

Lesson 21

Storage Layout

Examples of how different datatypes are stored, including dynamically sized items are illustrated on the [sol2uml site](#)

Command

```
sol2uml storage ./src/Counter.sol --contract Counter
```

Examples

BNB on Ethereum

BNB <<Contract>> 0xB8c77482e45F1F44dE1745F52C74426C631bDD52		
slot	type: <inherited contract>.variable (bytes)	
0	string: name (32)	
1	string: symbol (32)	
2	unallocated (31)	uint8: decimals (1)
3	uint256: totalSupply (32)	
4	unallocated (12)	address: owner (20)
5	mapping(address=>uint256): balanceOf (32)	
6	mapping(address=>uint256): freezeOf (32)	
7	mapping(address=>mapping(address=>uint256)): allowance (32)	

Contract Code

```
contract BNB is SafeMath{
    string public name;
    string public symbol;
    uint8 public decimals;
    uint256 public totalSupply;
    address public owner;

    /* This creates an array with all balances */
    mapping (address => uint256) public balanceOf;
    mapping (address => uint256) public freezeOf;
```

```
mapping (address => mapping (address => uint256)) public
allowance;
```

Staking / Farming









Staking

Strictly speaking staking involves providing a stake, for yourself or as a delegatee which is used to participate in the consensus mechanism. As a result you get a (share of) the block reward.

	Risk ⓘ	Complexity ⓘ	Reward ⓘ	Adj. Reward ⓘ	Minimum ⓘ	Lock Up ⓘ	Avg. Fee ⓘ	Stake Share ⓘ
Delegate BNB	low ⓘ	easy ⓘ	2.7% ⓘ	8.26% ⓘ	1 ⓘ	7 d ⓘ	6.86% ⓘ	97.82% ⓘ
Run A Validator Node	medium ⓘ	professional ⓘ	2.9% ⓘ	8.48% ⓘ	10,000 ⓘ	7 d ⓘ	- ⓘ	2.18% ⓘ

Show More

These are some of the validators.

Validators					
Validator	Voting Power / % ⓘ	Commission ⓘ	APR ⓘ	Status	Action
 Avengers cabinet	1,268,966.71207785 / 5.36%	10%	2.32%	Active	Delegate
 48 Club cabinet	1,212,327.28761957 / 5.12%	10%	1.66%	Active	Delegate
 Synclub cabinet	957,872.25880996 / 4.04%	5%	2.04%	Active	Delegate
 Claude Shannon cabinet	950,532.99611101 / 4.01%	10%	3.18%	Active	Delegate
 MathWallet cabinet	927,022.80702505 / 3.91%	5%	0.87%	Active	Delegate
 Legend II cabinet	923,451.94044577 / 3.9%	7%	2.58%	Active	Delegate
 Tranchess cabinet	918,335.79272914 / 3.88%	7%	2.41%	Active	Delegate
 CertiK cabinet	912,638.98498165 / 3.85%	5%	2.85%	Active	Delegate

