Developing Contracts on Starknet

Cairo programs are by default stateless, if we want to write contracts to run on Starknet we need additional context provided by the StarknetOS

We need to add the following to our code to declare it as a contract

```
# Declare this file as a StarkNet contract.
%lang starknet
```

Adding state

To store state within our contract we use a decorator

```
@storage_var
```

We then specify the variable as a function

```
@storage_var
func balance() -> (res : felt):
end
```

The decorator will create the ability for us to read and write to a variable 'balance'
For example

```
balance.read()
```

```
balance.write(1234)
```

You would then write getter and setter functions to interact with the variable as usual.
When the contract is deployed, all the storage is set to 0

Function visibility

Unlike a standalone cairo program, we don't have a main function, instead we can interact with any of the functions in the contract depending on their visibility.

The decorators are
`@external`
`@view`

Both external and view functions can be called from other contracts or externally.
If no decorator is specified, then the function is internal.
Although @view indicates that we are not changing state, this is not currently enforced.

Function implicit arguments

```
@view
func get_score{
    syscall_ptr : felt*,
    pedersen_ptr : HashBuiltin*,
    range_check_ptr,
}() -> (score : felt):
    let (score) = score.read()
    return (score)
end
```

We have an additional implicit argument `syscall_ptr` this allows the OS to handle the storage variables correctly.

Hints

The situation regarding hints is a little more complex, see Hints
You generally do not use hints in your contracts due to security concerns (the user and the operator running the code are likely to be different).
You may see hints in libraries however, and there is a whitelisting mechanism to ensure security.

Constructors

Constructors work in a similar way to Solidity constructors.

- It must be called `constructor`
- It is decorated with `@constructor`
- It is run once, only during deployment.

For example

```
@constructor
func constructor{
    syscall_ptr : felt*,
    pedersen_ptr : HashBuiltin*,
    range_check_ptr,
}(owner_address : felt):
    owner.write(value=owner_address)
    return ()
end
```

More complex storage variables

We can create the equivalent of a Solidity mapping as follows

```
@storage_var
func balance(user : felt) -> (res : felt):
end
```

The corresponding read and write functions have the following signatures

```
func read{
        syscall_ptr : felt*, range_check_ptr,
        pedersen_ptr : HashBuiltin*}(
    user : felt) -> (res : felt)

func write{
        syscall_ptr : felt*, range_check_ptr,
        pedersen_ptr : HashBuiltin*}(
    user : felt, value : felt)
```

Getting the user address

The common.syscalls library allows us to get the address of the user calling a function

```
from starkware.starknet.common.syscalls import get_caller_address

# ...

let (caller_address) = get_caller_address()
```

For reference this is the default contract created by protostar

```
%lang starknet

from starkware.cairo.common.math import assert_nn
from starkware.cairo.common.cairo_builtins import HashBuiltin

@storage_var
func balance() -> (res : felt):
end

@external

func increase_balance{syscall_ptr : felt*,
```

```
pedersen_ptr : HashBuiltin*, range_check_ptr}(

amount : felt):
with_attr error_message("Amount must be positive. Got: {amount}."):

assert_nn(amount)

end



let (res) = balance.read()
balance.write(res + amount)
return ()

end



@view
func get_balance{syscall_ptr : felt*,
pedersen_ptr : HashBuiltin*, range_check_ptr}() -> (
res : felt):

let (res) = balance.read()
return (res)

end

@constructor
func constructor{syscall_ptr : felt*,
pedersen_ptr : HashBuiltin*, range_check_ptr}():

balance.write(0)
return ()

end
```

Other Useful Libraries

1. Math.cairo
   This gives us many useful mathematical and assertion functions such as

- assert_nn
- abs_value
- unsigned_div_rem
- sqrt

2. Signature.cairo
   This gives us the function

```
func verify_ecdsa_signature{ecdsa_ptr : SignatureBuiltin*}(

message, public_key, signature_r, signature_s

):
```

Allowing us to verify that the prover knows a signature of the given public_key on the given message.

