, and override functions where you have different functionality.

4. ERC20 has an internal function `_mint` . When called, it mints token to the recipient. Create a constructor that calls the `_mint` function inside the constructor.

5. Mint the 10000 token supply to the owner.

6. Make a function that can mint more tokens to the owner. It should mint the latest tokenSupply amount to the owner.
   From the perspective of the potential investors in your token, what would they think of this functionality being available ?

7. Recompile and redeploy your contract. Try minting tokens to the owner, and play around with sending tokens from the owner to other accounts in the javascript VM.
   Is the payment record tracking all the payments ?

8. Deploy your token to Rinkeby and send tokens to your colleagues.

We now want to enrich the data we are holding in the payment records, and add extra functionality to our coin.
We will want to allow users to add details to the payments in a second transaction after it has completed.

Payments can be of 5 types

- Unknown
- Basic payment
- Refund
- Dividend

- Group payment

15. Add fields to the payment record for

- a payment identifier, this should be unique but human readable
- a timestamp
- the type of payment (see above)
- a comment

17. Update the transfer function to fill in these fields, initially the payment type will be Unknown, and the comment field blank.
18. Write a new function to allow a user to view all the details for payments they have made
19. Write a function to allow the user to update a particular payment with the payment type and comment, they will need to specify the identifier,payment type and comment.
20. Make sure you check the parameters in these functions for validity.
21. Create a address variable for an administrator, this will be set at deploy time.
22. The administrator should have the ability to update the type of payment for any payment record.
    Write a function, or change the existing one, to allow this.
23. If the administrator updates a payment record the existing comment should have “updated by” plus the administrators address appended to it.

Optional Tasks

You have been asked to add delayed payment functionality, such that the token is transfered at a specified time upto 24hrs in the future
Is this possible to implement in a contract ?
If so describe how you would do this.
If not, why not ?

Resources

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol (https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol)
https://docs.soliditylang.org/en/v0.8.7/index.html (https://docs.soliditylang.org/en/v0.8.7/index.html)