Yield Farming

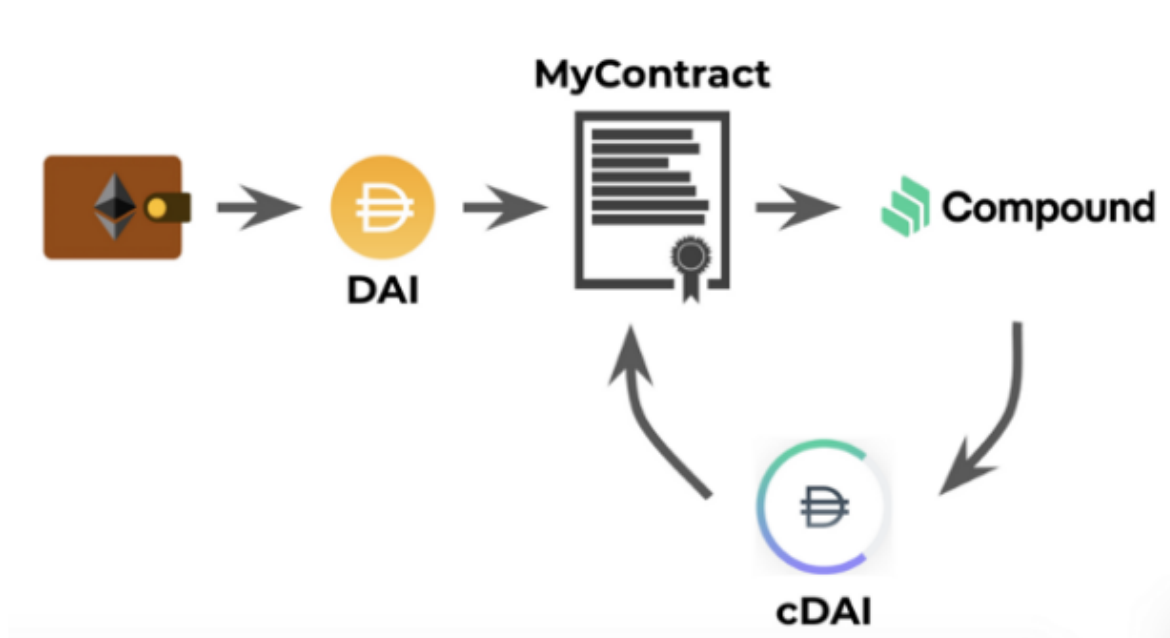
Yield farming is the process of providing liquidity to DeFi protocols such as liquidity pools and borrowing and lending services. It offers rewards in the form of interest, with a portion of transaction fees given to the yield farmer.

Compound

Compound [Documentation](#)

[Guide](#) to interacting with compound

Example adding DAI to compound



The process flow is

- You **transfer** Dai from your wallet to your custom contract. This is not done in Solidity, but instead with Web3.js and JSON RPC.
- You call your custom contract's function for supplying to the Compound Protocol.
- Your custom contract's function calls the approve function from the original ERC20 token contract. This allows an amount of the token to be withdrawn by cToken from your custom contract's token balance.
- Your custom contract's function calls the **mint** function in the Compound cToken contract.
- Finally, we call your custom contract's function for redeeming, to get the ERC20 token back.

The mint function transfers an asset into the protocol, which begins accumulating interest based on the current **Supply Rate** for the asset. The user receives a quantity of cTokens equal to the underlying tokens supplied, divided by the current **Exchange Rate**.

CErc20

```
1 function mint(uint mintAmount) returns (uint)
```

- `msg.sender`: The account which shall supply the asset, and own the minted cTokens.
- `mintAmount`: The amount of the asset to be supplied, in units of the underlying asset.
- **RETURN**: 0 on success, otherwise an **Error code**

Before supplying an asset, users must first **approve** the cToken to access their token balance.

cTokens

From the [documentation](#)

Each asset supported by the Compound Protocol is integrated through a cToken contract, which is an [EIP-20](#) compliant representation of balances supplied to the protocol. By minting cTokens, users

- (1) earn interest through the cToken's exchange rate, which increases in value relative to the underlying asset, and
- (2) gain the ability to use cTokens as collateral.

cTokens are the primary means of interacting with the Compound Protocol; when a user mints, redeems, borrows, repays a borrow, liquidates a borrow, or transfers cTokens, she will do so using the cToken contract.

There are currently two types of cTokens: CErc20 and CEther. Though both types expose the EIP-20 interface, CErc20 wraps an underlying ERC-20 asset, while CEther simply wraps Ether itself. As such, the core functions which involve transferring an asset into the protocol have slightly different interfaces depending on the type.

This is the interface to the cToken contract that you can use

```
interface CErc20 {  
  
    function mint(uint256) external returns (uint256);  
    function exchangeRateCurrent() external returns (uint256);  
}
```

```

function supplyRatePerBlock() external returns (uint256);
function redeem(uint) external returns (uint);
function redeemUnderlying(uint) external returns (uint);

}

```

Decimals for stablecoins

cToken	cToken Decimals	Underlying	Underlying Decimals
cETH	8	ETH	18
cAAVE	8	AAVE	18
cBAT	8	BAT	18
cCOMP	8	COMP	18
cDAI	8	DAI	18
cLINK	8	LINK	18
cMKR	8	MKR	18
cSUSHI	8	SUSHI	18
cTUSD	8	TUSD	18
cUNI	8	UNI	18
cUSDC	8	USDC	6
cUSDP	8	USDP	18
cUSDT	8	USDT	6

