Lesson 3 - Solidity

Solidity Part 2

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 = msg.sender; }
    address  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 Child1 is Owned {
    // The keyword `virtual` means that the function can change
    // its behaviour in derived classes ("overriding").
    function doThings()) virtual public {
        .... ;
    }
}
```

See Solidity Documentation

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

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

Further datatypes

See Documentation

Boolean

`bool`: The possible values are constants `true` and `false`.

Byte Arrays

Can be fixed size or dynamic
For fixed size : `bytes1, bytes2, bytes3, …, bytes32` are available
For dynamic arrays use : `bytes`

bytes.concat function

A recent change (0.8.4)
You can concatenate a variable number of bytes or bytes1 ... bytes32 using bytes.concat. The function returns a single bytes memory array
The simplest way to concatenate strings is now
`bytes.concat(bytes(s1), bytes(s2))`
where s1 and s2 are defind as string

string

Dynamically-sized UTF-8-encoded string
string is equal to bytes but does not allow length or index access.

Enums

See documentation
The keyword Enum can be used to create a user defined enumerations, similar to other languages.
For example

```
    enum ActionChoices { GoLeft, GoRight, GoStraight, SitStill }
    // we can then create variables
    ActionChoices choice;
    ActionChoices constant defaultChoice =
ActionChoices.GoStraight;
```

Storage, memory and calldata

See documentation

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 :

```
pragma solidity ^0.8.0;

contract Test {

function memoryTest(string memory _exampleString)
public pure
returns (string memory) {
    _exampleString = "example";  // You can modify memory
    string memory newString = _exampleString;
    // You can use memory within a function's logic
    return newString;  // You can return memory
}

function calldataTest(string calldata _exampleString) external
pure returns (string calldata) {
    // cannot modify _exampleString
    // but can return it
    return _exampleString;
}
}
```

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
"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

> 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.