Lesson 7 - Cairo / Starknet continued

Deploying contracts

If you are using the starknet CLI you can use
`starknet deploy --contract compiled_contract.json --address SELECTED_ADDRESS`

If you are using protostar, see docs
`protostar deploy ./build/main.json --network alpha-goerli`

Comparison operations

There is ambiguity about the functions to use for comparison, see this issue
Open Zeppelin have therefore created 2 libraries
Safe_cmp
and
FeltMath

Namespaces

To allow modularity in our contracts we have the namespace keyword

```
namespace encode  {
        func homework1(a: felt, b:felt) -> (c: felt){
                return (a*b);
        }
}
```

We can then reference this function as

```
encode.homework1(11, 13);
```

Contract Classes

A recent addition to starknet, since version 0.9
From medium article
"Taking inspiration from object-oriented programming, we distinguish between the contract code and its implementation. We do so by separating contracts into classes and instances."

The way it works is similar to the proxy pattern in Ethereum.

A contract class is the definition of the contract: Its Cairo bytecode, hint information, entry point names, and everything necessary to unambiguously define its semantics. Each class is identified by its class hash.

A contract instance, is a deployed contract corresponding to some class. Note that only contract instances behave as contracts, i.e., have their own storage and are callable by transactions/other contracts.
A contract class does not necessarily have a deployed instance in StarkNet.

The declare transaction type declares a class but does not deploy an instance of that class.

The deploy system call takes 3 arguments

- The class hash
- Salt
- Constructor arguments

This will deploy a new instance of the contract whose address depends on the above arguments, this is similar to the CREATE2 op code on Ethereum.

Contract Extensibility

See Forum post

Currently

- Cairo has no explicit smart contract extension mechanisms such as inheritance or composability
- There's no function overloading making function selector collisions very likely – more so considering selectors do not take function arguments into account
- Any `@external` function defined in an imported module will be automatically re-exposed by the importer (i.e. the smart contract)
- Builtins cannot be imported more than once in the entire imports hierarchy, resulting in errors on import (or errors on compilation if not added) – and most contracts will need the same common set of builtins such as `pedersen`, `range_check`, etc.

Contracts and libraries

Libraries define behavior and storage while contracts build on top of libraries.
Contracts can be deployed – libraries cannot.

Guidelines from Open Zeppelin when using libraries, see tips from Nethermind below

Considering the following types of functions:

- `private`: private to a library, not meant to be used outside the module or imported
- `public`: part of the public API of a library
- `internal`: subset of `public` that is either discouraged or potentially unsafe (e.g. `_transfer` on ERC20)
- `external`: subset of `public` that is ready to be exported as-is by contracts (e.g. `transfer` on ERC20)
- `storage`: storage variable functions

Then:

- Must implement `public` and `external` functions under a namespace
- Must implement `private` functions outside the namespace to avoid exposing them
- Must prefix `internal` functions with an underscore (e.g. `ERC20._mint`)
- Must not prefix `external` functions with an underscore (e.g. `ERC20.transfer`)
- Must prefix `storage` functions with the name of the namespace to prevent clashing with other libraries (e.g. `ERC20balances`)
- Must not implement any `@external`, `@view`, or `@constructor` functions
- Can implement initializers (never as `@constructor` or `@external`)
- Must not call initializers on any function

Namespaces allow us to better distinguish between four types of library functions and how to approach each of them for secure development:

Tips and best practices

There are some useful tips here
Some items from the coding guideline from Nethermind

Split the contract into a logic file and a contract file

- A library file (or logic file), named `my_contract_library.cairo`, contains the logic code of the contract. Namely, it contains: (i) internal and external functions encapsulated in a namespace, and (ii) storage variables and events defined outside the namespace.
- A contract file, named `my_contract.cairo`, exposes external functions from its corresponding library file and other library files. For instance, an implementation of cToken that inherits from an ERC20 contract would expose both functions in `c_token_library.cairo` and `erc20_library.cairo`.

Error messages

Use `with_attr error_message(...)` as shown in yesterday's notes, make sure only one thing can fail in the block.

Passing arrays in calldata

To pass an array of felt to a function, the usual pattern is to pass a pointer of felt and the array’s length. We recommend encapsulating this array in a struct `MyStruct` and use `MyStruct.SIZE` for the array length.