Calculating APY (General)

<https://javascript.plainenglish.io/how-to-calculate-apy-for-any-lp-token-using-javascript-in-four-simple-steps-956d8543a239>

Calculating APY (Compound)

Calculating the APY Using Rate Per Block

The Annual Percentage Yield (APY) for supplying or borrowing in each market can be calculated using the value of `supplyRatePerBlock` (for supply APY) or `borrowRatePerBlock` (for borrow APY) in this formula:

```
Rate = cToken.supplyRatePerBlock(); // Integer
Rate = 37893566
ETH Mantissa = 1 * 10 ^ 18 (ETH has 18 decimal places)
Blocks Per Day = 6570 (13.15 seconds per block)
Days Per Year = 365

APY = (((Rate / ETH Mantissa * Blocks Per Day + 1) ^ Days Per Year)) - 1) * 100
```

Here is an example of calculating the supply and borrow APY with Web3.js JavaScript:

```
const ethMantissa = 1e18;
const blocksPerDay = 6570; // 13.15 seconds per block
const daysPerYear = 365;

const cToken = new web3.eth.Contract(cEthAbi, cEthAddress);
const supplyRatePerBlock = await
cToken.methods.supplyRatePerBlock().call();
const borrowRatePerBlock = await
cToken.methods.borrowRatePerBlock().call();
const supplyApy = (((Math.pow((supplyRatePerBlock / ethMantissa
* blocksPerDay) + 1, daysPerYear))) - 1) * 100;
const borrowApy = (((Math.pow((borrowRatePerBlock / ethMantissa
* blocksPerDay) + 1, daysPerYear))) - 1) * 100;
console.log(`Supply APY for ETH ${supplyApy} %`);
console.log(`Borrow APY for ETH ${borrowApy} %`);
```

Arbitrage and Liquidation Bots

See our medium [article](#) for a discussion of arbitrage bots, but note this code is out of date.

From Compound liquidation bot [article](#)

A liquidation occurs when the value of your borrowed assets (**borrowing balance**) is greater than your borrowing capacity (**collateral factor**). This occurs when your collateral drops in value or when the borrowed asset rises too high in value.

If this event occurs arbitrageurs will be competing to liquidate your position. Liquidations are necessary to eliminate risks to the protocol. Liquidators repay up to 50% of the assets borrowed and in return they are eligible to a portion of your collateral at the market price minus a liquidation discount of 5% or more. The liquidation process will continue until the value of your borrowing is back below your borrowing capacity (**collateral factor**).

The Compound protocol does not rely on a central entity to perform liquidations. The Compound liquidation process relies on outside 3rd party systems to keep the protocol in balance. It incentivizes decentralized participants to participate (liquidate the collateral, repay the borrowed funds, and in return receive the collateral in another asset with some discount). Any Ethereum user can be a liquidator.

Maths Libraries

There are a number of maths libraries available to give some 'floating point' functionality to Solidity. See this [article](A useful article to give background [article] series (<https://medium.com/coinmonks/math-in-solidity-part-1-numbers-384c8377f26d>)) for an overview.

PRB Maths Library

See [repo](#)

See [article](#)

Smart contract library for advanced fixed-point math that operates with signed 59.18-decimal fixed-point and unsigned 60.18-decimal fixed-point numbers.

Features

- Operates with signed and unsigned denary fixed-point numbers, with 18 trailing decimals
- Offers advanced math functions like logarithms, exponentials, powers and square roots
- Gas efficient, but still user-friendly
- Bakes in overflow-safe multiplication and division
- Reverts with custom errors instead of reason strings
- Well-documented via NatSpec comments
- Thoroughly tested with Hardhat and Waffle

PRBMath comes in four flavours: basic signed, typed signed, basic unsigned and typed unsigned.