An example of its use see docs

```
@external
func increase_balance{
    syscall_ptr : felt*,
    pedersen_ptr : HashBuiltin*,
    range_check_ptr,
    ecdsa_ptr : SignatureBuiltin*,
}(user : felt, amount : felt, sig : (felt, felt)):
    # Compute the hash of the message.
    # The hash of (x, 0) is equivalent to the hash of (x).
    let (amount_hash) = hash2{hash_ptr=pedersen_ptr}(amount, 0)

    # Verify the user's signature.
    verify_ecdsa_signature(
        message=amount_hash,
        public_key=user,
        signature_r=sig[0],
        signature_s=sig[1],
    )
```

3. Common Library

- alloc.
- bitwise.
- cairo_builtins.
- default_dict.
- dict.
- dict_access.
- find_element.
- set.

You can import these using for example

```
from starkware.cairo.common.bitwise import bitwise_operations
```

Account Abstraction

See docs from Nethermind
Also this blog from Ethereum and this blog from Gnosis

Why Account Abstraction matters

The shift from EOAs to smart contract wallets with arbitrary verification logic paves the way for a series of improvements to wallet designs, as well as reducing complexity for end users. Some of the improvements Account Abstraction brings include:

- Paying for transactions in currencies other than ETH
- The ability for third parties to cover transaction fees
- Support for more efficient signature schemes (Schnorr, BLS) as well as quantum-safe ones (Lamport, Winternitz)
- Support for multisig transations
- Support for social recovery

Previous solutions relied on centralized relay services or a steep gas overhead, which inevitably fell on the users' EOA.

EIP-4337 is a collaborative effort between the Ethereum Foundation, OpenGSN, and Nethermind to achieve Account Abstraction in a user-friendly, decentralized way.

Account Abstraction on Starknet

An account is a contract address.
With abstract accounts, it's the **who** (address) that matters, not the **how** (signature).

A user, will still have an address that can be shared publicly so that someone can send a token to that address.
The user still has a wallet with private keys that are used to sign transactions.

The difference is that the address will be a contract. The contract can contain any code. Below is an example of a stub contract that could be used as an account. The user deploys this contract and calls `initialize()`, storing the public key that their wallet generated.

```
A minimalist Account contract ###

# This function is called once to set up the account contract.

@external
func initialize(public_key):
    store_public_key(public_key)
    return ()
end

# This function is called for every transaction the user makes.
@external
func execute(destination, transaction_data, signature):
    # Check the signature used matches the one stored.
    check_signature(transaction_data, signature)
    # Increment the transaction counter for safety.
    increase_nonce()
    # Call the specified destination contract.
    call_destination(destination, transaction_data)
    return ()
end
```

Then to transfer a token, they put together the details (token quantity, recipient) and sign the message with their wallet. Then `execute()` is called, passing the signed message and intended destination (the address of the token contract).

A user can define what they want their account to "be". For many users, an account contract will perform a signature check and then call the destination.

However, the contract may do anything:

- Check multiple signatures.

- Receive and swap a stable coin before paying for the transaction fee.
- Only agree to pay for the transaction if a trade was profitable.
- Use funds from a mixer to pay for the transaction, separating the deposit from withdrawl.
- Send a transaction on behalf of itself - a DAO.

If you wish to explore this more, this is a useful workshop explaining account abstraction.

Developing contracts

An equivalent to the Cairo playground is the Starknet Playground
https://starknet.io/playground/?lesson=starknet_contract

Tools

Protostar

Lesson 5 has details of how to install protostar, we can also use this to develop contracts.

Testing in protostar

See the notes from Lesson 5, in additio the following flags may be useful

```
protostar test test/test_erc20.cairo --disable-hint-validation
protostar test test/test_ex1.cairo --stdout-on-success
```

Cheatcodes

See documentation

There are a number of keywords to add functionality to your tests, such as mocking, deploying contracts. We will cover these in more detail later.