Recursion

For each loop, we recommend defining an internal function suffixed with `_inner` or `_loop` to do the job.

```
func sum_array(array_len : felt, array : felt*) -> felt {
    let sum = 0;
    let (res) = _sum_array_inner{array_len=array_len, array=array, sum=sum}
(0);
    return res;
}



func _sum_array_inner{array_len : felt, array : felt*, sum : felt}
(current_index : felt) -> felt {
    if (current_index - array_len == 0) {
        return (sum);
    }
    let sum = sum + array[current_index];
    return _sum_array_inner(current_index + 1);
}
```

Variable names

To avoid collisions, prefix variable names with the namespace that was specified with the `namespace` keyword

```
// in my_contract_library.cairo
@storage_var
func MyContract_name() {
}

namespace MyContract {
    ...
}
```

Calling functions in other contracts

You can call external functions in other contracts, but to do this you need to provide an interface, for this we use the `@contract_interface` decorator.
The body of the function and implicit arguments are not needed.

For example

```
@contract_interface
namespace IBalanceContract {
    func increase_balance(amount: felt) {
    }

    func get_balance() -> (res: felt) {
    }
}
```

This can be called from another contract as follows.
We need to pass the contract address as an additional argument.

```
@external
func call_increase_balance{syscall_ptr: felt*, range_check_ptr}(
    contract_address: felt, amount: felt
) {
    IBalanceContract.increase_balance(
        contract_address=contract_address, amount=amount
    );
    return ();
}
```

Starknet architecture

See this article for a good overview

Full Nodes

These run the Pathfinder client to keep a record of all the transactions performed in the rollup and to track the current global state of the system.
Full Nodes receive this information through a p2p network where changes in the global state and the validity proofs associated with it are shared everytime a new block is created.
When a new Full Node is set up it is able to reconstruct the history of the rollup by connecting to an Ethereum node and processing all the L1 transactions associated with StarkNet.