You get these functions

- Absolute
- Arithmetic and geometric average
- Exponentials (binary and natural)
- Floor and ceil
- Fractional
- Inverse
- Logarithms (binary, common and natural)

- Powers (fractional number and basic integers as exponents)
- Multiplication and division
- Square root

In addition, there are getters for mathematical constants:

- Euler's number
- Pi
- Scale (1e18, which is 1 in fixed-point representation)

Installation

With yarn:

```
yarn add @prb/math
```

Or npm:

```
npm install @prb/math
```

Other Libraries

Uniswap [Full Maths](#)

Potential problems and best practices

Flash Loan Attacks and Oracle Manipulation

Flash loans can be used to submit large volumes to a project, and can be used maliciously to move the price of an asset.

See previous notes for details of Oracle manipulation and our medium [article](#)

Volume Limiting

Projects often limit the size of trades that they will accept when a project initially goes live, to mitigate the effect of any bugs.

Simulate transactions

Tools such as [Tenderly](#) can be used to simulate transactions covering a range of contexts for your project.

MEV

MEV Mitigation

Pre Merge approach - Flashbots auction

Flashbots Auction provides a private communication channel between Ethereum users and miners for efficiently communicating preferred transaction order within a block.

Flashbots Auction consists of [mev-geth](#), a patch on top of the go-ethereum client, along with the [mev-relay](#), a transaction bundle relayer.

Proposer/block builder separation (PBS)

(Status: In development not implemented/Not a solved problem)

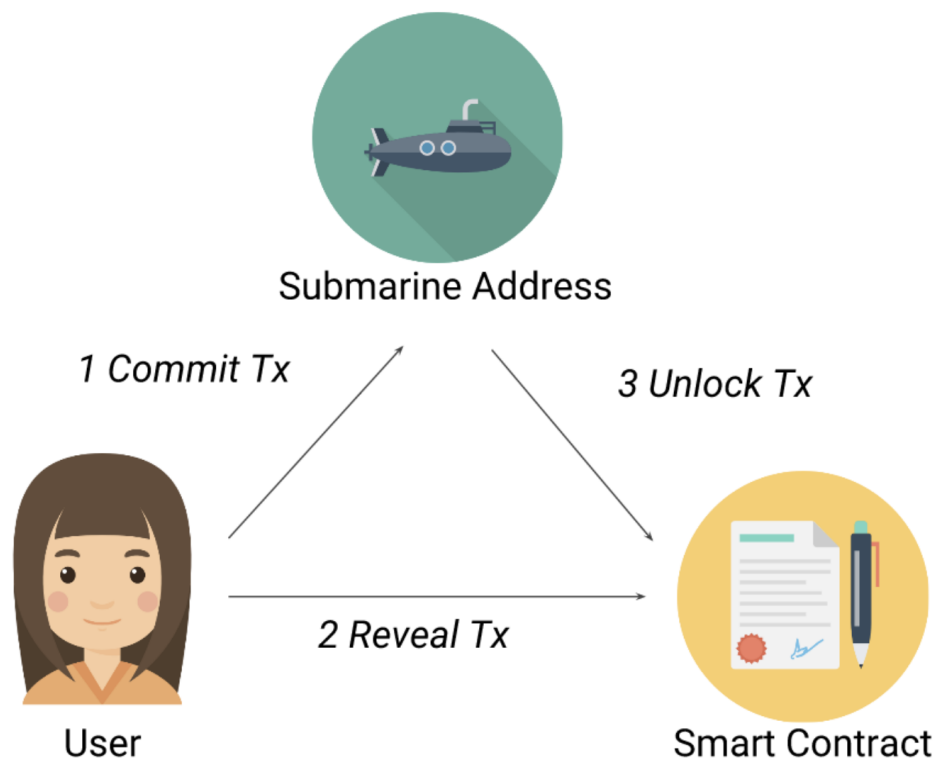
TL;DR: Split the roles of validators/builders so that the centralisation forces of MEV are contained in builder layer.

- Builders are highly specialized actors that construct candidate blocks and make bids to get them included
- Proposers are validators that naively accept the highest bid
- Goal: builders absorb economies of scale, proposers stay decentralized

Submarine Sends

See [Submarine sends](#)

This is a commit reveal scheme can be used (possibly with other obfuscating transactions)



Attempts to trick bots

Salmonella

See [Repo](#)

Salmonella intentionally exploits the generalised nature of front-running

setups. The goal of sandwich trading is to exploit the slippage of unintended victims, so this strategy turns the tables on the exploiters. It's a regular ERC20 token, which behaves exactly like any other ERC20 token in normal use-cases. However, it has some special logic to detect when anyone other than the specified owner is transacting it, and in these situations it only returns 10% of the specified amount - despite emitting event logs which match a trade of the full amount.

DeFi protocol changes

1. Protocols can whitelist bots that will share the MEV from arbitrage or liquidation with the user being front run
 2. Enhancements to the DeFi protocols, for example with Request For Quote (RFQ) orders to specify which address will have their order filled, or progressive liquidations from [Euler](#)
-

Protocol Approaches

Some protocols have taken steps to prevent MEV such as using Verifiable Delay Functions in order to prevent gaming of transaction ordering, as [Solana has done on its base layer](#) to ensure transactions are ordered by time of arrival, or simply delegate ordering to something like Chainlink's Fair Sequencing Service (FSS)

Fair Sequencing Service

See [Blog](#)

"In a nutshell, the idea behind FSS is to *have an oracle network order the transactions sent to a particular contract SC*, including both user transactions and oracle reports.

[Oracle](#) nodes ingest transactions and then reach consensus on their ordering, rather than allowing a single leader to dictate it. Oracle nodes then forward the transactions to the contract *SC*. They sequence these transactions by attaching nonce or sequence numbers to them or sending them in batches."

You can read more about Fair Sequencing Services in the Chainlink 2.0 White paper (Section5): <https://research.chain.link/whitepaper-v2.pdf>

5.2 FSS Details

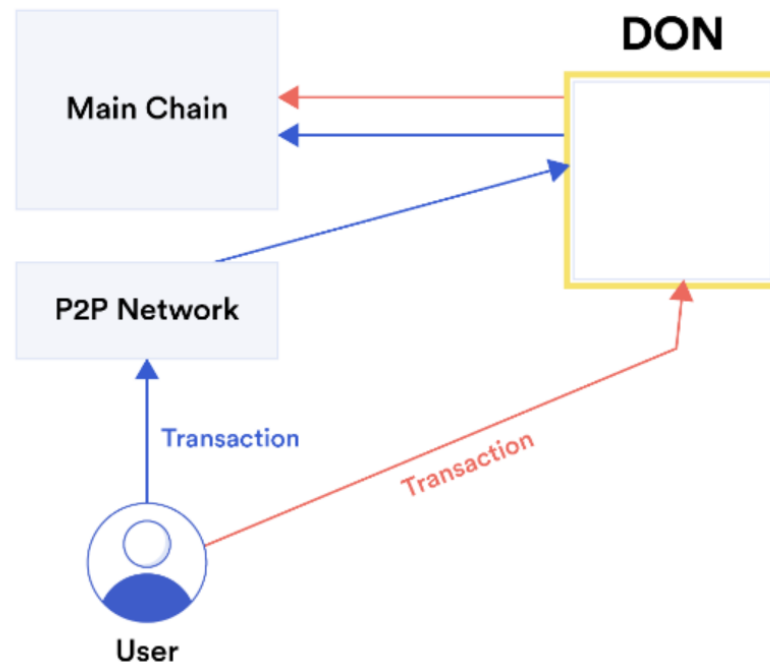


Figure 12: Order-fair mempool with two different transaction paths: direct and mempool-based.

1. Direct: The direct approach is conceptually simplest, but requires changes to user clients so that transactions are sent directly to the Decentralized Oracle Network nodes, rather than to the nodes of the main chain. The DON collects user transactions destined to a specific smart contract SC and orders them based on some ordering policy. The DON then sends the ordered transactions to the smart contract on the main chain. Some ordering mechanisms also require the direct approach because the user that creates a transaction must cryptographically protect it before sending it to FSS.
2. Mempool-based: To facilitate the integration of FSS with legacy clients, the DON can use Mempool Services (MS) to monitor the main chain's mempool and collect transactions.

Oracle Extractable Value (OEV)

OEV is the value that can be extracted from the system by the address that updates an Oracle.