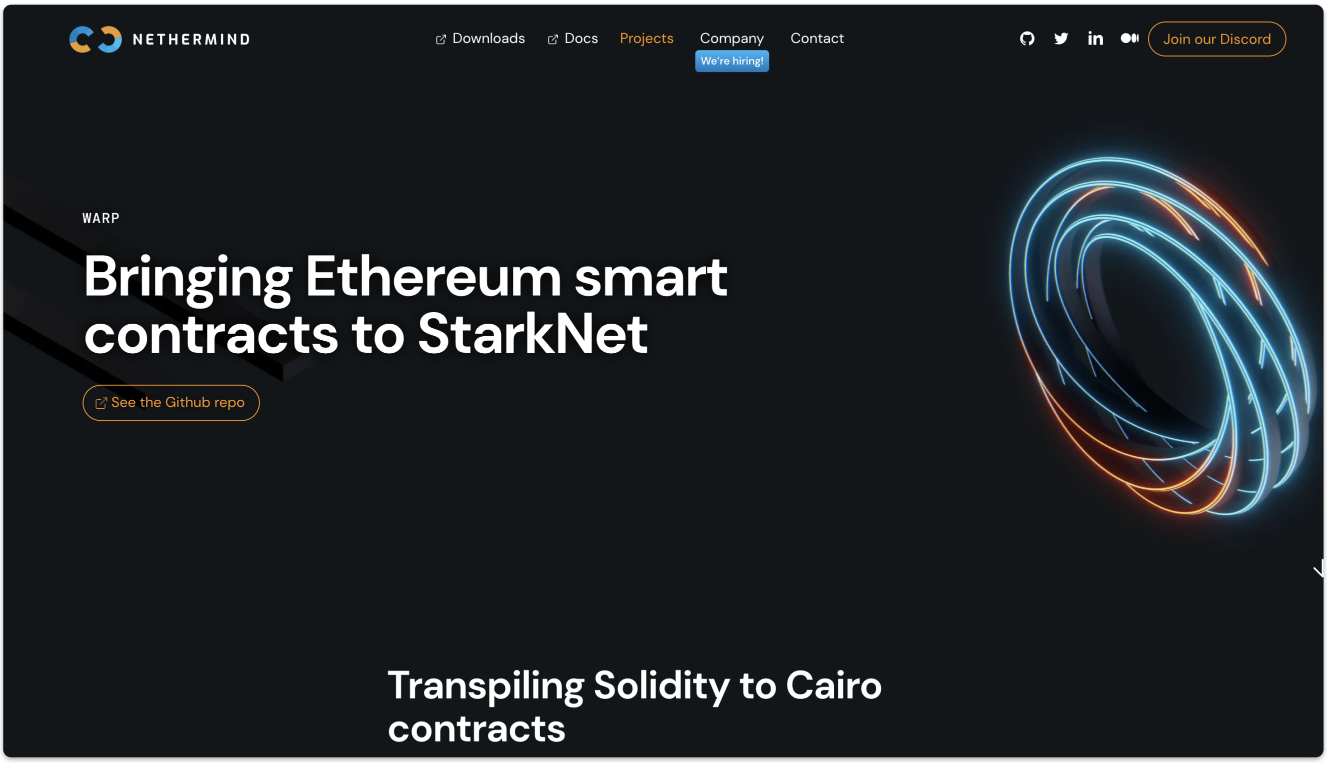
Transaction lifecycle

Transactions start in the NOT_RECEIVED state.
They are passed via the L2 network to the full nodes and the sequencer, where they are added to the mempool.

When the transaction is added to the sequencers mempool the state is changed to RECEIVED.
The transactions are added to a pending (L2) block, once there, the transation state is changed to PENDING.
When enough transactions have been added the block is executed by the sequencer / VM, if the transaction completes successfully its state is changed to ACCEPTED_ON_L2 otherwise it is changed to REJECTED

The sequencer and prover then create the proof and sen this along with the transaction data to L1, once the proof is validated, the transaction status changes to ACCEPTED_ON_L1.
If validation fails, it will be marked as REJECTED

Warp



WARP

Bringing Ethereum smart contracts to StarkNet

Transpiling Solidity to Cairo contracts

Warp allows you transpile Solidity contracts into Cairo

Installation Instructions

See Warp installation instructions

1. On macos:

```
brew install z3
```

2. On ubuntu:

```
sudo apt install libz3-dev
```

Make sure that you have the `venv` module for your python installation.

Installation

Without any virtual environment activated run the following in order:

```
yarn global add @nethermindeth/warp
```

Run the following to see the version and that it was installed:

```
warp version
```

Finally run the following to install the dependencies:

`warp install`

Test installation works by transpiling an example ERC20 contract:

`warp transpile example_contracts/ERC20.sol`

Using Docker

`docker build -t warp .`

`docker run --rm -v $PWD:/dapp --user $(id -u):$(id -g) warp transpile example_contracts/ERC20.sol`

Using Warp

`warp transpile example_contracts/ERC20.sol`

`warp transpile example_contracts/ERC20.sol --compile-cairo`

You can then deploy your cairo code to the network, with the following commands you need to specify the network, in our case alpha-goerli

```
warp deploy test.json --network alpha-goerli
```

```
Deploy transaction was sent.
Contract address:
0x0403bd2f0abdd765398d6a50ff89cfe9ac48760f3b94ba2728bfbacdaff9f59a
Transaction hash:
0x32ca42d1341703cc957845ea53a71b3eb2e762ff148cb9dc522322eede94b65
```

You can invoke a transaction on your contract

```
warp invoke --program test.json --address
0x0403bd2f0abdd765398d6a50ff89cfe9ac48760f3b94ba2728bfbacdaff9f59a  --network
alpha-goerli --function store --inputs [13]
```

```
Invoke transaction was sent.
Contract address:
0x0403bd2f0abdd765398d6a50ff89cfe9ac48760f3b94ba2728bfbacdaff9f59a
Transaction hash:
0x1d1ec8278ccf41452737e80a54e7626299e598528363ced7a527d810f9d6881
```

And check the status

```
warp status 0x1d1ec8278ccf41452737e80a54e7626299e598528363ced7a527d810f9d6881
--network alpha-goerli
```

which will give a answer similar to

```
        {
                "block_hash":
"0x1c55254f16d087f0bf7776183c4d38549680e68600394167f304f1afe5a035e",
                "tx_status": "ACCEPTED_ON_L1"
        }
```

You should be able to see the details on the block explorer
Voyager Block Explorer

There is also now a vyper transpiler