For example imagine an Oracle is providing price data to a DeFi lending protocol, if there is a sudden price change, then some positions could be at risk of liquidation.

The address updating the Oracle could exploit their advanced knowledge of the price change to submit a liquidation transaction.

The pros and cons of OEV pretty much follow that for MEV, it can be viewed as a good or a bad thing, and as unavoidable.

Monitoring the mempool

From Quick Node guide

Example code :

```
var ethers = require("ethers");
var url = "ADD_YOUR_NODE_WSS_URL";

var init = function () {
  var customWsProvider = new
  ethers.providers.WebSocketProvider(url);

  customWsProvider.on("pending", (tx) => {
    customWsProvider.getTransaction(tx).then(function
    (transaction) {
      console.log(transaction);
    });
  });

  customWsProvider._websocket.on("error", async () => {
    console.log(`Unable to connect to ${ep.subdomain}
    retrying in 3s...`);
    setTimeout(init, 3000);
  });

  customWsProvider._websocket.on("close", async (code) => {
    console.log(
      `Connection lost with code ${code}! Attempting
      reconnect in 3s...`
    );
    customWsProvider._websocket.terminate();
    setTimeout(init, 3000);
  });
};
```



```
init();
```

Example Sandwich Bot Contract

(We do not encourage people to run these types of bots)

See [Repo](#)

Overview

From the README

"In every Uniswap V2 trade, the user (victim) will specify a minimum amount of output tokens they're willing to receive.

The job of the sandwich bot is to calculate how much of the output tokens they should buy (to push the price of the token up) to match the victim's minimum out requirement. This minimum out requirement on most cases will be 2%, but on extreme cases it can be as high as 20% on volatile pairs (such as the SHIBA-WETH pair during the craze).

Once the sandwich bot has calculated the optimal number of tokens to buy, it'll wait for the victim to buy their tokens, and immediately sell to gain a profit."

The bot

1. Gets transactions from the mempool that are for the Uniswap router
2. Checks they don't have a receipt.
3. Parse the Uniswap data for 'swapExactETHForToken' transactions
4. Calculate a profitable sandwich
5. Work out what fees are likely to be involved
6. Submit the slices of the sandwich

Interestingly, the front and back slices are submitted via the Flashbots network.

Account Abstraction

See [docs](#) from Nethermind

Also this [blog](#) from Ethereum and this [blog](#) from Gnosis

See this [article](#) from Binance Academy

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 transactions
- Support for social recovery

Previous solutions relied on centralised 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, decentralised way.

Features

- **Achieve the key goal of account abstraction:** allow users to use smart contract wallets containing arbitrary verification logic instead of EOAs as their primary account. Completely remove any need at all for users to also have EOAs (as status quo SC wallets and [EIP-3074](#) both require)
- **Decentralization**
 - Allow any bundler (think: block builder) to participate in the process of including account-abstracted user operations

- Work with all activity happening over a public mempool; users do not need to know the direct communication addresses (eg. IP, onion) of any specific actors
- Avoid trust assumptions on bundlers
- **Do not require any Ethereum consensus changes:** Ethereum consensus layer development is focusing on the merge and later on scalability-oriented features, and there may not be any opportunity for further protocol changes for a long time. Hence, to increase the chance of faster adoption, this proposal avoids Ethereum consensus changes.
- **Try to support other use cases**
 - Privacy-preserving applications
 - Atomic multi-operations (similar goal to [EIP-3074](#))
 - Pay tx fees with [ERC-20](#) tokens, allow developers to pay fees for their users, and [EIP-3074](#)-like **sponsored transaction** use cases more generally
 - Support aggregated signature (e.g. BLS)

Instead of transactions, users send 'UserOperation' objects into a separate mempool, then actors called 'bundlers' package these up as transactions to a special contract.

The entry point's `handleOps` function must perform the following steps. It must make two loops, the **verification loop** and the **execution loop**. In the verification loop, the `handleOps` call must perform the following steps for each `UserOperation`:

- **Create the account if it does not yet exist**
- **Call `validateUserOp` on the account**, passing in the `UserOperation`, the required fee and aggregator (if there is one).

There is a certain synergy here with some of the MEV solutions.